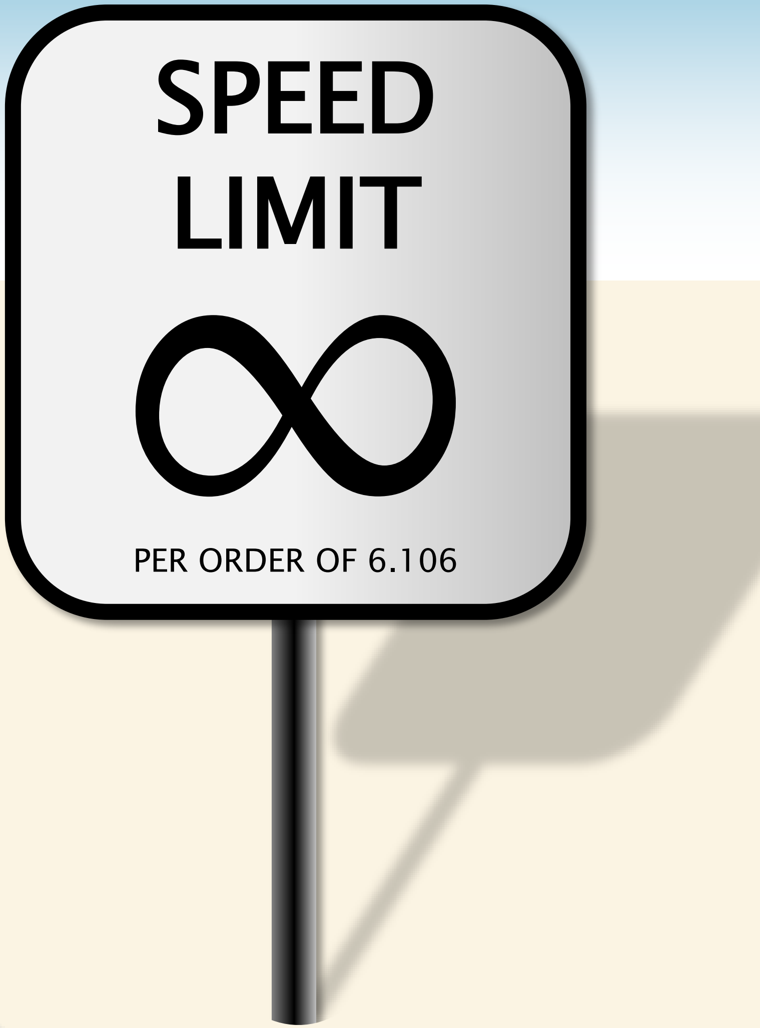
Performance
Engineering of
Software Systems



Lecture 5
C to Assembly

Tao B. Schardl

September 22, 2022

Where We Stand

Lecture 4: Computer Architecture

- Basics of x86-64 assembly: instructions, registers, data types, memory addressing modes, condition codes, etc.

C code fib.c

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

```
$ clang -O1 fib.c -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, %rdi
        jl        LBB0_2
        leaq      -1(%rbx), %rdi
        callq     _fib
        movq      %rax, %r14
        addq      $-2, %rbx
        movq      %rbx, %rdi
        callq     _fib
        movq      %rax, %rbx
        addq      %r14, %rbx
LBB0_2:
        movq      %rbx, %rax
        popq      %rbx
        popq      %r14
        popq      %rbp
        retq
```

This lecture:

- How C code becomes x86-64 assembly.

Mapping C Code to Assembly

It's not always clear how C code relates to assembly!

C code `fib.c`

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

```
$ clang -O1 fib.c -S
```

