

6.172
Performance
Engineering
of Software
Systems



LECTURE 5
C to Assembly

Tao B. Schardl

Review: Why Assembly?

Why look at the assembly of your program?

- Assembly is more precise than C and can reveal program details such as type-cast operations and usage of registers and memory.
- The assembly reveals what the compiler did and did not do, e.g., to optimize basic operations.
- Bugs can arise at a low level. For example, a bug in the code might only have an effect when compiling at `-O3`. Bugs might also be caused by the compiler!
- You can modify the assembly by hand to make it run fast.
- **Reverse engineering**: You can decipher what a program does when you only have access to its binary.

Where We Stand

Lecture 4: Computer Architecture

- Basics of the x86–64 assembly language: instructions, registers, data types, memory addressing modes, the RFLAGS register, and condition codes.

This lecture:

- How C code is implemented in x86–64 assembly.

How Does C Code Become Assembly?

Source

bitarray.c

Preprocess

Preprocessed
source

bitarray.i

Compile

Assembly

bitarray.s

The compiler does **a lot** of stuff to translate C code to assembly:

- Choose assembly instructions to implement C operations.
- Implement C conditionals and loops using jumps and branches.
- Choose registers and memory locations to store data.
- Move data among the registers and memory to satisfy dependencies.
- Coordinate function calls.
- Try to make the assembly fast.

Example: fib.s

As a result, the mapping from C to assembly is **not** always obvious.

C code fib.c

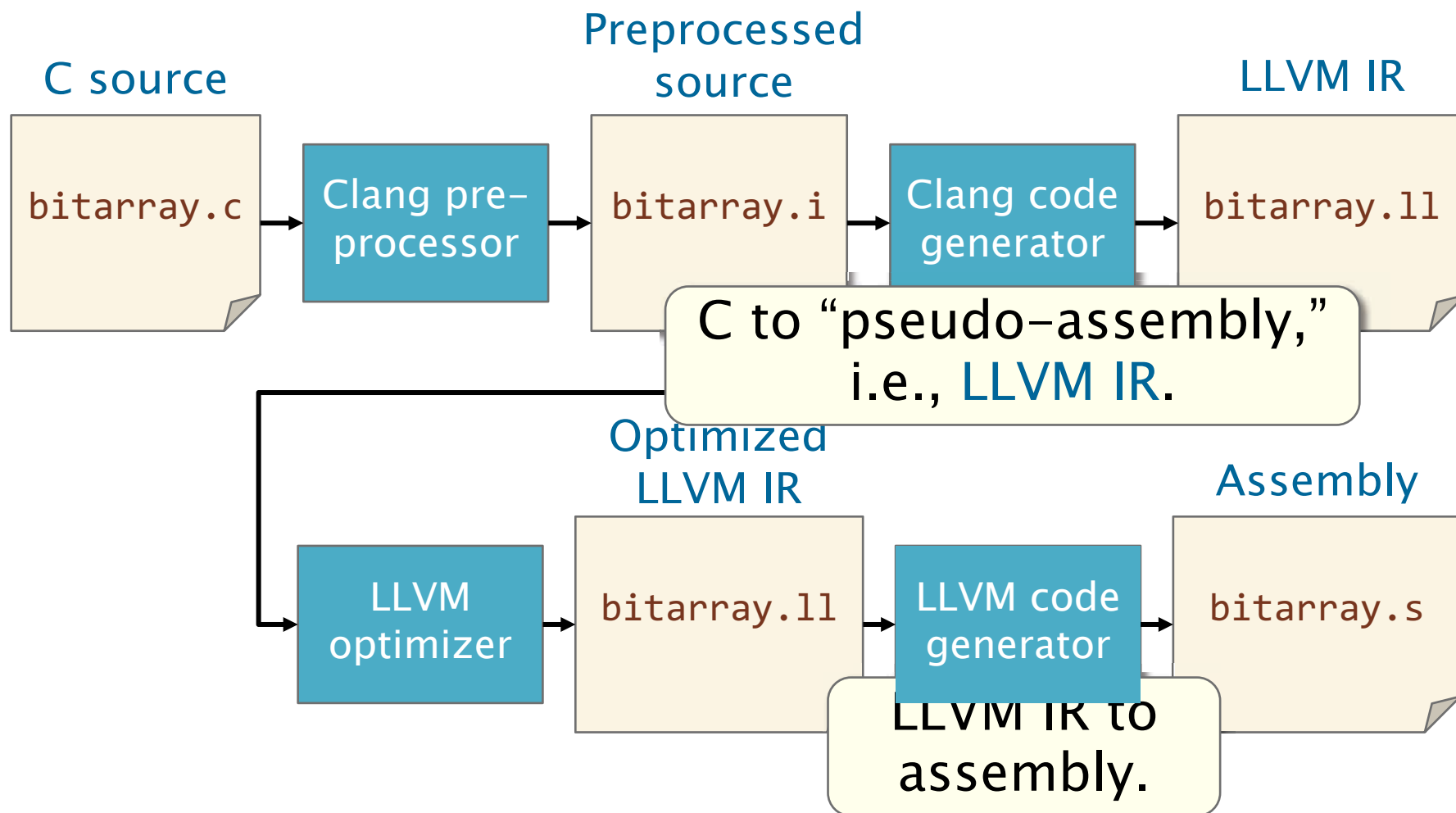
```
int64_t fib(int64_t n) {  
    if (n < 2)  
        return n;  
    return (  
        fib(n-1)  
        +  
        fib(n-2));  
}
```

Assembly code fib.s

```
.globl    _fib  
.p2align  4, 0x90  
_fib:  
    pushq  %rbp  
    movq   %rsp, %rbp  
    pushq  %r14  
    pushq  %rbx  
    movq   %rdi, %rbx  
    cmpq   $2, %rbx  
    jge    LBB0_1  
    movq   %rbx, %rax  
    jmp    LBB0_3  
LBB0_1:  
    leaq  -1(%rbx), %rdi  
    callq _fib  
    movq  %rax, %r14  
    addq  $-2, %rbx  
    movq  %rbx, %rdi  
    callq fib  
    addq  %r14, %rax  
LBB0_3:  
    popq  %rbx  
    popq  %r14  
    popq  %rbp  
    retq
```

Clang/LLVM Compilation Pipeline

To understand this translation process, let us see how the **compiler** reasons about it.



Viewing LLVM IR

You can see what the `clang` compiler does by looking at the LLVM IR.

Source code `fib.c`

```
int64_t fib(int64_t n) {  
    if (n < 2) return n;  
    return (fib(n-1) + fib(n-2));  
}
```

```
$ clang -O3 fib.c \  
> -S -emit-llvm
```

Clang flags:

- “-S” produces assembly.
- “-S -emit-llvm” produces LLVM IR.

LLVM IR code `fib.ll`

```
define i64 @fib(i64) local_unnamed_addr #0 {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:                                ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:                                ; preds = %1  
    ret i64 %0  
}
```

Compiling LLVM IR

LLVM IR can be translated directly into assembly.

LLVM IR code fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:                                ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:                                ; preds = %1  
  ret i64 %0  
}
```

```
$ clang fib.ll -S
```

Assembly code fib.s

```
.globl    _fib  
.p2align  4, 0x90  
_fib:  
## @fib  
pushq    %rbp  
movq     %rsp, %rbp  
pushq    %r14  
pushq    %rbx  
movq     %rdi, %rbx  
cmpq     $2, %rbx  
jge     LBB0_1  
movq     %rbx, %rax  
jmp      LBB0_3  
  
LBB0_1:  
leaq     -1(%rbx), %rdi  
callq   _fib  
movq     %rax, %r14  
addq     $-2, %rbx  
movq     %rbx, %rdi  
callq   _fib  
addq     %r14, %rax  
  
LBB0_3:  
popq     %rbx  
popq     %r14  
popq     %rbp  
retq
```


Outline

LLVM IR PRIMER

C TO LLVM IR

- STRAIGHT-LINE C CODE TO LLVM IR
- C FUNCTIONS TO LLVM IR
- C CONDITIONALS (E.G., IF-THEN-ELSE) TO LLVM IR
- C LOOPS TO LLVM IR
- LLVM IR ATTRIBUTES

LLVM IR TO ASSEMBLY

- LINUX X86-64 CALLING CONVENTION

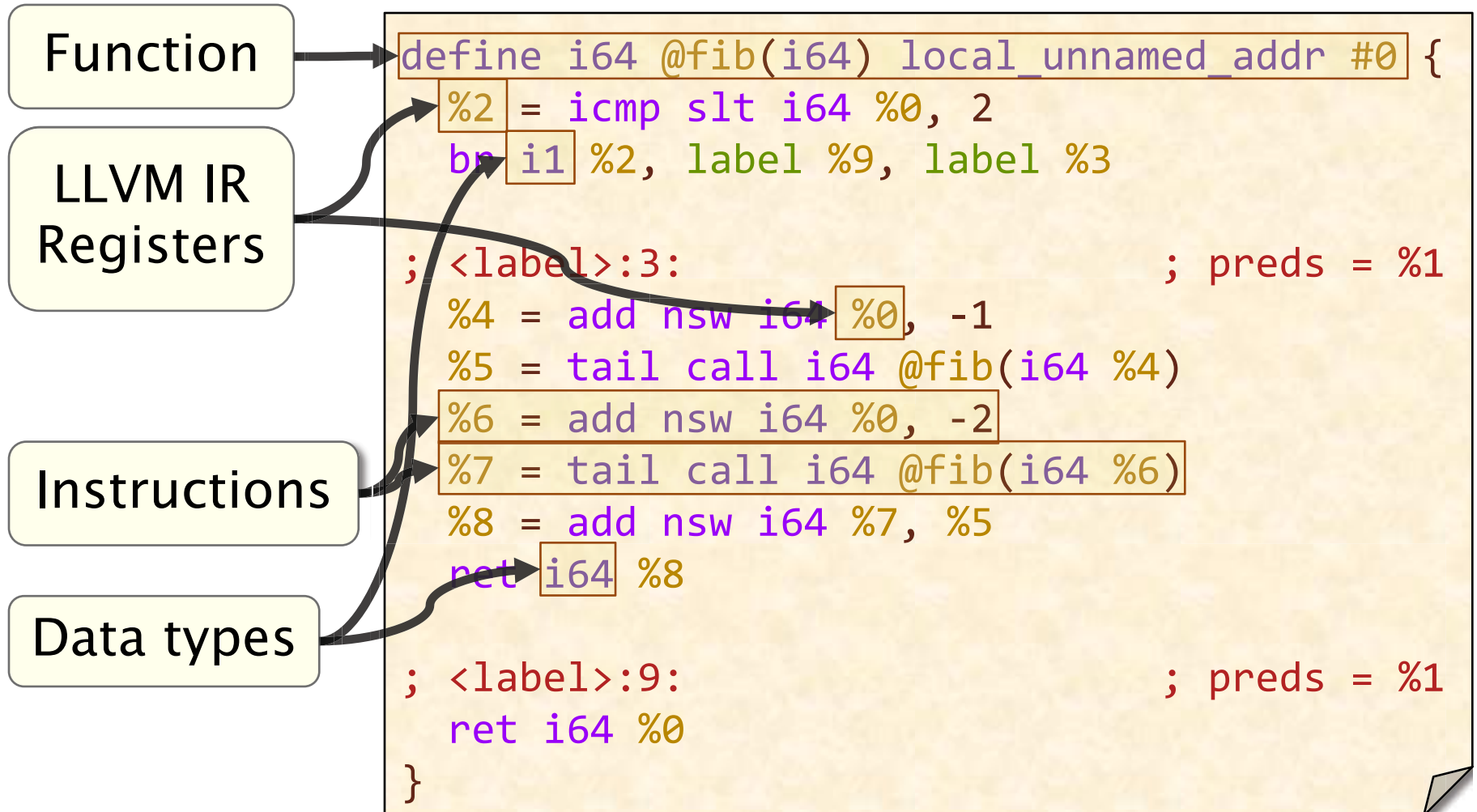
CASE STUDY: FIB

LLVM IR PRIMER



Components of LLVM IR

LLVM IR code fib.ll



LLVM IR Versus Assembly

LLVM IR is *similar* to assembly.

- LLVM IR uses a **simple instruction format**, i.e.,
⟨destination operand⟩ = ⟨opcode⟩ ⟨source operands⟩
- LLVM IR code adopts a **similar structure** to assembly code.
- Control flow is implemented using conditional and unconditional branches.

LLVM IR is *simpler* than assembly.

- Smaller instruction set.
- Infinite LLVM IR registers, similar to variables in C.
- No implicit FLAGS register or condition codes.
- No explicit stack pointer or frame pointer.
- C-like type system.
- C-like functions.

LLVM IR Registers

LLVM IR stores values variables, called *registers*.

- **Syntax:** %<name>
- LLVM registers are **like C variables**: LLVM supports an infinite number of registers, each distinguished by name.
- Register names are **local** to each LLVM IR function.

Registers in an LLVM IR snippet.

```
%4 = add nsw i64 %0, -1
%5 = tail call i64 @fib(i64 %4)
%6 = add nsw i64 %0, -2
%7 = tail call i64 @fib(i64 %6)
%8 = add nsw i64 %7, %5
ret i64 %8
```

One catch: We shall see that LLVM hijacks its syntax for registers to refer to “basic blocks.”

LLVM IR Instructions

LLVM-IR code is organized into *instructions*.

- Syntax for instructions that produce a value:
`%<name> = <opcode> <operand list>`
- Syntax for other instructions:
`<opcode> <operand list>`
- Operands are registers, constants, or “basic blocks.”

Instruction that produces a value.

Instruction that does not produce a value.