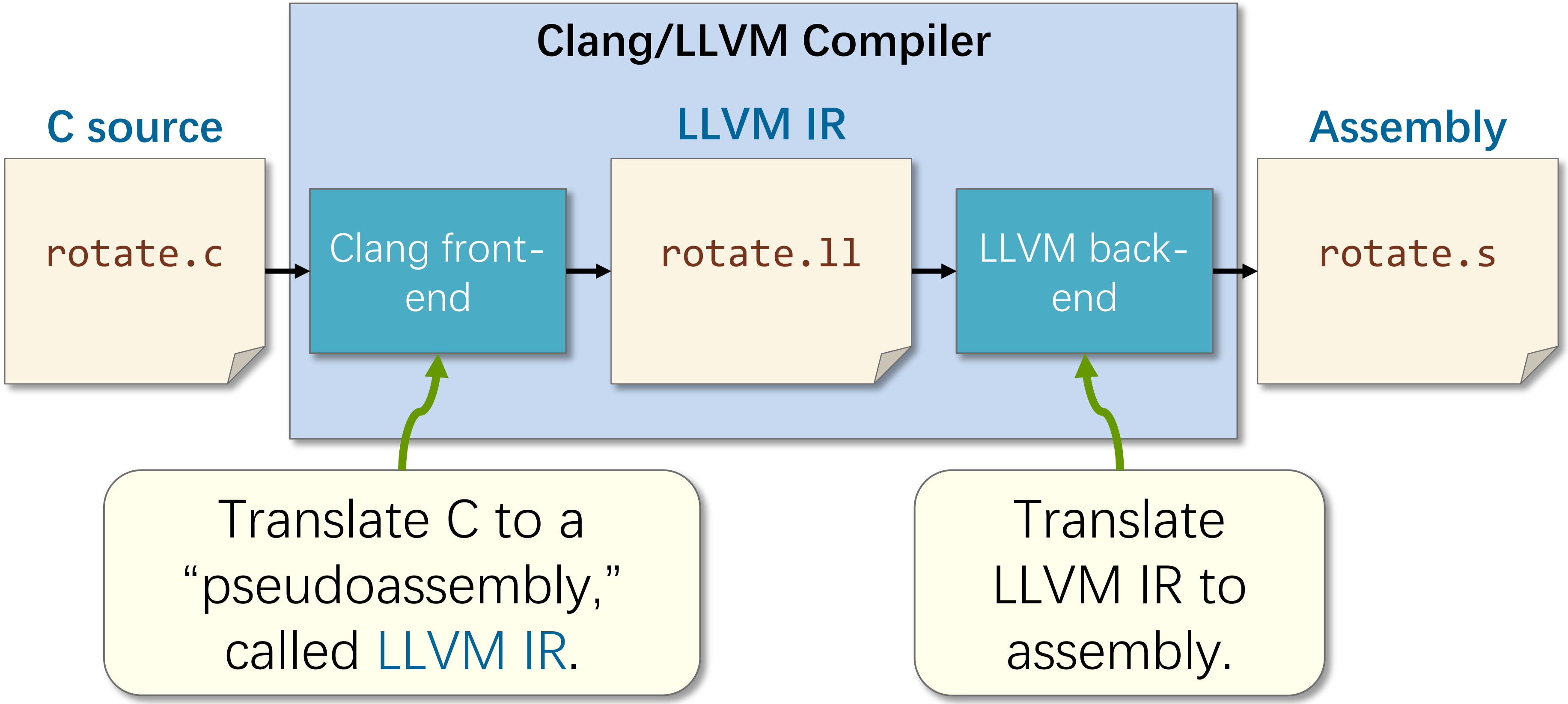
???

Assembly code `fib.s`

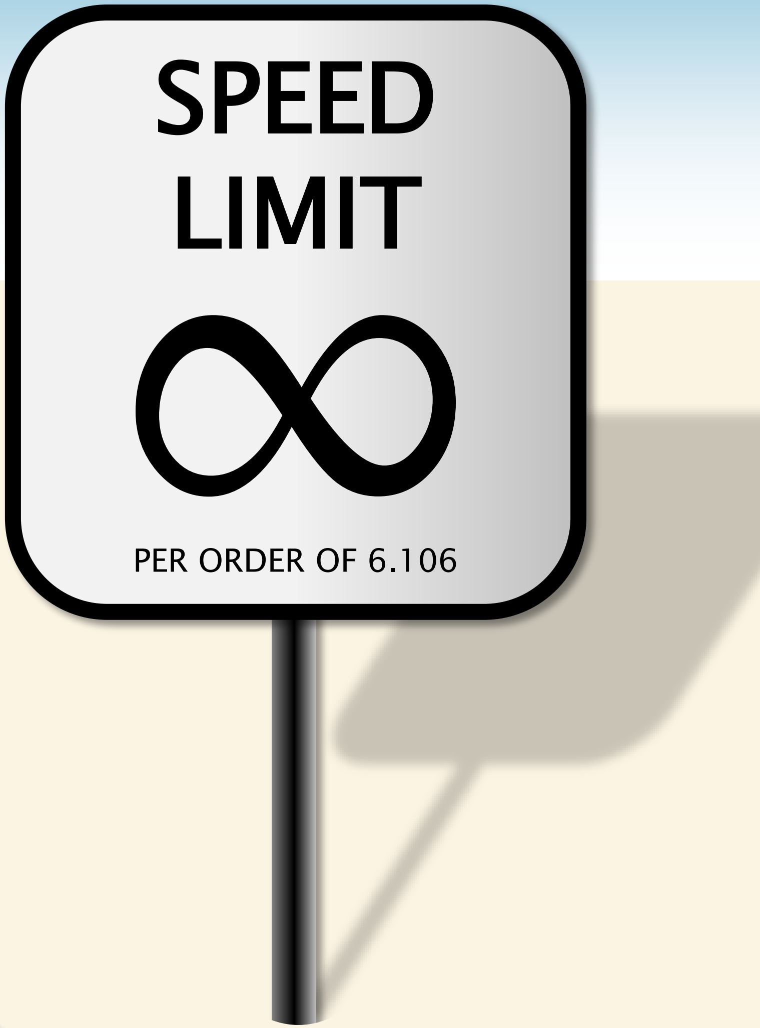
```
            .globl      _fib
            .p2align    4, 0x90
_fib:       ## @fib
            pushq       %rbp
            movq        %rsp, %rbp
            pushq       %r14
            pushq       %rbx
            movq        %rdi, %rbx
            cmpq        $2, %rdi
            jl          LBB0_2
            leaq        -1(%rbx), %rdi
            callq       _fib
            movq        %rax, %r14
            addq        $-2, %rbx
            movq        %rbx, %rdi
            callq       _fib
            movq        %rax, %rbx
            addq        %r14, %rbx
LBB0_2:
            movq        %rbx, %rax
            popq        %rbx
            popq        %r14
            popq        %rbp
            retq
```

Clang/LLVM Compiler Pipeline

To understand this correspondence, let us see how the compiler reasons about it.



Performance Engineering of Software Systems



LECTURE 5

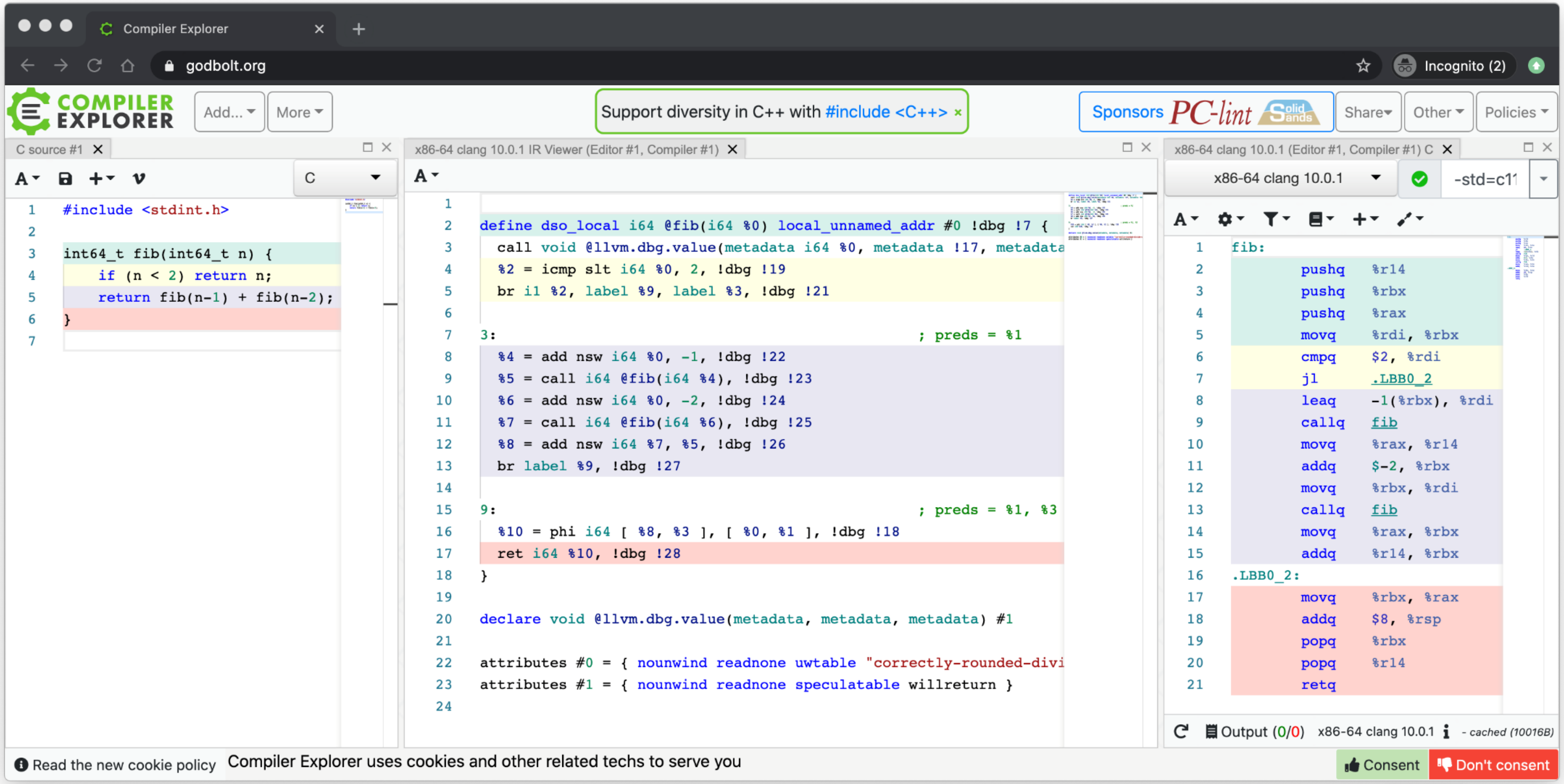
C to LLVM IR to Assembly

Tao B. Schardl

September 22, 2022

Compiler Explorer

We will use Compiler Explorer (`https://godbolt.org/`) to study this topic.



Running Example: `fib.c`

The C function `fib` computes the nth 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`

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

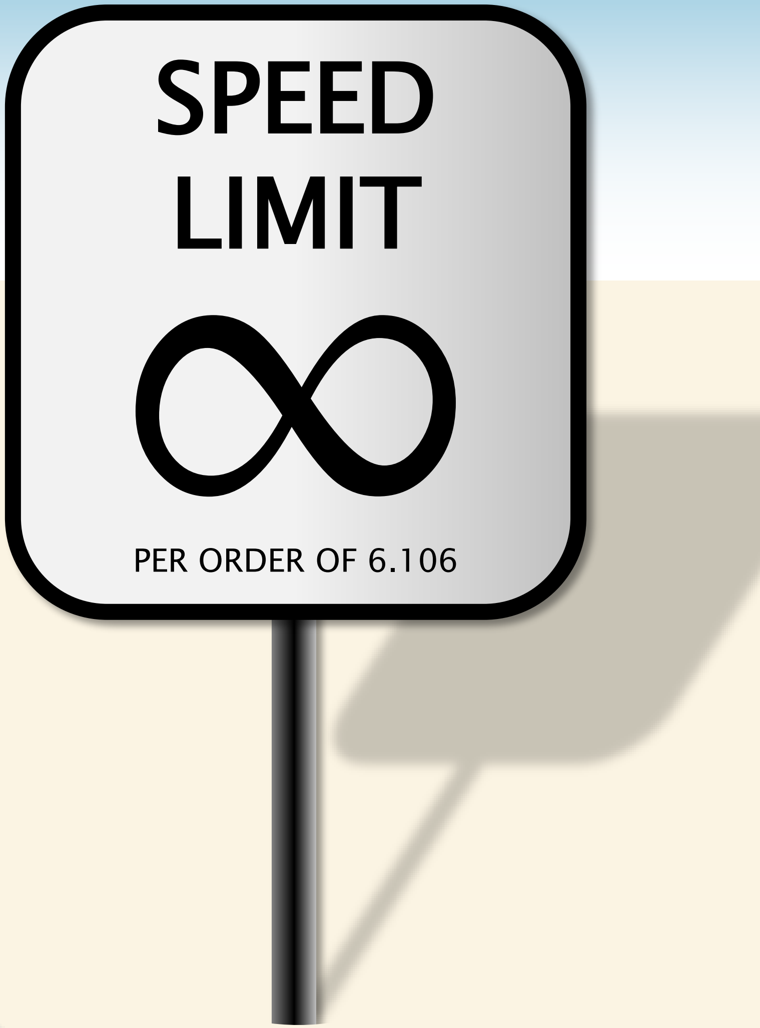
Outline

Running Example: Fib

- C to LLVM IR
  - Basic Organization of LLVM IR
  - C Conditionals in LLVM IR
  - Static Single Assignment Form
- LLVM IR to Assembly
  - Linux x86-64 Calling Convention

Optional Slides

- Handling LLVM IR Manually
- LLVM IR Primer
- C Loops in LLVM IR
- Memory Operations
- LLVM IR Attributes



Basic Organization of LLVM IR

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                                    ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                    ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Components of LLVM IR

LLVM IR code `fib.ll`

Function

LLVM IR registers

Instructions

Data types

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

3:                                  ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                  ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Comparing LLVM IR and Assembly

LLVM IR is *similar* to assembly.

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

LLVM IR is *simpler* than assembly.

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

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 function parameters map directly to their C counterparts.