```
%4 = add nsw i64 %0, -1
%5 = tail call i64 @fib(i64 %4)
%6 = add nsw i64 %0, -2
%7 = tail call i64 @fib(i64 %6)
%8 = add nsw i64 %7, %5
ret i64 %8
```

Common LLVM IR Instructions

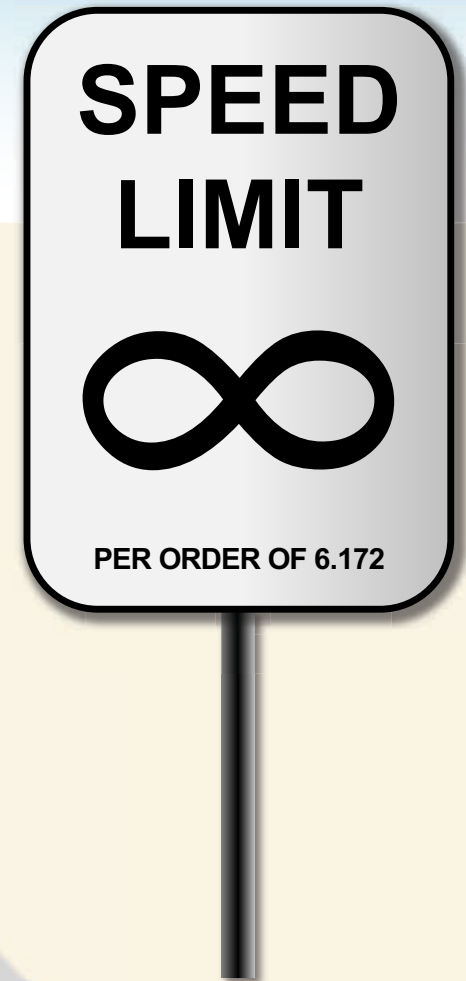
| Type or operation | | Example(s) |
|----------------------|---------------------------|-----------------------------------|
| Data movement | Stack allocation | alloca |
| | Memory read | load |
| | Memory write | store |
| Arithmetic and logic | Type conversion | bitcast, ptrtoint |
| | Integer arithmetic | add, sub, mul, div, shl, shr |
| | Floating-point arithmetic | fadd, fmul |
| | Binary logic | and, or, xor, not |
| | Boolean logic | icmp |
| Control flow | Address calculation | getelementptr |
| | Unconditional jump | br <location> |
| | Conditional jump | br <condition>, <>true>, <>false> |
| | Subroutines | call, ret |
| | Maintaining SSA form | phi |

LLVM IR Data Types

LLVM IR supports a variety of *data types*.

- Integers: `i<number>`
 - Example: A 64-bit integer: `i64`
 - Example: A 1-bit integer: `i1`
- Floating-point values: `double`, `float`
- Arrays: `[<number> x <type>]`
 - Example: An array of 5 `int`'s: `[5 x i32]`
- Structs: `{ <type>, ... }`
- Vectors: `< <number> x <type> >`
- Pointers: `<type>*`
 - Example: A pointer to an 8-bit integer: `i8*`
- Labels (i.e., basic blocks): `label`

STRAIGHT-LINE C CODE TO LLVM IR



Straight-Line C Code in LLVM IR

Straight-line C code (i.e., containing no conditionals or loops) becomes a **sequence** of LLVM IR instructions.

- Arguments are evaluated before the C operation.
- Intermediate results are stored in **registers**.

LLVM IR (register `%0` holds the value of `n`)

C code

```
foo(n-1) + bar(n-2)
```

```
%4 = add nsw i64 %0, -1
%5 = tail call i64 @foo(i64 %4)
%6 = add nsw i64 %0, -2
%7 = tail call i64 @bar(i64 %6)
%8 = add nsw i64 %7, %5
```

Aggregate Types

A variable with an **aggregate type** (i.e., an array or a struct) is typically stored in memory.

Accessing the aggregate type involves computing an address and then reading or writing memory.

C code

```
int A[7];  
A[x];
```

LLVM IR

(register %4 stores the value of x)

```
%5 = getelementptr inbounds [7 x i32],  
    [7 x i32]* %2, i64 0, i64 %4  
%6 = load i32, i32* %5, align 4
```

Compute an address and store it into register %5.

Aggregate Types

A variable with an **aggregate type** (i.e., an array or a struct) is typically stored in memory.

Accessing the aggregate type involves computing an address and then reading or writing memory.

C code

```
int A[7];  
A[x];
```

LLVM IR

(register %4 stores the value of x)

```
%5 = getelementptr inbounds [7 x i32],  
    [7 x i32]* %2, i64 0, i64 %4  
%6 = load i32, i32* %5, align 4
```

Read memory at the address stored in %5.

The `getelementptr` Instruction

The `getelementptr` instruction computes a memory address from a **pointer** and a **list of indices**.

Example: Compute the address $\%2 + 0 + \%4$

```
%5 = getelementptr inbounds [7 x i32],  
    [7 x i32]* %2, i64 0, i64 %4
```

Pointer into
memory

Indices

See <https://llvm.org/docs/GetElementPtr.html>

C FUNCTIONS TO LLVM IR



LLVM IR Functions

Functions in LLVM IR resemble functions in C.

C code fib.c

```
int64_t fib(int64_t n) {  
    ...  
    return n;  
    ...  
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {  
    ...  
    ret i64 %0  
}
```

Function declarations and definitions are C-like.

A `ret` instruction terminates the function, just like a `return` statement in C.

Function Parameters

LLVM IR function parameters map **directly** to their C counterparts.

C code `mm.c`

LLVM IR: `mm.ll`

```
void mm_base(  
  double *restrict C,  
  int n_C,  
  double *restrict A,  
  int n_A,  
  double *restrict B,  
  int n_B,  
  int n) { ... }
```

```
define void @mm_base(  
  double* noalias nocapture,  
  i32,  
  double* noalias nocapture readonly,  
  i32,  
  double* noalias nocapture readonly,  
  i32,  
  i32) local_unnamed_addr #0 { ... }
```

Function parameters are automatically named `%0`, `%1`, `%2`, etc.

Basic Blocks

The body of a function definition is partitioned into *basic blocks*: sequences of instructions (i.e., straight-line code) where control only enters through the first instruction and only exits from the last.

LLVM IR fib.ll

C code fib.c

```
int64_t fib(int64_t n) {  
    if (n < 2)  
        return n;  
    return (fib(n-1) + fib(n-2));  
}
```

```
define i64 @fib(i64) local_unnamed_addr #0 {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3: ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9: ; preds = %1  
    ret i64 %0  
}
```

Control-Flow Graphs

Control-flow instructions (e.g., `br` instructions) induce **control-flow edges** between the basic blocks of a function, creating a **control-flow graph (CFG)**.

Control-flow graph for `fib`

Block 1

```
%2 = icmp slt i64 %0, 2  
br i1 %2, label %9, label %3
```

Block 3

```
; <label>:3: ; preds = %1  
%4 = add nsw i64 %0, -1  
%5 = tail call i64 @fib(i64 %4)  
%6 = add nsw i64 %0, -2  
%7 = tail call i64 @fib(i64 %6)  
%8 = add nsw i64 %7, %5  
ret i64 %8
```

Block 9

```
; <label>:9: ; preds = %1  
ret i64 %0
```

C CONDITIONALS (E.G., IF-THEN-ELSE) TO LLVM IR



C Conditionals

A conditional in C is translated into a *conditional branch instruction*, `br`, in LLVM IR.

C code `fib.c`

```
int64_t fib(int64_t n) {  
    if (n < 2)  
        return n;  
    return (fib(n-1) + fib(n-2));  
}
```

LLVM IR `fib.ll`

```
define i64 @fib(i64)  
local_unnamed_addr #0 {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
...  
  
; <label>:9:                ; preds = %1  
    ret i64 %0  
}
```

The comparison in C becomes an `icmp` instruction.

Arguments of a Conditional Branch

The conditional branch in LLVM IR takes as arguments a 1-bit integer and two basic-block labels.

LLVM IR fib.ll

Predicate

Destination block if the predicate is true.

Destination block if the predicate is false.

```
define i64 @fib(i64)
  local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
...
; <label>:9:                                ; preds = %1
    ret i64 %0
  }
```

Conditional Branches in the CFG

A conditional branch **terminates** its basic block and creates **2** outgoing control-flow edges in the CFG.

Control-flow graph for `fib`

Block 1

```
%2 = icmp slt i64 %0, 2  
br i1 %2, label %9, label %3
```

False branch

Block 3

```
; <label>:3: ; preds = %1  
%4 = add nsw i64 %0, -1  
%5 = tail call i64 @fib(i64 %4)  
%6 = add nsw i64 %0, -2  
%7 = tail call i64 @fib(i64 %6)  
%8 = add nsw i64 %7, %5  
ret i64 %8
```

True branch

Block 9

```
; <label>:9: ; preds = %1  
ret i64 %0
```

Unconditional Branches

If a `br` instruction has just one operand, it is an *unconditional branch*.

LLVM IR

```
; <label>:4:          ; preds = %1  
tail call void @foo() #2  
br label %6
```

Unconditional
branch to block 6.

An unconditional branch **terminates** its basic block and produces **1** outgoing control-flow edge.

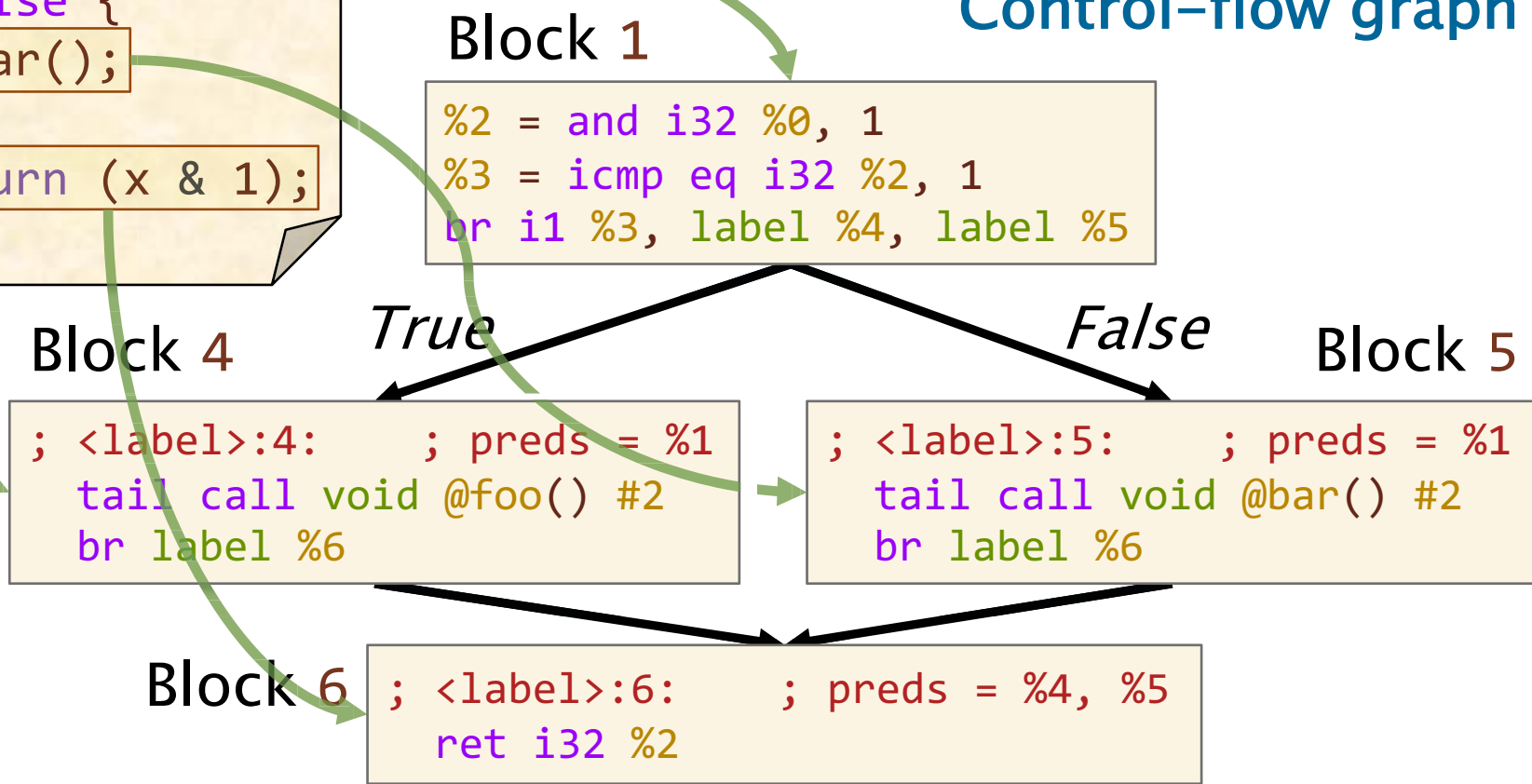
C Conditionals in CFG Form

C code

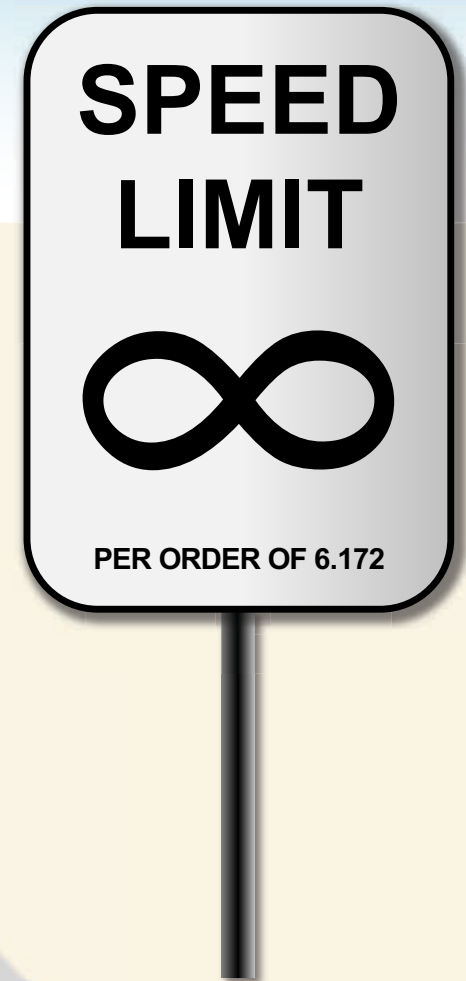
```
int baz(int x) {  
    if (x & 1) {  
        foo();  
    } else {  
        bar();  
    }  
    return (x & 1);  
}
```

In general, a C conditional typically creates a *diamond pattern* in the CFG.

Control-flow graph



C LOOPS TO LLVM IR



Components of a C Loop