Function declarations and definitions are C-like.

LLVM IR `fib.ll`

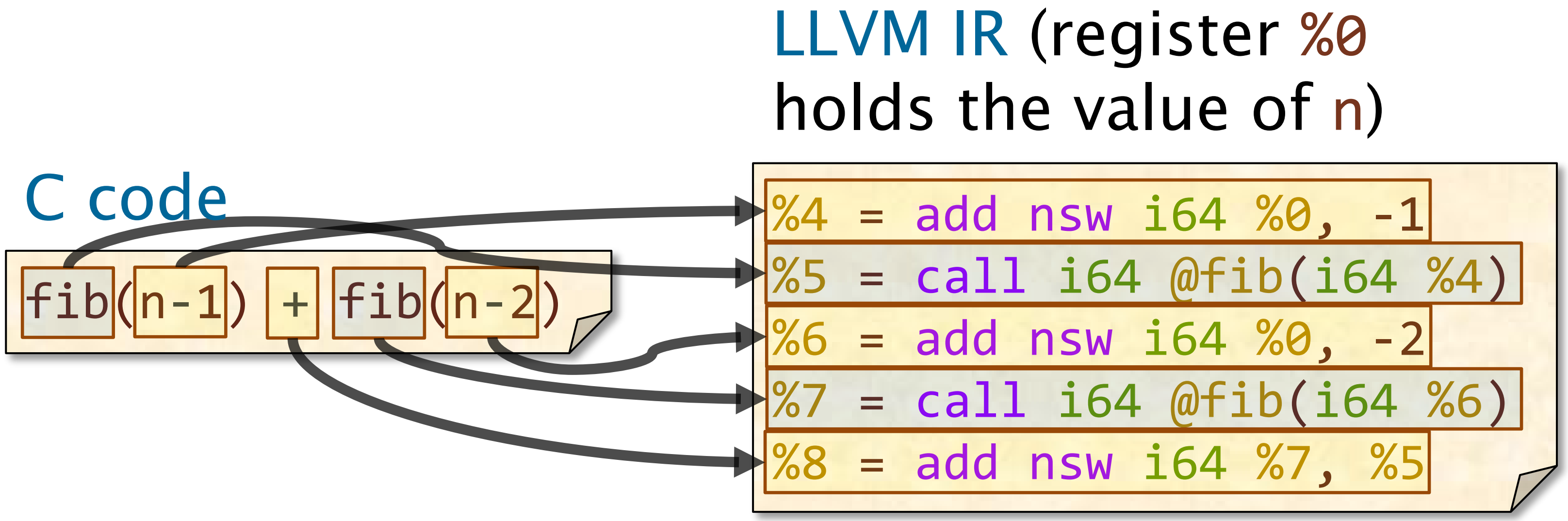
```
define i64 @fib(i64 %0) {
  …
  ret i64 %10
}
```

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

Straight-Line C Code in LLVM IR

Straight-line C code — code 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.



Basic Blocks

The body of a function definition is partitioned into ***basic blocks:*** sequences of instructions where control only enters through the first instruction and only exits from the last.

C code `fib.c`

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

LLVM IR `fib.ll`

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

3:                                  ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                  ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Basic-Block Terminators

A basic block is terminated by a control-flow instruction, typically a br instruction, that describes how control exits that block.

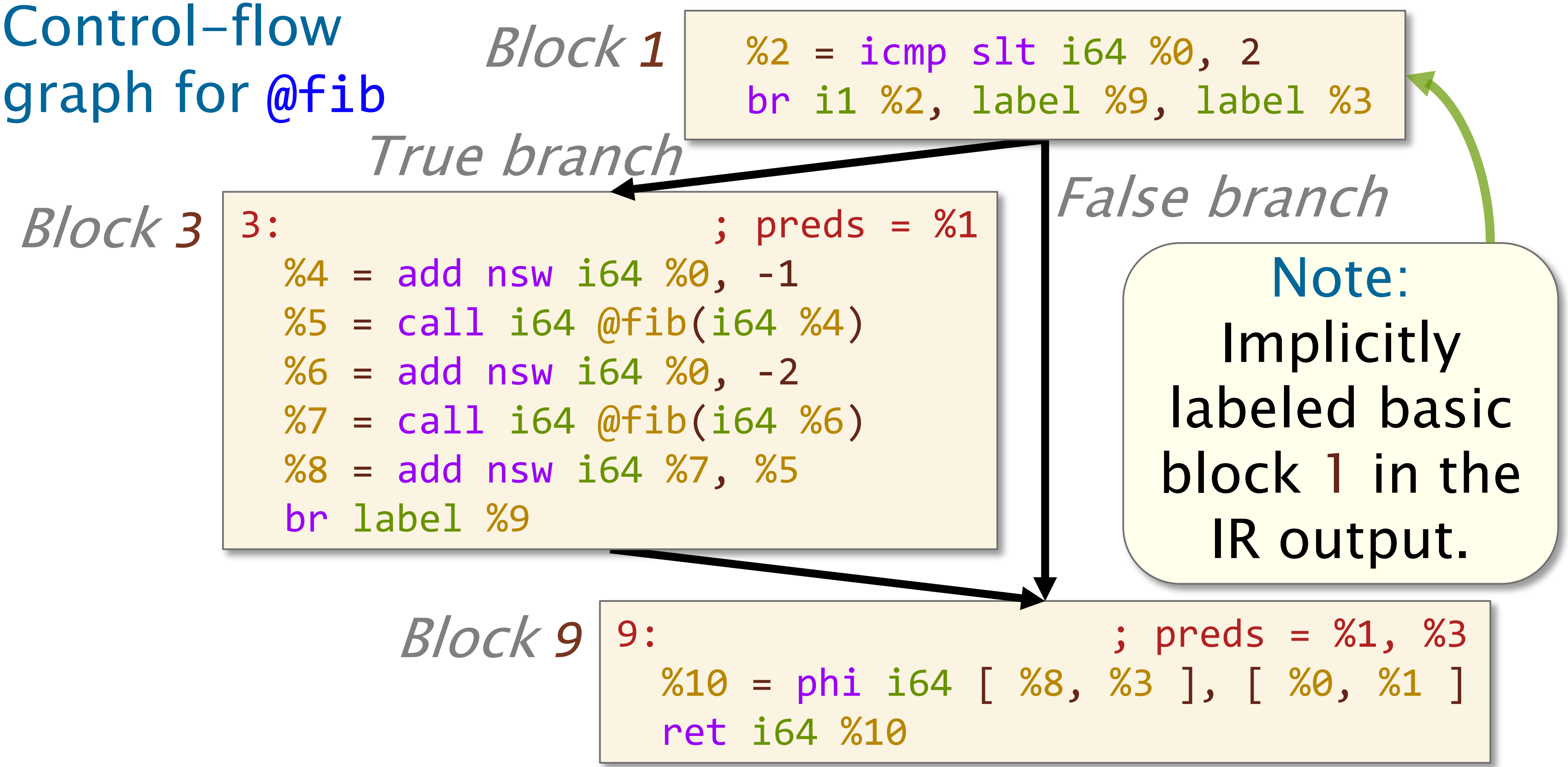
LLVM IR fib.ll

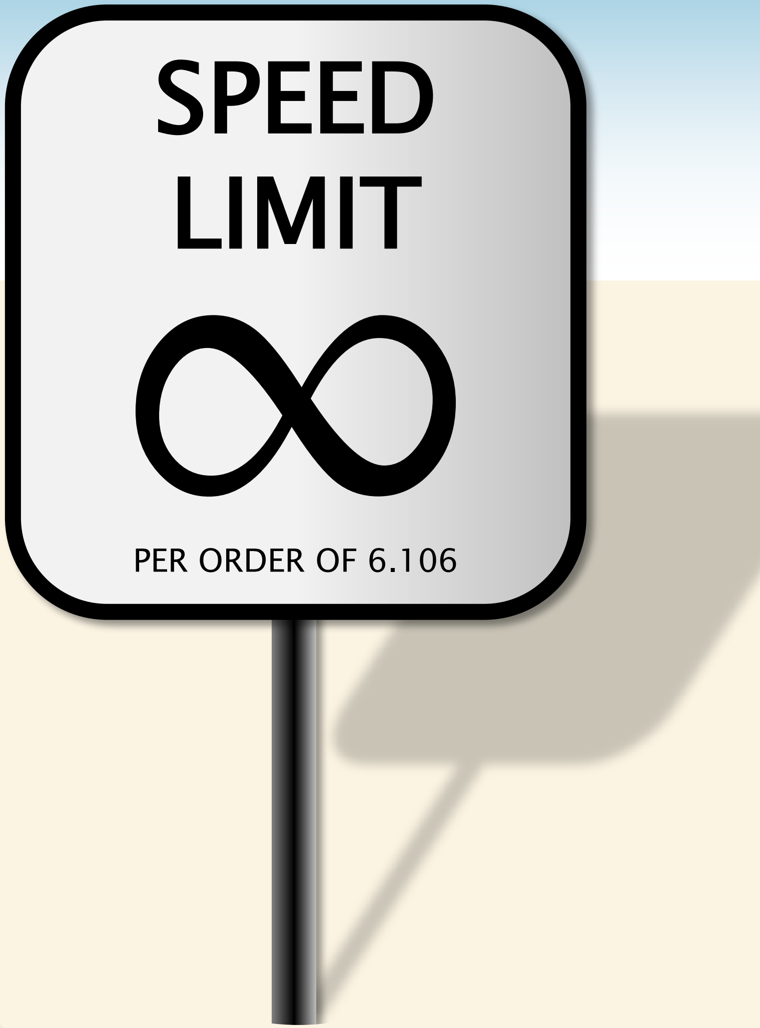
```
define i64 @fib(i64 %0) {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                                    ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                    ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Control-Flow Graphs (CFGs)

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

Control-flow graph for @fib





C Conditionals in LLVM IR

C Conditionals

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

C code `fib.c`