```
void dax(  
    double *restrict y, double a,  
    const double *restrict x,  
    int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] = a * x[i];  
}
```

C code

Loop body

Loop control

We'll look at the `phi` instruction soon.

A C loop involves a *loop body* and *loop control*.

LLVM IR snippet

```
; <label>:8; preds = %6, %8  
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]  
%10 = getelementptr inbounds double,  
    double* %2, i64 %9  
%11 = load double, double* %10, align 8  
%12 = fmul double %11, %1  
%13 = getelementptr inbounds double,  
    double* %0, i64 %9  
store double %12, double* %13, align 8  
%14 = add nuw nsw i64 %9, 1  
%15 = icmp eq i64 %14, %3  
br i1 %15, label %7, label %8
```

Loops in the CFG

A C loop produces a *loop pattern* in the control-flow graph.

C code

```
for (int64_t i = 0;
     i < n;
     ++i)
    ...
return;
```

Loop block has 2 incoming edges.

Early test of $0 < n$.

Control-flow graph

```
%5 = icmp sgt i64 %3, 0
br i1 %5, label %8, label %7
```

```
; <label>:8:                ; preds = %6, %8
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]
...
%14 = add nuw nsw i64 %9, 1
%15 = icmp eq i64 %14, %3
br i1 %15, label %7, label %8
```

Exit from the loop.

```
; <label>:7:
ret void
```

Back edge

Loop Control

The *loop control* for a C loop consists of a loop induction variable, an initialization, a condition, and an increment.

C code

```
for (int64_t i = 0; i < n; ++i)  
...
```

Initialization

Increment

Condition

LLVM IR

```
; <label>:8:                                , preds = %6, %8  
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]  
...  
%14 = add nuw nsw i64 %9, 1  
%15 = icmp eq i64 %14, %3  
br i1 %15, label %7, label %8
```

Where's the induction variable in the LLVM IR?

Loop Induction Variables

The induction variable **changes registers** at the code for the loop increment.

```
for (int64_t i = 0; i < n; ++i)  
    y[i] = a * x[i];
```

C code

Why change registers?

Because the loop increment redefines its value.

LLVM IR

```
; <label>:8: ; preds = %6, %8  
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]  
%10 = getelementptr inbounds double,  
    double* %2, i64 %9  
%11 = load double, double* %10, align 8  
%12 = fmul double %11, %1  
%13 = getelementptr inbounds double,  
    double* %0, i64 %9  
store double %12, double* %13, align 8  
%14 = add nuw nsw i64 %9, 1  
%15 = icmp eq i64 %14, %3  
br i1 %15, label %7, label %8
```

Static Single Assignment

LLVM IR maintains the *static single assignment (SSA)* invariant: a register is defined by at most **one** instruction in a function.

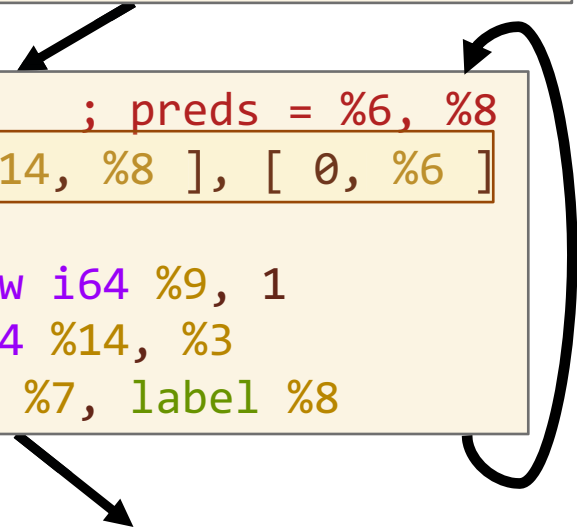
PROBLEM: What happens when control flow merges, e.g., at the entry point of a loop?

SOLUTION: The **phi** instruction.

Loop control-flow graph

```
%5 = icmp sgt i64 %3, 0  
br i1 %5, label %8, label %7
```

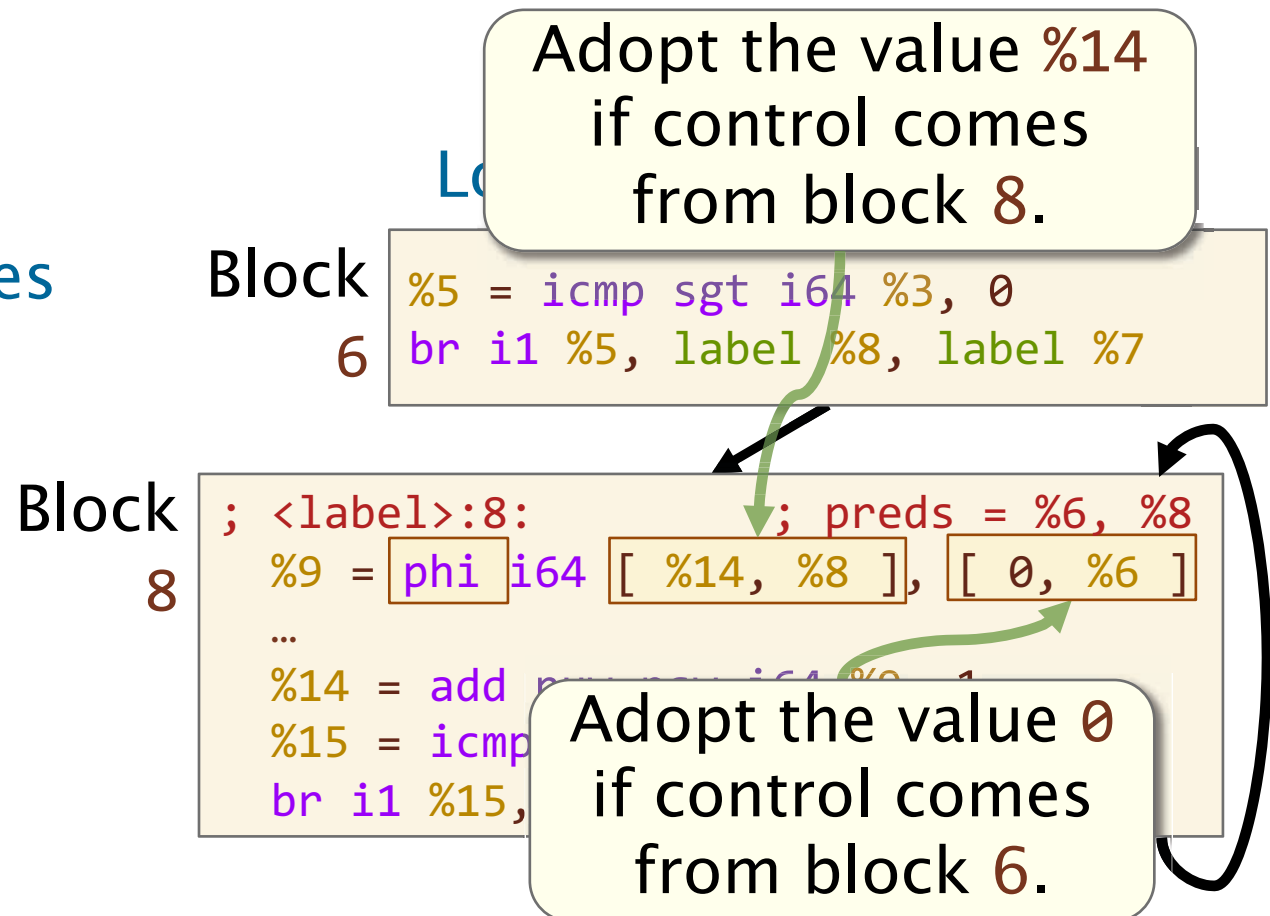
```
; <label>:8: ; preds = %6, %8  
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]  
...  
%14 = add nuw nsw i64 %9, 1  
%15 = icmp eq i64 %14, %3  
br i1 %15, label %7, label %8
```



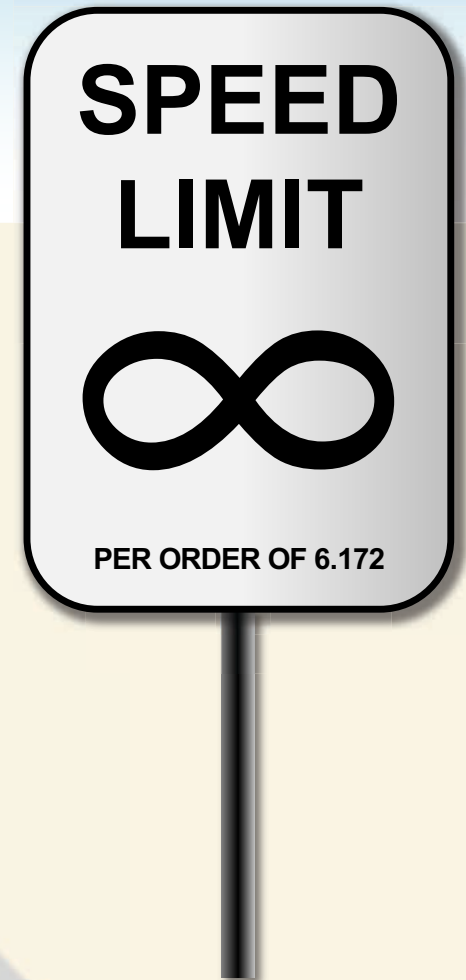
The Phi Instruction

The **phi** instruction specifies, for each predecessor **P** of a basic block **B**, the value of the destination register if control enters **B** via **P**.

- A block with **multiple incoming edges** may have **phi** instructions.
- The **phi** instruction is **not** a real instruction.



LLVM IR ATTRIBUTES



Attributes

LLVM IR constructs (e.g., instructions, operands, functions, and function parameters) might be decorated with *attributes*.

C code

```
const uint64_t deBruijn = 0x022fdd63cc95386d;  
const int convert[64] = { ... };  
int r = convert[(x * deBruijn) >> 58];
```

LLVM IR

```
%4 = getelementptr inbounds [64 x i32],  
[64 x i32]* @convert, i64 0, i64 %3  
%5 = load i32, i32* %4, align 4, !tbaa !2
```

Attribute describing the **alignment** of the read from memory.

Where Do Attributes Come From?

Some attributes are derived from the **source code**.

C code saxpy.c

```
void saxpy(...  
            const double *restrict x,  
            ...)
```

LLVM IR saxpy.ll

```
define void @saxpy(  
    ...  
    double* noalias nocapture readonly,  
    ...)
```

Other attributes are determined by **compiler analysis**.

LLVM IR

```
%15 = load double, double* %14, align 8
```

Analysis determined the alignment of this read.

Summary of LLVM IR

LLVM IR is **similar** to assembly, but **simpler**.

- All computed values are stored in *registers*.
- *Static single assignment*: Each register name is written on at most **one** line of IR.
- A function is modeled as a *control-flow graph*, whose nodes are *basic blocks*, and whose edges denote control flow between basic blocks.
- Compared to C, all operations are **explicit**.
 - All integer sizes are apparent.
 - There are no implicit operations, e.g., type casts.

LLVM IR TO ASSEMBLY



Mapping LLVM IR To Assembly

LLVM IR is structurally similar to assembly.

LLVM IR code fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0
{
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = tail call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = tail call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  ret i64 %8

; <label>:9:                                ; preds = %1
  ret i64 %0
}
```

Assembly code fib.s

```
.globl    _fib
.p2align  4, 0x90
_fib:
## @fib
pushq    %rbp
movq     %rsp, %rbp
pushq    %r14
pushq    %rbx
movq     %rdi, %rbx
cmpq    $2, %rbx
jge     LBB0_1
movq    %rbx, %rax
jmp     LBB0_3

LBB0_1:
leaq    -1(%rbx), %rdi
callq   _fib
movq    %rax, %r14
addq    $-2, %rbx
movq    %rbx, %rdi
callq   _fib
addq    %r14, %rax

LBB0_3:
popq    %rbx
popq    %r14
popq    %rbp
retq
```

Translating LLVM IR to Assembly

The compiler must perform **three tasks** to translate LLVM IR into x86-64 assembly.

- **Select** assembly instructions to implement LLVM IR instructions.
- **Allocate** x86-64 general-purpose registers to hold values.
- **Coordinate** function calls.

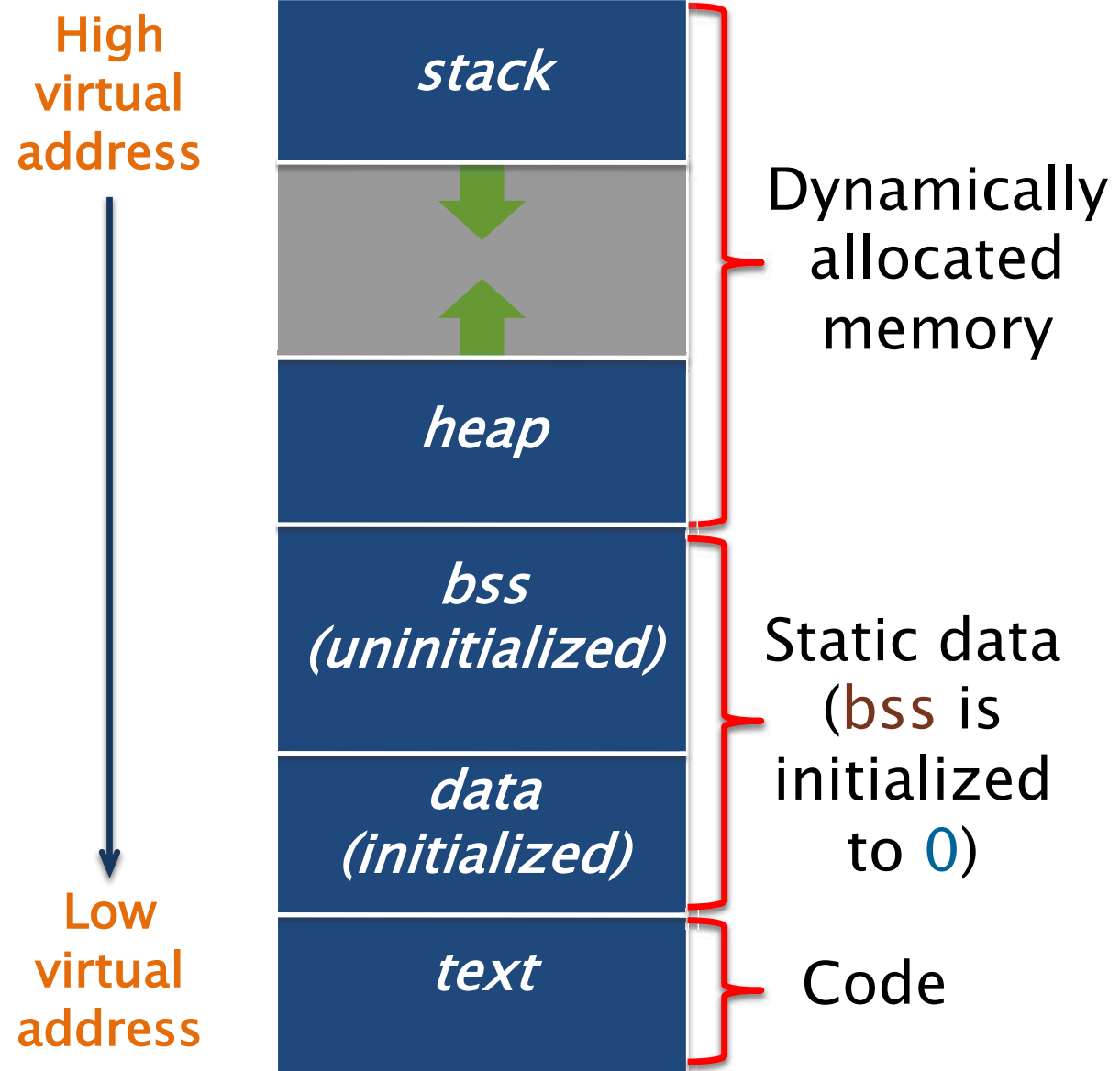
Our main focus

THE LINUX x86-64 CALLING CONVENTION



Layout of a Program in Memory

When a program executes, virtual memory is organized into *segments*.



Assembler Directives

Assembly code contains *directives* that refer to and operate on sections of assembly.

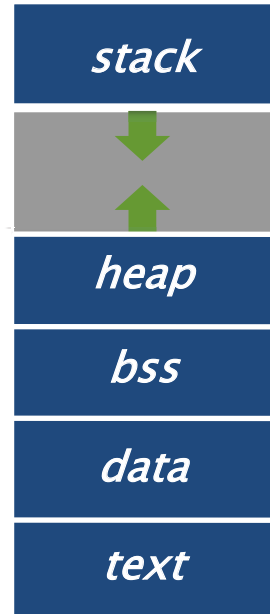
- *Segment directives* organize the contents of an assembly file into segments.
 - “.text”: Identifies the text segment.
 - “.bss”: Identifies the bss segment.
 - “.data”: Identifies the data segment.
- *Storage directives* store content into the current segment.

Examples:

| | |
|--------------------------------|---|
| x: <code>.space 20</code> | Allocates 20 bytes at location x. |
| y: <code>.long 172</code> | Stores the constant 172L at location y. |
| z: <code>.asciz "6.172"</code> | Stores the string “6.172\0” at location z. |
| <code>.align 8</code> | Align the next content to an 8-byte boundary. |

- *Scope and linkage directives* control linking.

Example: “.globl fib”: Makes “fib” visible to other object files.

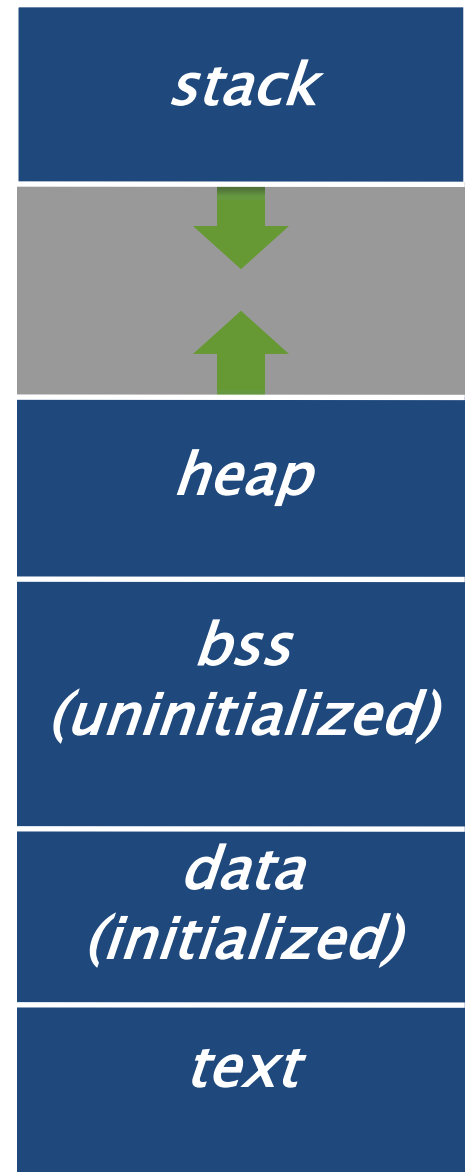


The Call Stack

The *stack* segment stores data in memory to manage **function calls** and **returns**.

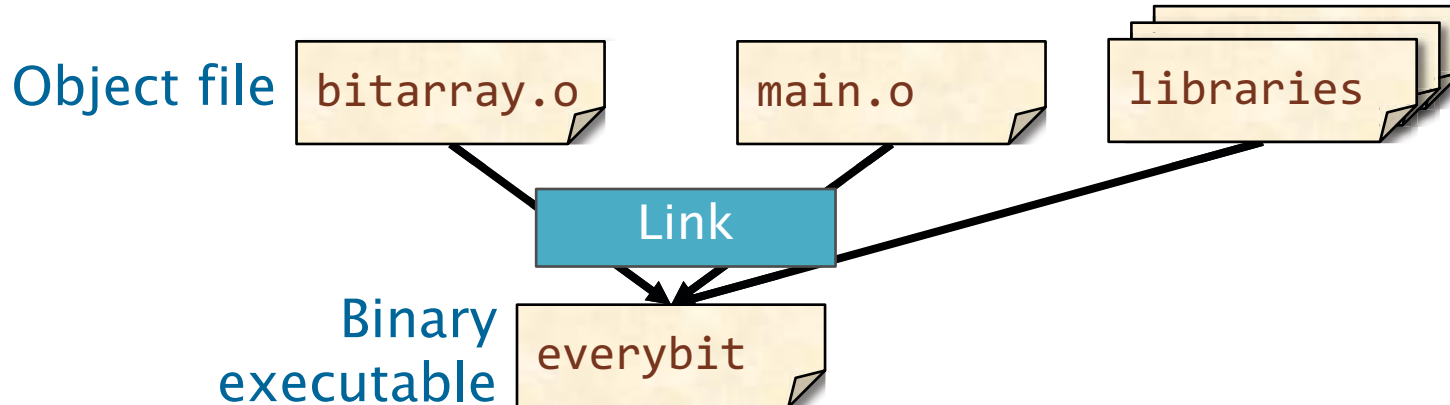
More specifically, what data is stored on the stack?

- The **return address** of a function call.
- **Register state**, so different functions can use the same registers.
- **Function arguments** and **local variables** that don't fit in registers.



Coordinating Function Calls

PROBLEM: How do functions in **different** object files **coordinate** their use of the stack and of register state?



ANSWER: Functions abide by a *calling convention*.

Linux x86-64 Calling Convention

The Linux x86-64 calling convention organizes the stack into *frames*, where each function instantiation gets a single frame of its own.

- The `%rbp` register points to the **top** of the current stack frame.
- The `%rsp` register points to the **bottom** of the current stack frame.

The `call` and `ret` instructions use the stack and the instruction pointer, `%rip`, to manage the **return address** of each function call.

- A `call` instruction in x86-64 **pushes** `%rip` onto the stack and **jumps** to the operand, which is the address of a function.
- A `ret` instruction in x86-64 **pops** `%rip` from the stack and returns to the caller.

Maintaining Registers Across Calls

PROBLEM: Who's responsible for preserving the register state across a function call and return?

- The *caller* might waste work saving register state that the callee doesn't use.
- The *callee* might waste work saving register state that the caller wasn't using.

ANSWER: The Linux x86-64 calling convention does a bit of both.

- *Callee-saved registers:* %rbx, %rbp, %r12-%r15.
- All other registers are *caller-saved*.

C Linkage for x86-64 GPR's

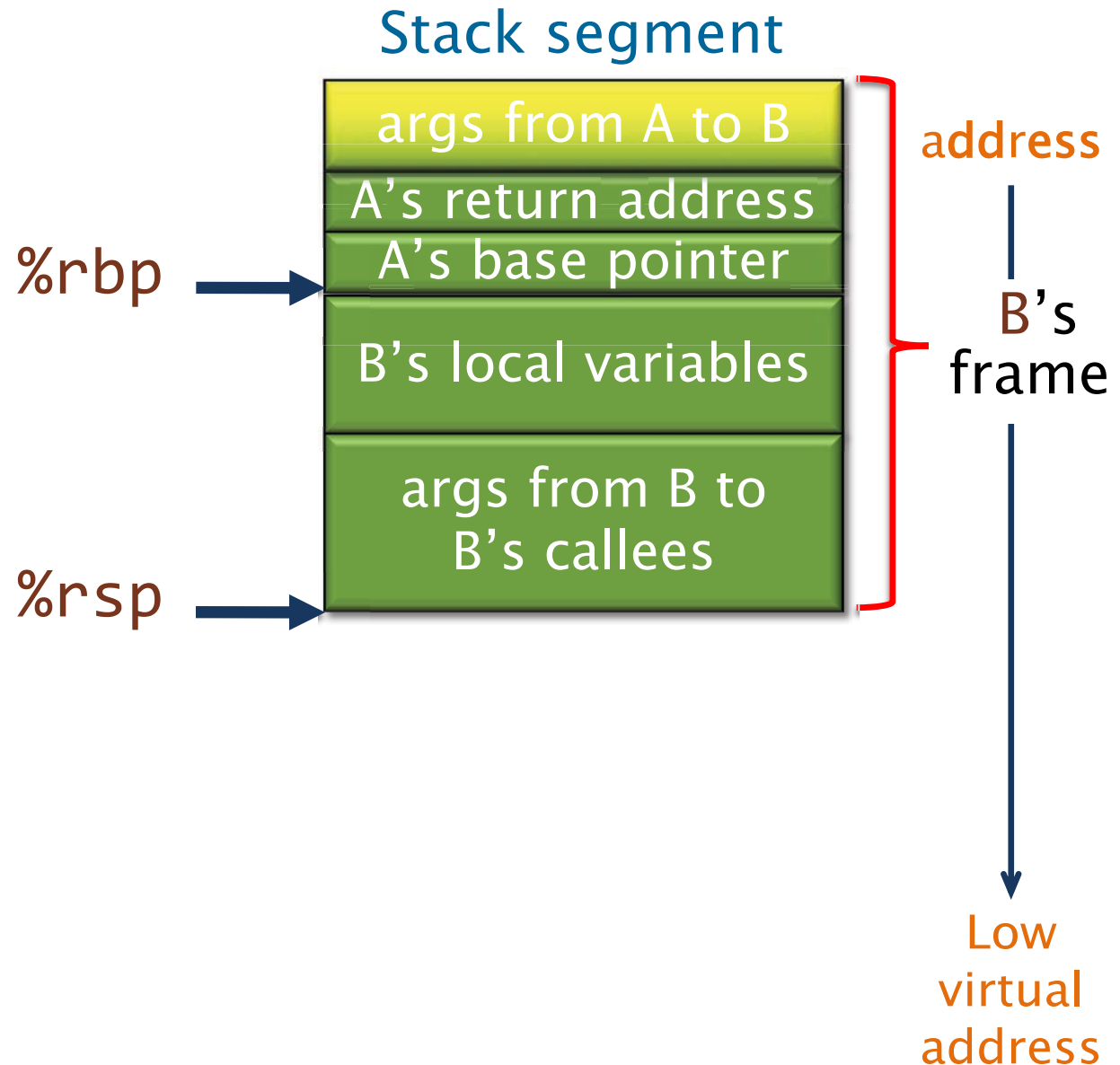
| C linkage | 64-bit name | 32-bit name | 16-bit name | 8-bit name(s) |
|--------------------------|-------------|-------------|-------------|---------------|
| Return value | %rax | %eax | %ax | %ah, %al |
| Callee saved | %rbx | %ebx | %bx | %bh, %bl |
| 4 th argument | %rcx | %ecx | %cx | %ch, %cl |
| 3 rd argument | %rdx | %edx | %dx | %dh, %dl |
| 2 nd argument | %rsi | %esi | %si | %sil |
| 1 st argument | %rdi | %edi | %di | %dil |
| Base pointer | %rbp | %ebp | %bp | %bpl |
| Stack pointer | %rsp | %esp | %sp | %spl |
| 5 th argument | %r8 | %r8d | %r8w | %r8b |
| 6 th argument | %r9 | %r9d | %r9w | %r9b |
| Callee saved | %r10 | %r10d | %r10w | %r10b |
| For linking | %r11 | %r11d | %r11w | %r11b |
| Callee saved | %r12 | | | |
| Callee saved | %r13 | | | |
| Callee saved | %r14 | | | |
| Callee saved | %r15 | | | |

The registers **%xmm0–%xmm7** are used to pass floating-point arguments.