```c
int64_t fib(int64_t n)
{
  if (n < 2)
    …
  return …
}
```

LLVM IR `fib.ll`

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

3:                            ; preds = %1
  …

9:                                      %1, %3
  …
  ret i64 %10
}
```

The comparison in C becomes an `icmp` instruction, with a flag that describes the comparison.

The `slt` flag means "signed less than."

Arguments of a Conditional Branch

The conditional branch in LLVM IR takes as operands a 1-bit integer and two basic-blocks, thereby producing two control-flow edges.

LLVM IR `fib.ll`

Predicate

Destination block if the predicate is true.

Destination block if the predicate is false.

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

3:                                    ; preds = %1
  …

9:                                    ; preds = %1, %3
  ret i64 %0
}
```

Unconditional Branches

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

LLVM IR

```
3:                    ; preds = %1
  …
  %8 = add nsw i64 %7, %5
  br label %9
```

Unconditional branch to basic block 9.

An unconditional branch terminates its basic block and produces one control-flow edge.

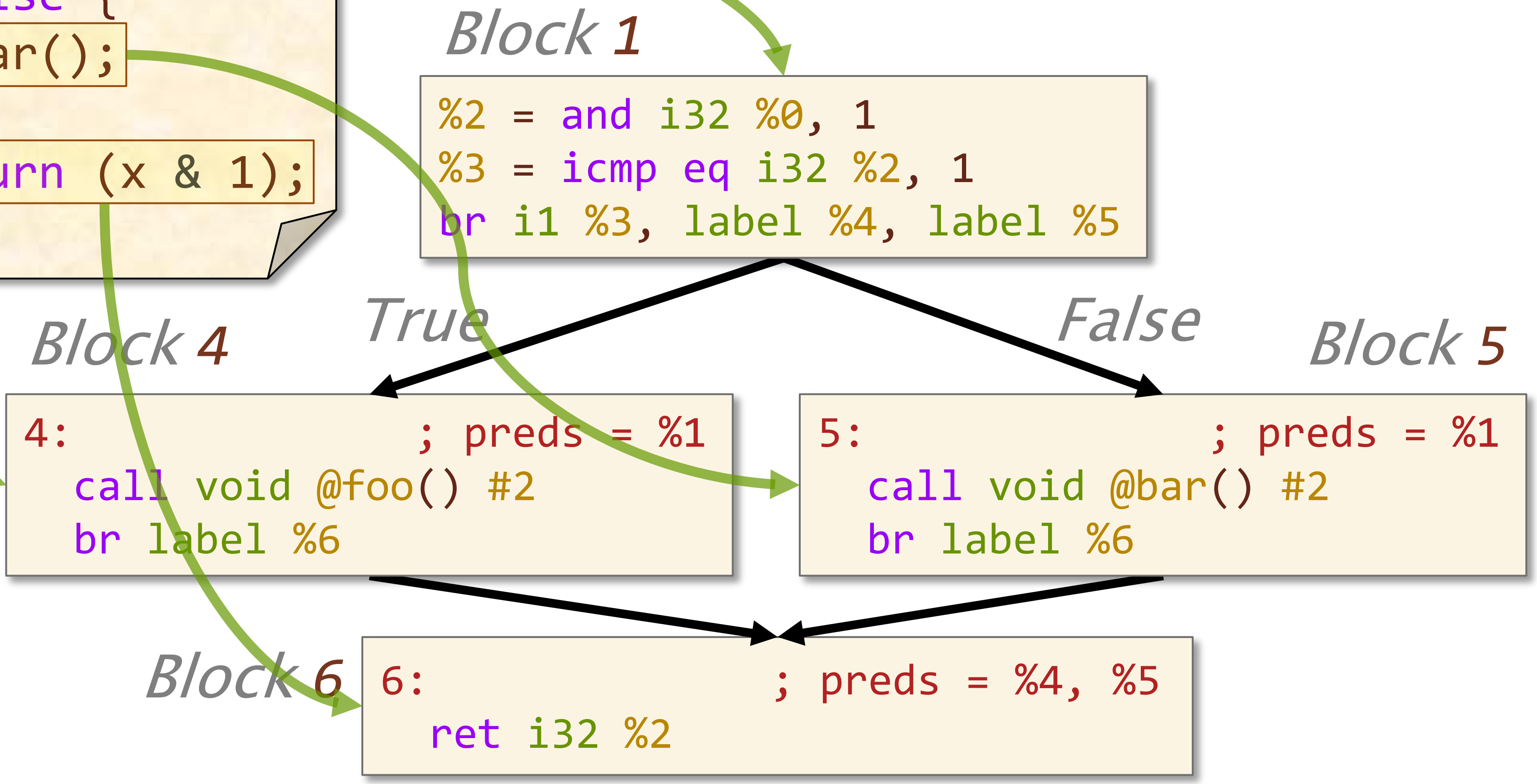