Example: Linux C Subroutine Linkage

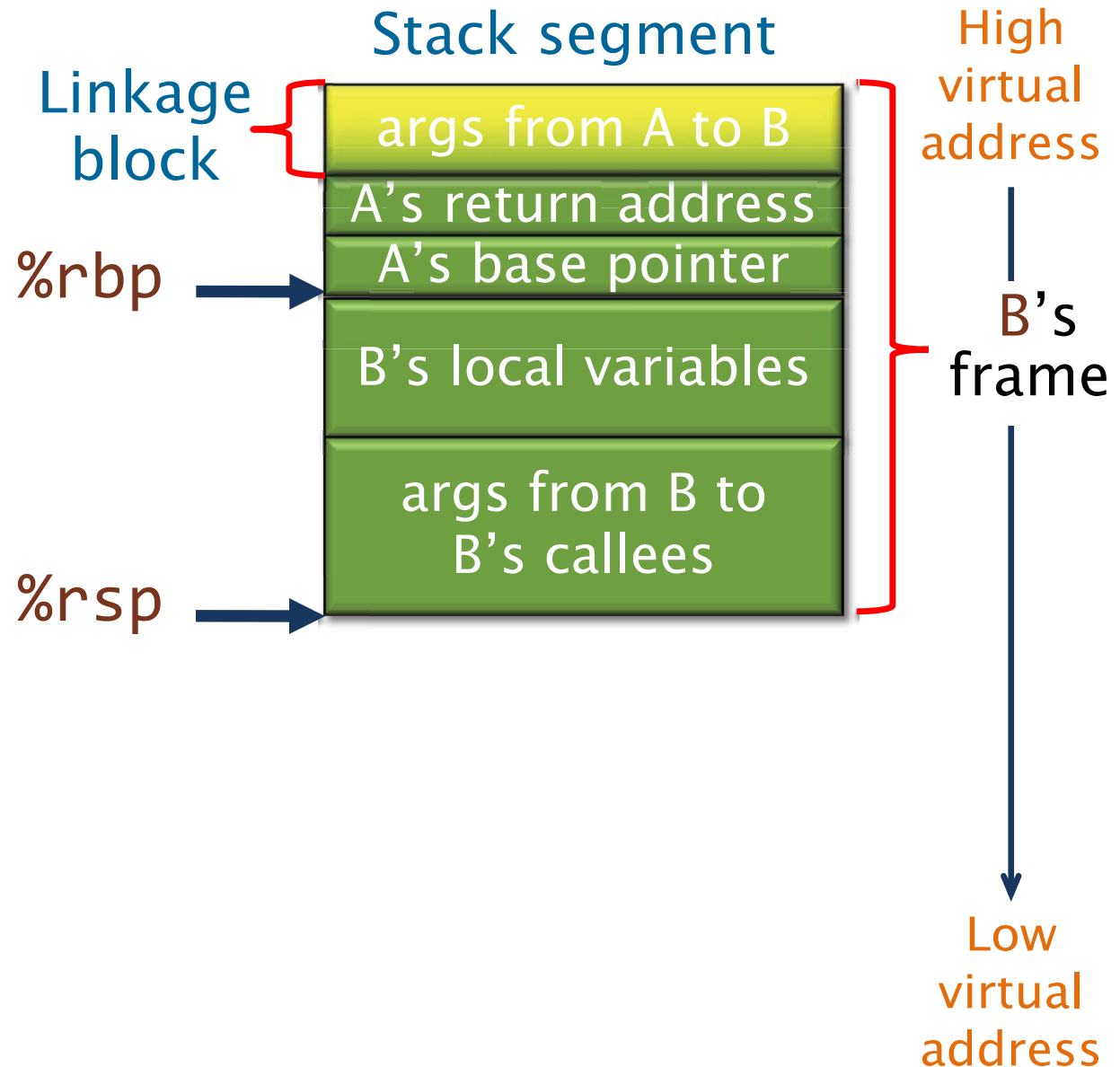
Let's work through an **example** function call in action.

SETUP: Function **B** was called from function **A** and is about to call function **C**.



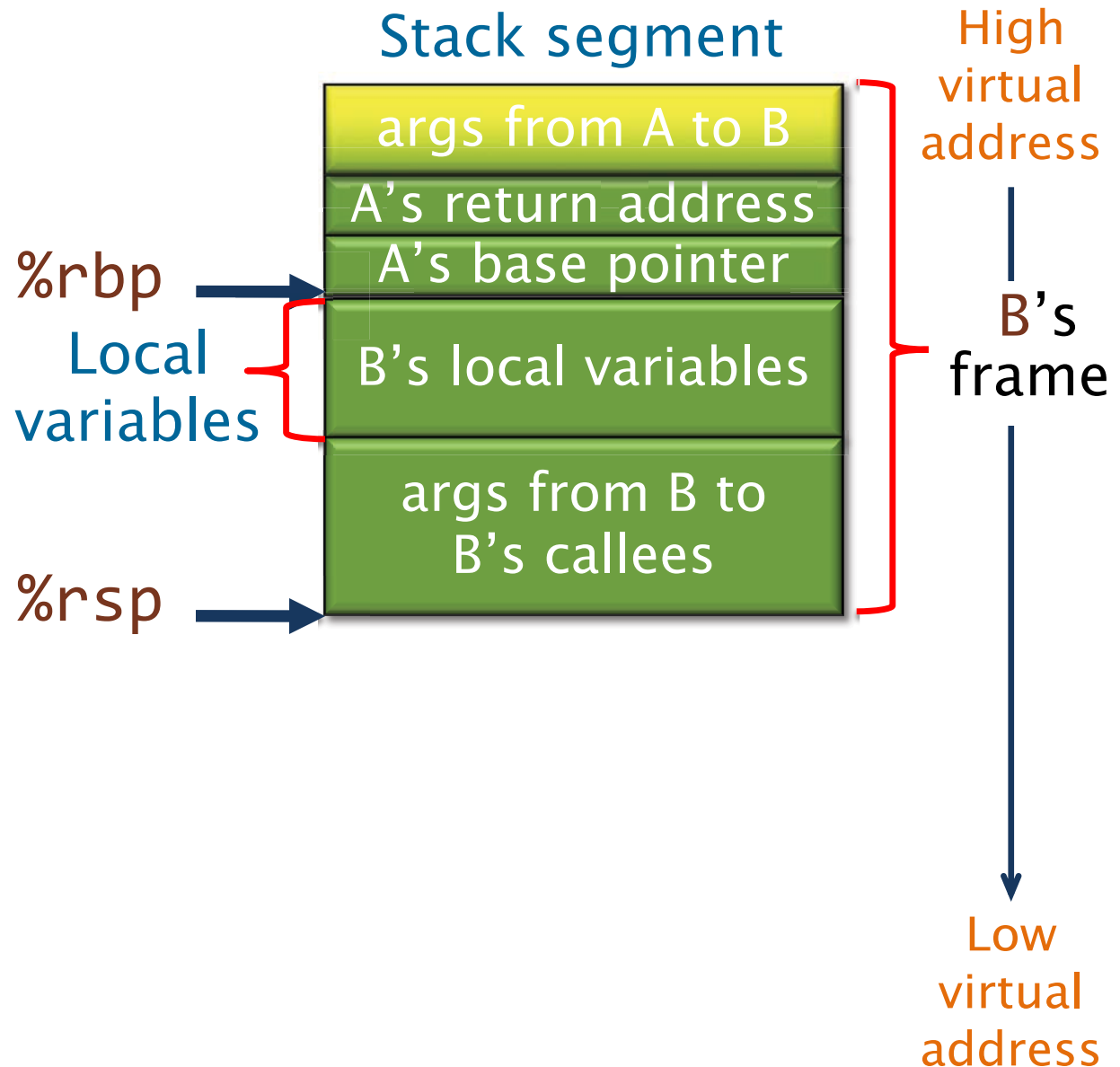
Example: Linux C Subroutine Linkage

Function B accesses its nonregister arguments from A, which lie in a *linkage block*, by indexing `%rbp` with positive offsets.



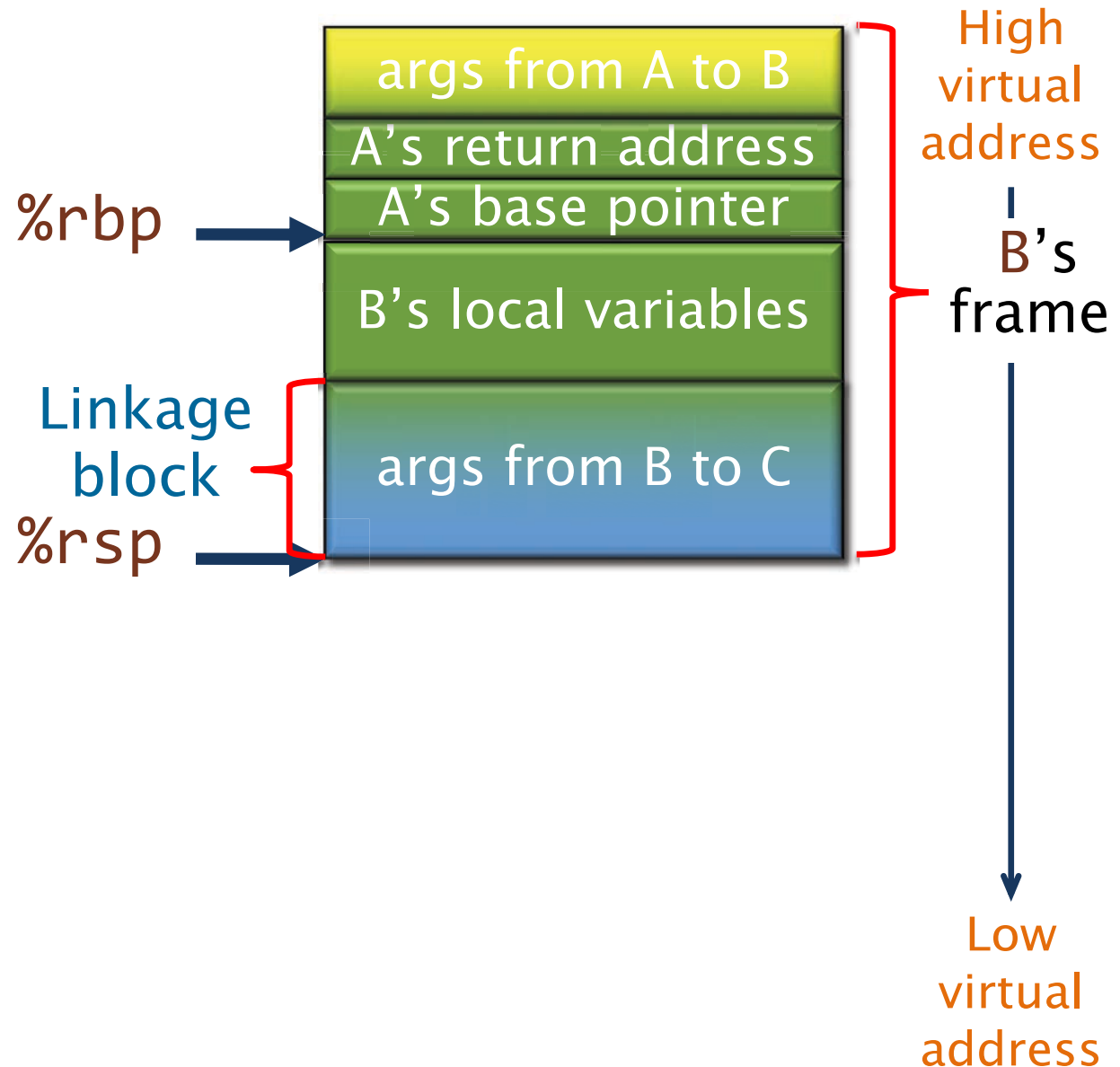
Example: Linux C Subroutine Linkage

Function B accesses its local variables by indexing `%rbp` with negative offsets.



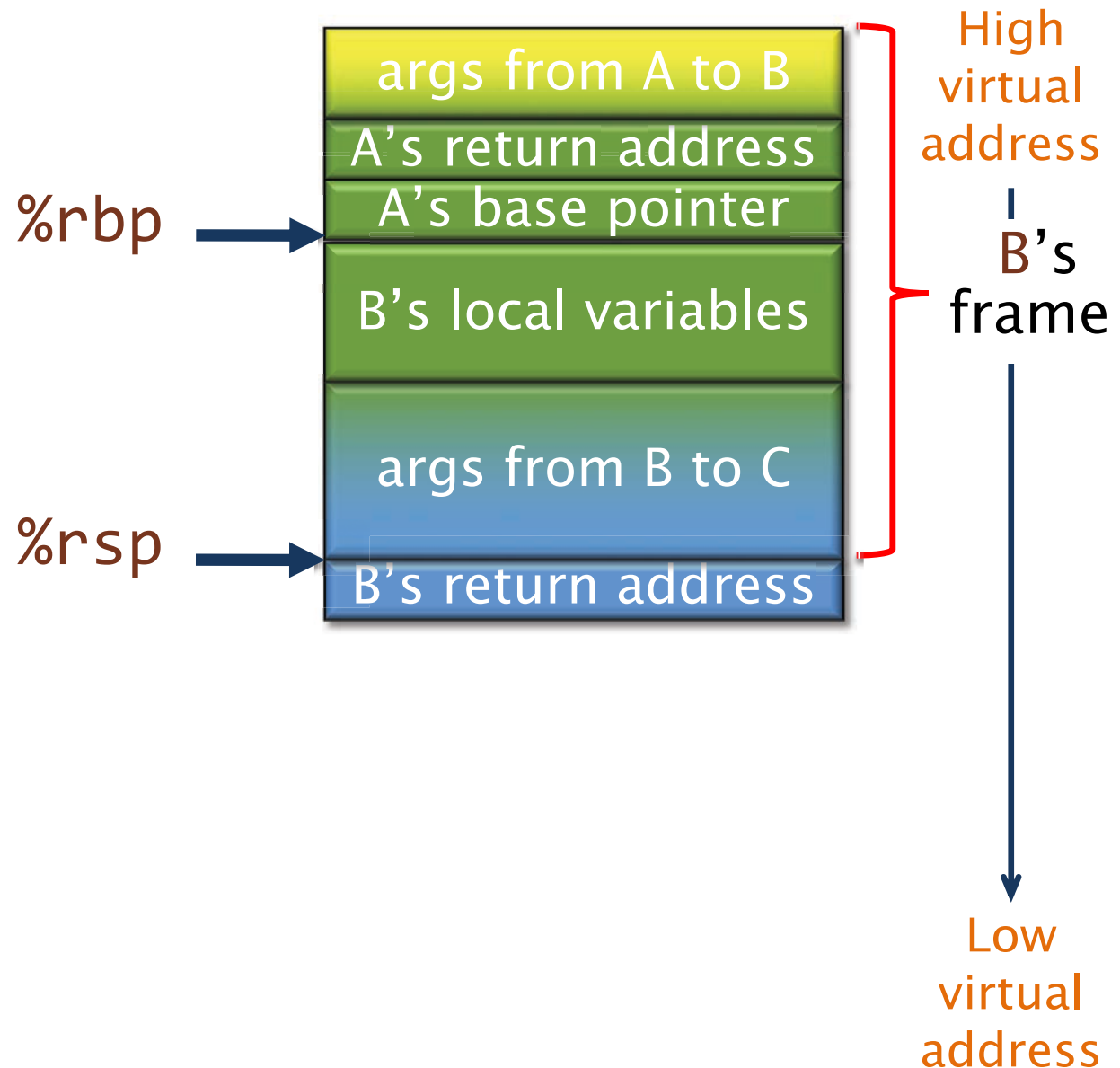
Example: Linux C Subroutine Linkage

Before calling C, B places the **nonregister arguments** for C into the reserved **linkage block** it will share with C, which B accesses by indexing `%rbp` with **negative offsets**.



Example: Linux C Subroutine Linkage

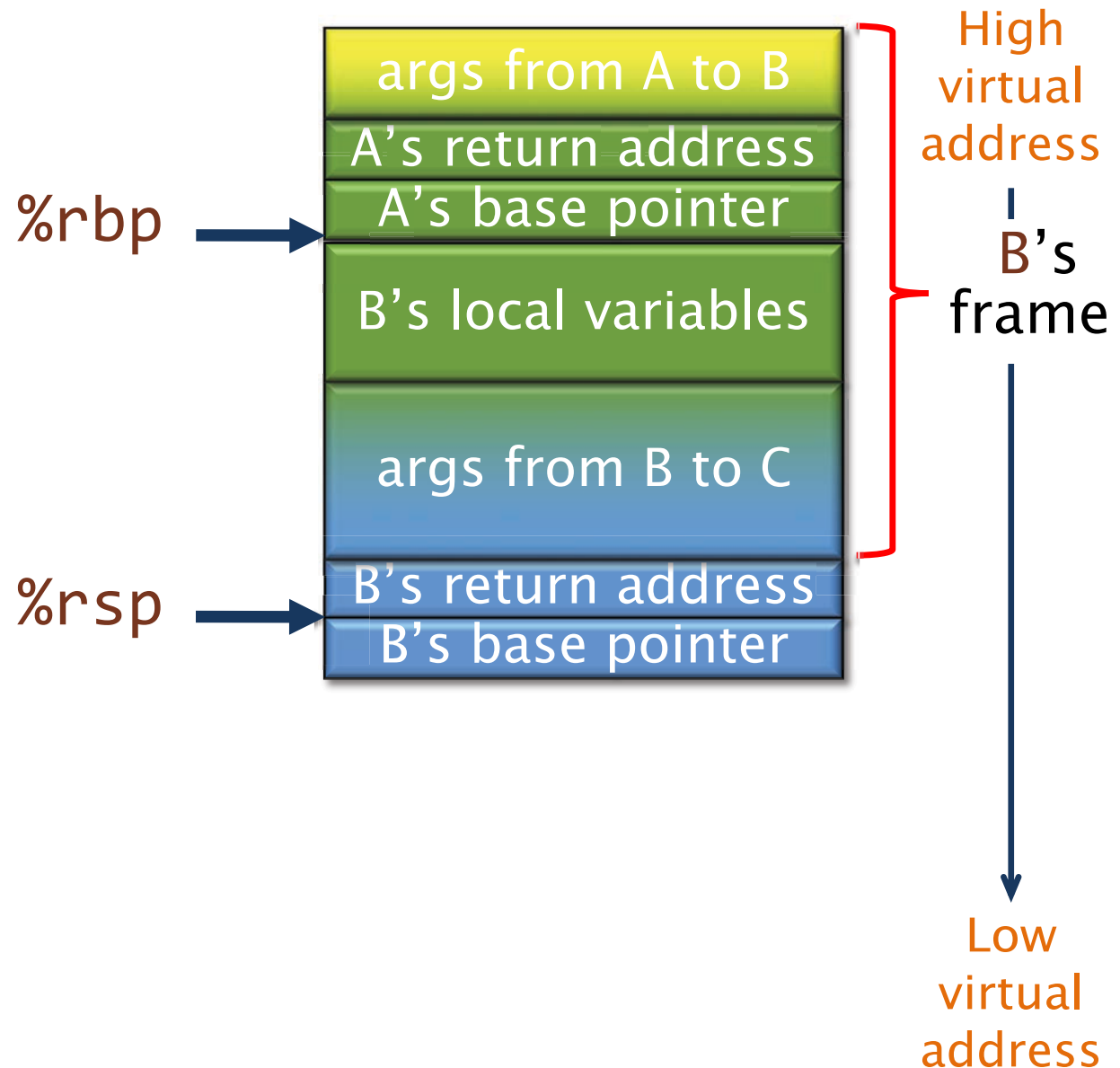
B calls C, which saves the return address for B on the stack and transfers control to C.



Example: Linux C Subroutine Linkage

When function C starts, it executes a **function prologue**:

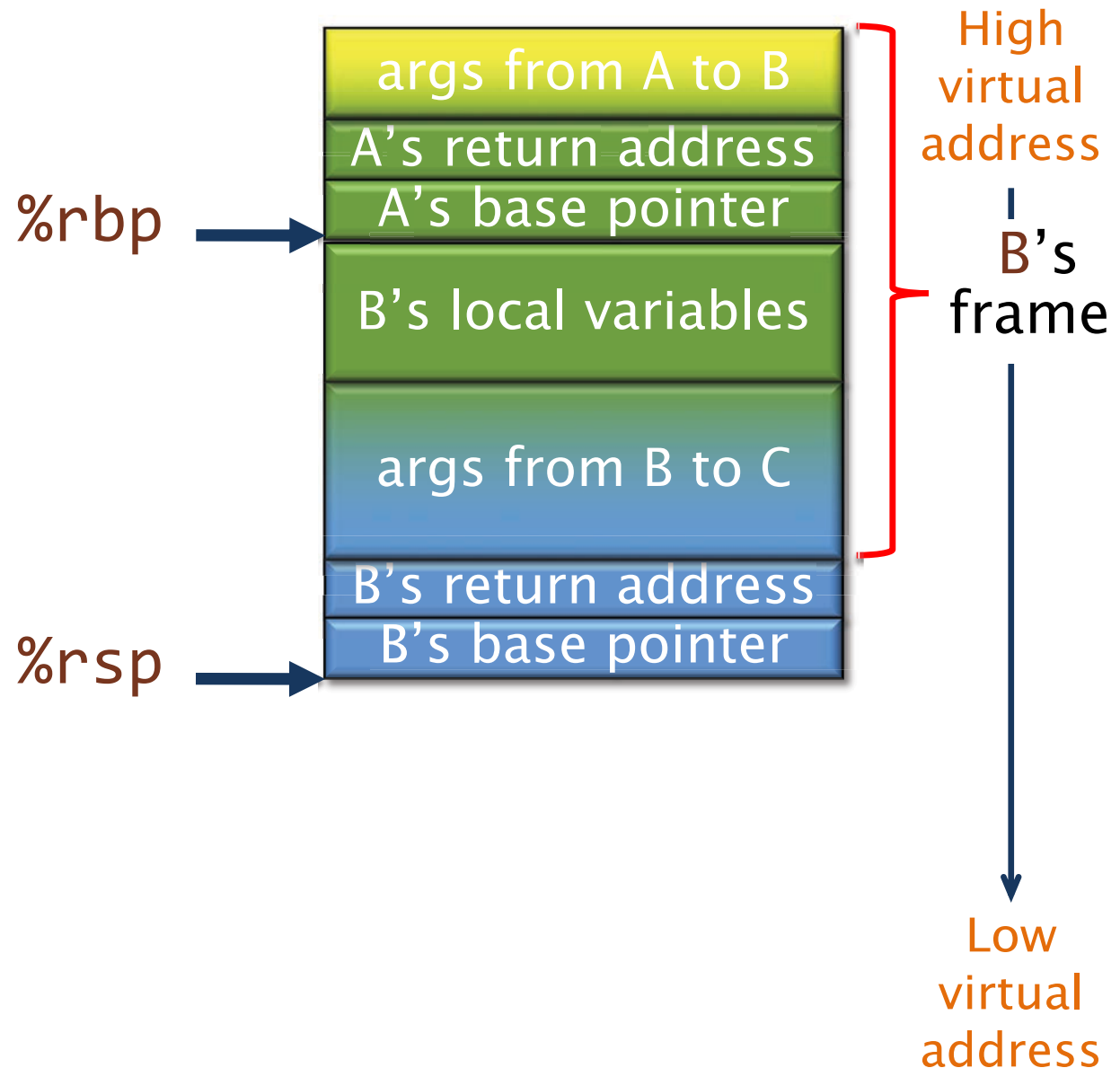
1. Save B's base pointer on the stack,



Example: Linux C Subroutine Linkage

When function C starts, it executes a **function prologue**:

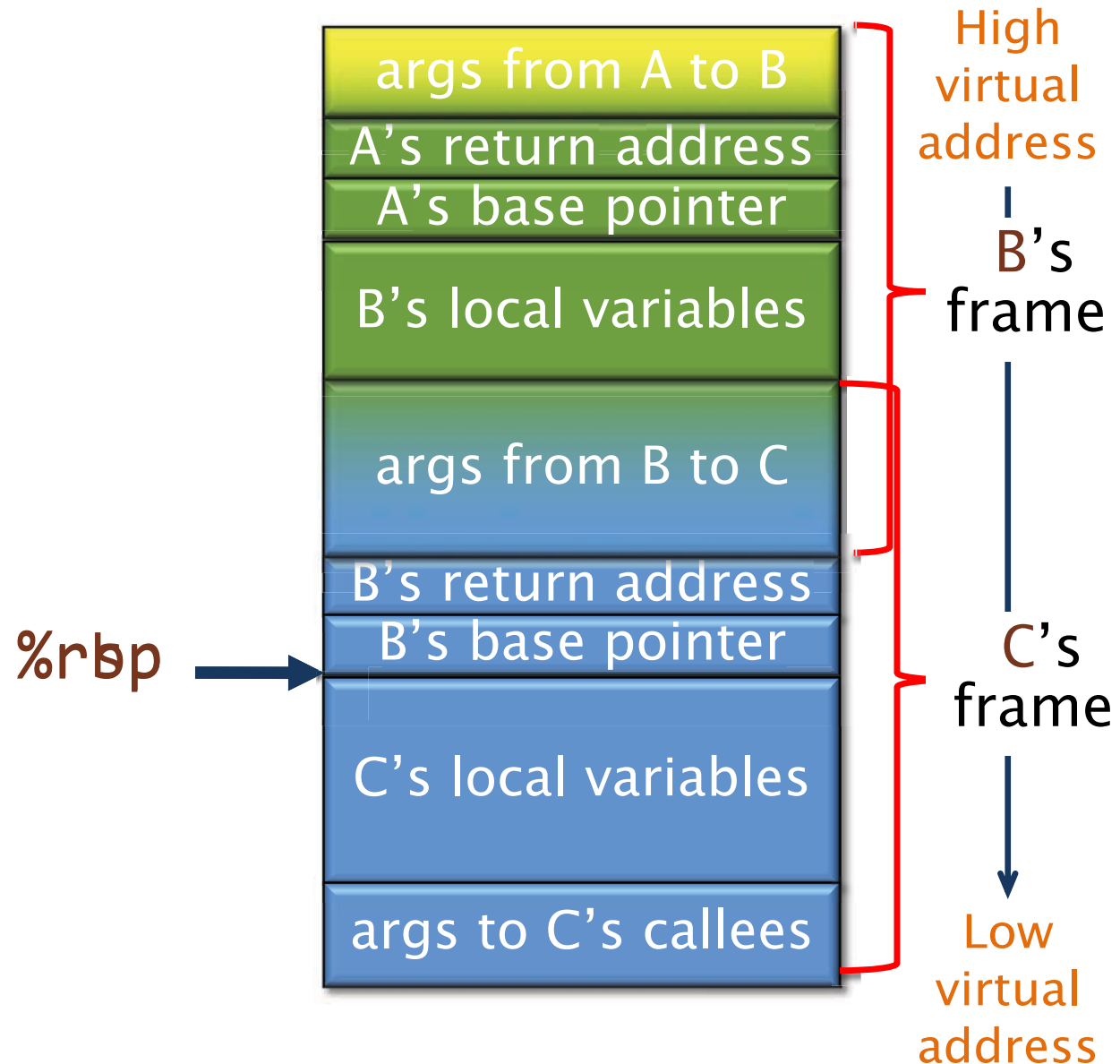
1. Save B's base pointer on the stack,
2. Set $\%rbp = \%rsp$,



Example: Linux C Subroutine Linkage

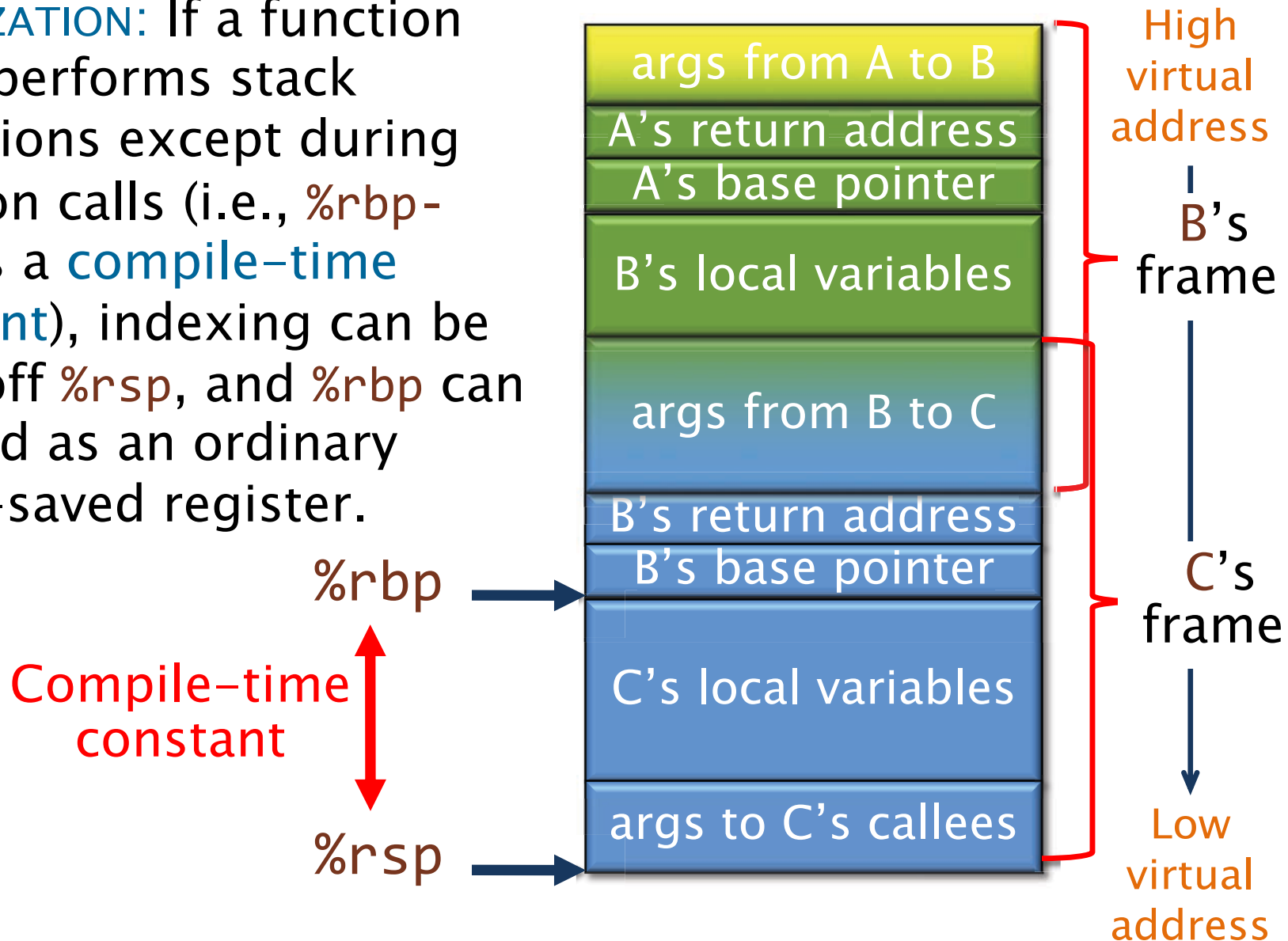
When function **C** starts, it executes a **function prologue**:

1. Save **B**'s base pointer on the stack,
2. Set $\%rbp = \%rsp$,
3. Advance $\%rsp$ to allocate space for **C**'s local variables and linkage block.



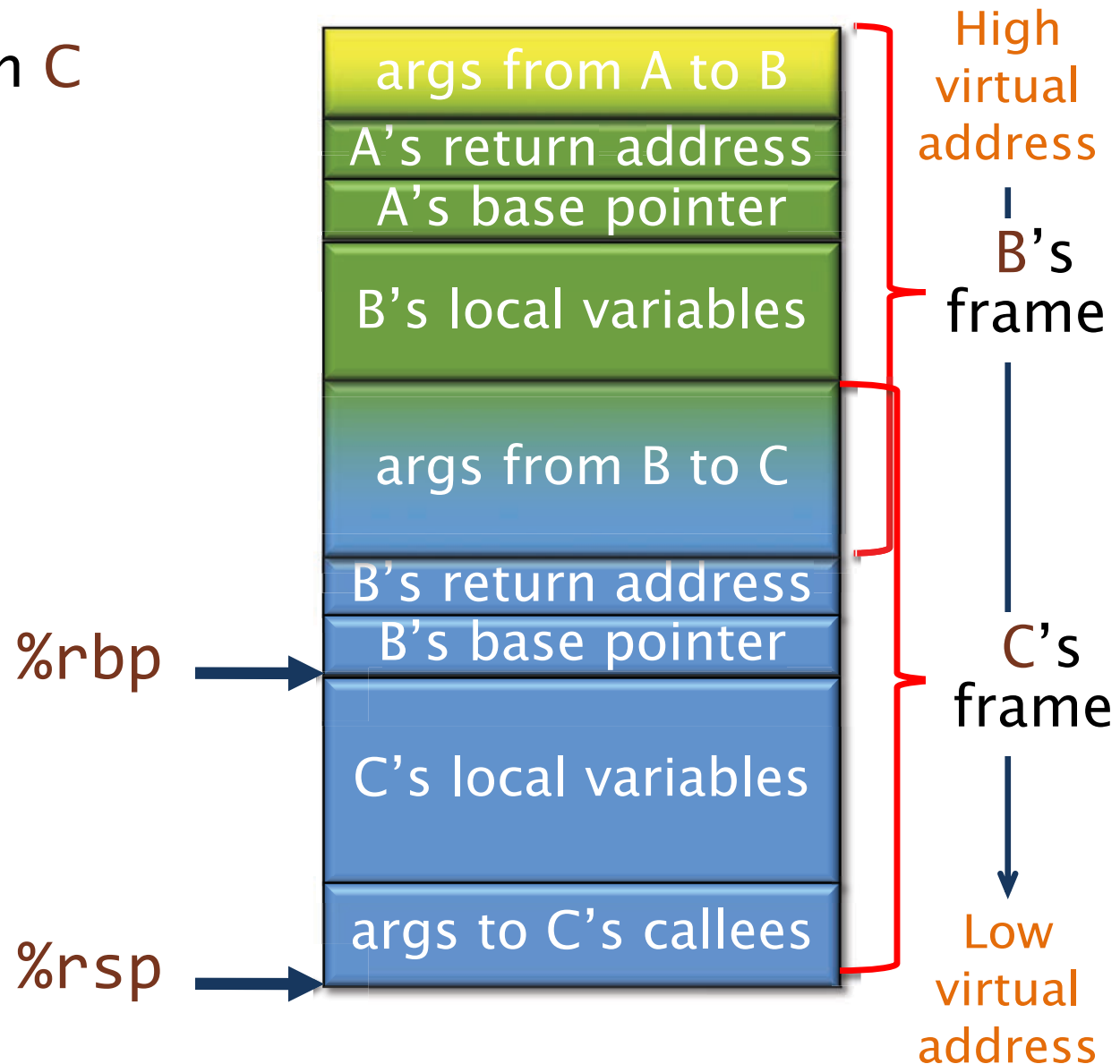
Example: Linux C Subroutine Linkage

OPTIMIZATION: If a function never performs stack allocations except during function calls (i.e., `%rbp-%rsp` is a **compile-time constant**), indexing can be done off `%rsp`, and `%rbp` can be used as an ordinary callee-saved register.



Example: Linux C Subroutine Linkage

For more details on C linkage, see the [System V ABI](#).



CASE STUDY: FIB



Starting Point: fib.c

The C function `fib` computes the n th Fibonacci number $F(n)$ recursively using the formula:

$$F(n) = \begin{cases} n & \text{if } n \in \{0,1\} \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

C code fib.c

```
int64_t fib(int64_t n) {
    if (n < 2)
        return n;
    return (fib(n-1) + fib(n-2));
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```



From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                     ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                     ; preds = %1
    ret i64 %0
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```


From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```

From fib.c to fib.ll

C code fib.c

```
int64_t fib(int64_t n)
{
    if (n < 2)
        return n;
    return (
        fib(n-1)
        +
        fib(n-2));
}
```

LLVM IR fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0 {
    %2 = icmp slt i64 %0, 2
    br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
    %4 = add nsw i64 %0, -1
    %5 = tail call i64 @fib(i64 %4)
    %6 = add nsw i64 %0, -2
    %7 = tail call i64 @fib(i64 %6)
    %8 = add nsw i64 %7, %5
    ret i64 %8

; <label>:9:                                ; preds = %1
    ret i64 %0
}
```

Compiling LLVM IR To Assembly

LLVM IR code fib.ll

```
define i64 @fib(i64) local_unnamed_addr #0
{
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:                                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = tail call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = tail call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  ret i64 %8

; <label>:9:                                ; preds = %1
  ret i64 %0
}
```

Roughly speaking, we can translate LLVM IR into assembly **line by line**.

Assembly code fib.s

```
.globl _fib
.p2align 4, 0x90
_fib:
## @fib
pushq %rbp
movq %rsp, %rbp
pushq %r14
pushq %rbx

movq %rdi, %rbx
cmpq $2, %rbx
jge LBB0_1
movq %rbx, %rax
jmp LBB0_3

LBB0_1:
leaq -1(%rbx), %rdi
callq _fib
movq %rax, %r14
addq $-2, %rbx
movq %rbx, %rdi
callq _fib
addq %r14, %rax

LBB0_3:
popq %rbx
popq %r14
popq %rbp
retq
```

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:          ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:          ; preds = %1  
  ret i64 %0  
}
```

Assembly code fib.s

```
.globl    _fib  
.p2align  4, 0x90  
_fib:    ## @fib  
pushq   %rbp  
movq    %rsp, %rbp  
pushq   %r14  
pushq   %rbx  
movq    %rdi, %rbx  
cmpq    $2, %rbx  
jge     LBB0_1  
movq    %rbx, %rax  
jmp     LBB0_3  
  
LBB0_1:  
leaq   -1(%rbx), %rdi  
callq  _fib  
movq   %rax, %r14  
addq   $-2, %rbx  
movq   %rbx, %rdi  
callq  _fib  
addq   %r14, %rax  
  
LBB0_3:  
popq   %rbx
```

```
.globl    _fib  
.p2align  4, 0x90  
_fib:    ## @fib
```

Declare the `_fib` label to be global.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:          ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:          ; preds = %1  
  ret i64 %0  
}
```

```
pushq  %rbp  
movq   %rsp, %rbp
```

Assembly code fib.s

```
.globl  _fib  
.p2align 4, 0x90  
_fib:  
## @fib  
pushq  %rbp  
movq   %rsp, %rbp  
pushq  %r14  
pushq  %rbx  
movq   %rdi, %rbx  
cmpq   $2, %rbx  
jge    LBB0_1  
movq   %rbx, %rax  
jmp    LBB0_3  
  
LBB0_1:  
leaq   -1(%rbx), %rdi  
callq  _fib  
movq   %rax, %r14  
addq   $-2, %rbx  
movq   %rbx, %rdi  
callq  _fib  
addq   %r14, %rax  
  
LBB0_3:  
popq   %rbx
```

Function prologue:
Save %rbp and sets
%rbp = %rsp.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:           ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:           ; preds = %1  
  ret i64 %0  
}
```

```
pushq   %r14  
pushq   %rbx
```

Assembly code fib.s

```
.globl   _fib  
.p2align 4, 0x90  
_fib:  
## @fib  
pushq   %rbp  
movq    %rsp, %rbp  
pushq   %r14  
pushq   %rbx  
movq    %rdi, %rbx  
cmpq    $2, %rbx  
jge     LBB0_1  
movq    %rbx, %rax  
jmp     LBB0_3  
  
LBB0_1:  
leaq    -1(%rbx), %rdi  
callq   _fib  
movq    %rax, %r14  
addq    $-2, %rbx  
movq    %rbx, %rdi  
callq   _fib  
addq    %r14, %rax  
  
LBB0_3:  
popq    %rbx
```

Save any callee-saved registers that fib will use.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:           ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = tail call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = tail call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  ret i64 %8

; <label>:9:           ; preds = %1
  ret i64 %0
}
```

`movq %rdi, %rbx`

Assembly code fib.s

```
.globl _fib
.p2align 4, 0x90
_fib:
## @fib
pushq %rbp
movq %rsp, %rbp
pushq %r14
pushq %rbx
movq %rdi, %rbx
cmpq $2, %rbx
jge LBB0_1
movq %rbx, %rax
jmp LBB0_3

LBB0_1:
leaq -1(%rbx), %rdi
```

Register `%rdi` stores the function argument `n`.

Copy the incoming argument `n` into `%rbx`.

`popq %rbx`

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:          ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:          ; preds = %1  
  ret i64 %0  
}
```

`cmpq $2, %rbx`

Assembly code fib.s

```
.globl _fib  
.p2align 4, 0x90  
_fib:  
  ## @fib  
  pushq %rbp  
  movq %rsp, %rbp  
  pushq %r14  
  pushq %rbx  
  movq %rdi, %rbx  
  
  cmpq $2, %rbx  
  jge LBB0_1  
  movq %rbx, %rax  
  jmp LBB0_3  
  
LBB0_1:  
  leaq -1(%rbx), %rdi  
  callq _fib  
  movq %rax, %r14  
  addq $-2, %rbx  
  movq %rbx, %rdi  
  callq _fib  
  addq %r14, %rax  
  
LBB0_3:  
  popq %rbx
```

Compare `n` against 2.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:          ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:          ; preds = %1  
  ret i64 %0  
}
```

jge LBB0_1

Assembly code fib.s

```
.globl _fib  
.p2align 4, 0x90  
_fib:  
## @fib  
pushq %rbp  
movq %rsp, %rbp  
pushq %r14  
pushq %rbx  
movq %rdi, %rbx  
cmpq $2, %rbx  
jge LBB0_1  
movq %rax, %rbx  
jmp LBB0_3  
  
LBB0_1:  
leaq -1(%rbx), %rdi  
callq _fib  
movq %rax, %r14  
addq $-2, %rbx  
movq %rbx, %rdi  
callq _fib  
adq %r14, %rax  
  
LBB0_3:  
popq %rbx
```

**False side of LLVM
branch: If $n \geq 2$,
jump to label LBB0_1.**

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:           ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:           ; preds = %1  
    ret i64 %0  
}
```

```
movq    %rbx, %rax  
jmp     LBB0_3
```

Assembly code fib.s

```
.globl  _fib  
.p2align 4, 0x90  
_fib:  
## @fib  
pushq  %rbp  
movq   %rsp, %rbp  
pushq  %r14  
pushq  %rbx  
movq   %rdi, %rbx  
cmpq   $2, %rbx  
jge    LBB0_1  
movq   %rbx, %rax  
jmp    LBB0_3  
  
LBB0_1:  
leaq   -1(%rbx), %rdi  
callq  _fib  
movq   %rax, %r14  
addq   $-2, %rbx  
movq   %rbx, %rdi  
callq  _fib  
addq   %r14, %rax
```

True side of LLVM branch: If $n < 2$, move n into $\%rax$, and jump to label `LBB0_3`.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:           ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:           ; preds = %1  
  ret i64 %0  
}
```

LBB0_1:

Assembly code fib.s

```
.globl _fib  
.p2align 4, 0x90  
_fib:  
  ## @fib  
  pushq %rbp  
  movq %rsp, %rbp  
  pushq %r14  
  pushq %rbx  
  movq %rdi, %rbx  
  cmpq $2, %rbx  
  jge LBB0_1  
  movq %rbx, %rax  
  jmp LBB0_3  
  
LBB0_1:  
  leaq -1(%rbx), %rdi  
  callq _fib  
  movq %rax, %r14  
  addq $-2, %rbx  
  movq %rbx, %rdi  
  callq _fib  
  addq %r14, %rax
```

Label for the false side of the LLVM branch.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:                ; preds = %1  
    ret i64 %0  
}
```

`leaq -1(%rbx), %rdi`

Assembly code fib.s

```
.globl    _fib  
.p2align 4, 0x90  
_fib:  
    ## @fib  
    pushq   %rbp  
    movq    %rsp, %rbp  
    pushq   %r14  
    pushq   %rbx  
    movq    %rdi, %rbx  
    cmpq    $2, %rbx  
    jge     LBB0_1  
    movq    %rbx, %rax  
    jmp     LBB0_3  
  
LBB0_1:  
    leaq    -1(%rbx), %rdi  
    callq   _fib  
    movq    %rax, %r14  
    addq    $-2, %rbx  
    movq    %rbx, %rdi  
    callq   _fib  
    addq    %r14, %rax
```

Compute $n-1$. Store the result in `%rdi`.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = tail call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = tail call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  ret i64 %8

; <label>:9:                ; preds = %1
  ret i64 %0
}
```

callq _fib

Assembly code fib.s

```
.globl _fib
.p2align 4, 0x90
_fib:
  ## @fib
  pushq %rbp
  movq %rsp, %rbp
  pushq %r14
  pushq %rbx
  movq %rdi, %rbx

  cmpq $2, %rbx
  jge LBB0_1
  movq %rbx, %rax
  jmp LBB0_3

LBB0_1:
  leaq -1(%rbx), %rdi
  callq _fib
  movq %rax, %r14
  addq $-2, %rbx
  movq %rbx, %rdi
  callq _fib
  addq %r14, %rax
```

Call fib. The argument $n-1$ has already been stored in %rdi.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:                ; preds = %1  
  ret i64 %0  
}
```

```
movq    %rax, %r14
```

Assembly code fib.s

```
.globl    _fib  
.p2align  4, 0x90  
_fib:     ## @fib  
        .type    %rax, @dword
```

Register `%rax` stores the result of the last function call.

```
LBB0_1:  
        leaq    -1(%rbx), %rdi  
        callq   _fib  
        movq    %rax, %r14  
        addq   $-2, %rbx  
        movq    %rbx, %rdi  
        callq   _fib  
        addq   %r14, %rax
```

Save the result of calling `fib` into `%r14`.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = tail call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = tail call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  ret i64 %8

; <label>:9:                ; preds = %1
  ret i64 %0
}
```

```
addq    $-2, %rbx
movq    %rbx, %rdi
```

Assembly code fib.s

```
.globl  _fib
.p2align 4, 0x90
_fib:
    ## @fib
    pushq   %rbp
    movq   %rsp, %rbp
    pushq   %r14
    pushq   %rbx
    movq   %rdi, %rbx
    cmpq   $2, %rbx
    jge    LBB0_1
    movq   %rbx, %rax
    jmp    LBB0_3

LBB0_1:
    leaq   -1(%rbx), %rdi
    callq  _fib
    movq   %rax, %r14
    addq   $-2, %rbx
    movq   %rbx, %rdi
    callq  _fib
    addq   %r14, %rax
```

Compute $n-2$, then
move the result into
`%rdi`.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:                ; preds = %1  
    ret i64 %0  
}
```

callq _fib

Assembly code fib.s

```
.globl _fib  
.p2align 4, 0x90  
_fib:  
    ## @fib  
    pushq %rbp  
    movq %rsp, %rbp  
    pushq %r14  
    pushq %rbx  
    movq %rdi, %rbx  
    cmpq $2, %rbx  
    jge LBB0_1  
    movq %rbx, %rax  
    jmp LBB0_3  
  
LBB0_1:  
    leaq -1(%rbx), %rdi  
    callq _fib  
    movq %rax, %r14  
    addq $-2, %rbx  
    movq %rbx, %rdi  
    callq _fib  
    addq %r14, %rax
```

Call fib. The argument n-2 is in %rdi.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:                ; preds = %1  
  ret i64 %0  
}
```

`addq %r14, %rax`

Assembly code fib.s

```
.globl _fib  
__declspec(align(4))  
  
movq %rax, %r14  
jmp LBB0_3  
  
LBB0_1:  
  leaq -1(%rbx), %rdi  
  callq _fib  
  movq %rax, %r14  
  addq $-2, %rbx  
  movq %rbx, %rdi  
  callq _fib  
  addq %r14, %rax  
  
LBB0_3:  
  popq %rbx  
  popq %r14  
  popq %rbp  
  retq
```

Add the results of `fib(n-1)` and `fib(n-2)`. Save the sum in `%rax`.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:                ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:                ; preds = %1  
    ret i64 %0  
}
```

LBB0_3:

Assembly code fib.s

```
.globl    _fib  
@_fib:    .align 4, 0x00  
  
        jmp     LBB0_3  
  
LBB0_1:  
        leaq   -1(%rbx), %rdi  
        callq  _fib  
        movq   %rax, %r14  
        addq   $-2, %rbx  
        movq   %rbx, %rdi  
        callq  _fib  
        addq   %r14, %rax  
  
LBB0_3:  
        popq   %rbx  
        popq   %r14  
        popq   %rbp  
        retq
```

Label for the true side
of the LLVM branch.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
  %2 = icmp slt i64 %0, 2  
  br i1 %2, label %9, label %3  
  
; <label>:3:           ; preds = %1  
  %4 = add nsw i64 %0, -1  
  %5 = tail call i64 @fib(i64 %4)  
  %6 = add nsw i64 %0, -2  
  %7 = tail call i64 @fib(i64 %6)  
  %8 = add nsw i64 %7, %5  
  ret i64 %8  
  
; <label>:9:           ; preds = %1  
  ret i64 %0  
}
```

```
popq    %rbx  
popq    %r14  
popq    %rbp
```

Assembly code fib.s

```
.globl    _fib  
_fib:  
    .align 4, 0x00  
  
    movq   %rax, %rax  
    jmp    LBB0_3  
  
LBB0_1:  
    leaq   -1(%rbx), %rdi  
    callq  _fib  
    movq   %rax, %r14  
    addq   $-2, %rbx  
    movq   %rbx, %rdi  
    callq  _fib  
    addq   %r14, %rax  
  
LBB0_3:  
    popq   %rbx  
    popq   %r14  
    popq   %rbp  
    retq
```

Function epilogue:
Restore the callee-saved registers that `fib` used.

From fib.ll to fib.s

LLVM IR code fib.ll

```
define i64 @fib(i64) {  
    %2 = icmp slt i64 %0, 2  
    br i1 %2, label %9, label %3  
  
; <label>:3:           ; preds = %1  
    %4 = add nsw i64 %0, -1  
    %5 = tail call i64 @fib(i64 %4)  
    %6 = add nsw i64 %0, -2  
    %7 = tail call i64 @fib(i64 %6)  
    %8 = add nsw i64 %7, %5  
    ret i64 %8  
  
; <label>:9:           ; preds = %1  
    ret i64 %0  
}
```

retq

Assembly code fib.s

```
.globl _fib  
        .balign 4, 0x00  
  
        movq    %rax, %rax  
        jmp     LBB0_3  
  
LBB0_1:  
        leaq   -1(%rbx), %rdi  
        callq  _fib  
        movq   %rax, %r14  
        addq   $-2, %rbx  
        movq   %rbx, %rdi  
        callq  _fib  
        addq   %r14, %rax  
  
LBB0_3:  
        popq   %rbx  
        popq   %r14  
        popq   %rbp  
        retq
```

Return from the function.

Summary: From C to Assembly

We can reason through the mapping from C code to assembly in two steps: **C to LLVM IR**, and then **LLVM IR to assembly**.

- LLVM IR organizes a C function into a *control-flow graph*.
 - Nodes are *basic blocks*, which correspond to straight-line code in C.
 - C control-flow constructs (i.e., conditionals and loops) induce *control-flow edges*.
- Assembly implements the LLVM IR code using ISA registers and the stack, according to a *calling convention*.

References

Quick reference on assembly instructions:

http://en.wikipedia.org/wiki/X86_instruction_listings

LLVM IR language reference

<https://llvm.org/docs/LangRef.html>

MIT OpenCourseWare
<https://ocw.mit.edu>

6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.