Aside: C Conditionals in General

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

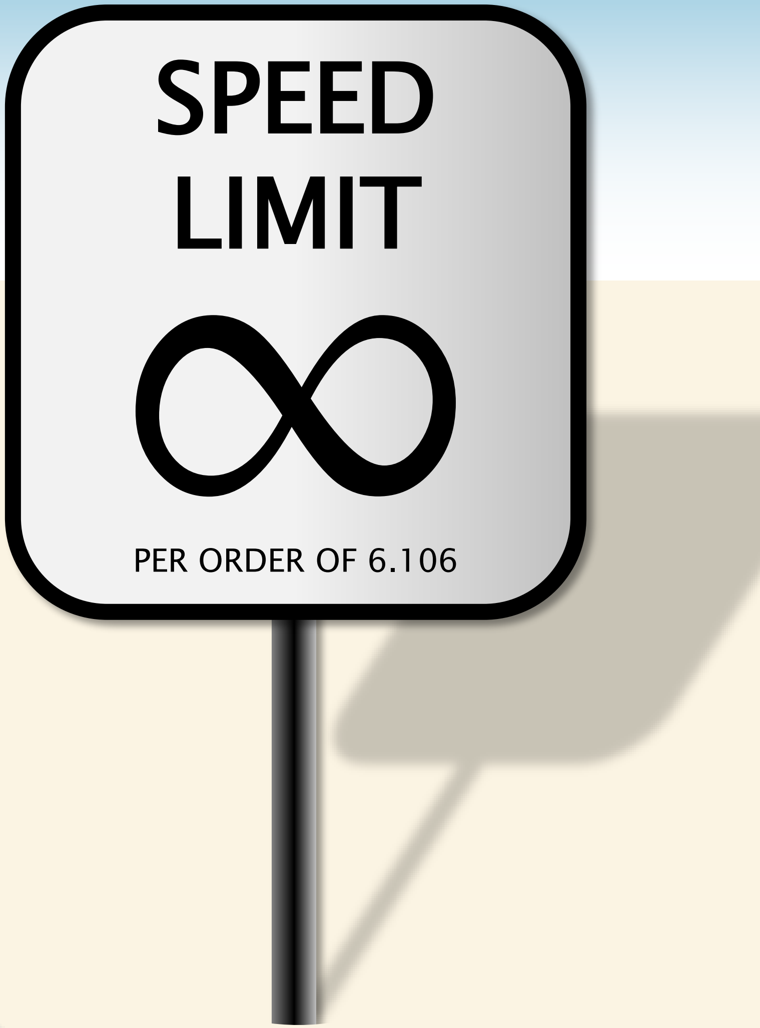
C code

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

Control-flow graph



Static Single Assignment Form



Static Single Assignment

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

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

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

C code

```
fib(n-1) + fib(n-2)
```

SSA Problem

PROBLEM: How does LLVM represent a variable whose value depends on control flow?

C code

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

NOTE: This code generates the same LLVM IR as `fib.c`.

How does LLVM IR express this value, which depends on the conditional?

Control-Flow-Dependent Values

ANSWER: The phi instruction specifies the value of a register depending on control flow.

C code

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

Simplified CFG

Block 1

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

Block 3

```
3:                ; preds = %1
  …
  %8 = add nsw i64 %7, %5
  br label %9
```

Block 9

```
9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
```

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*.

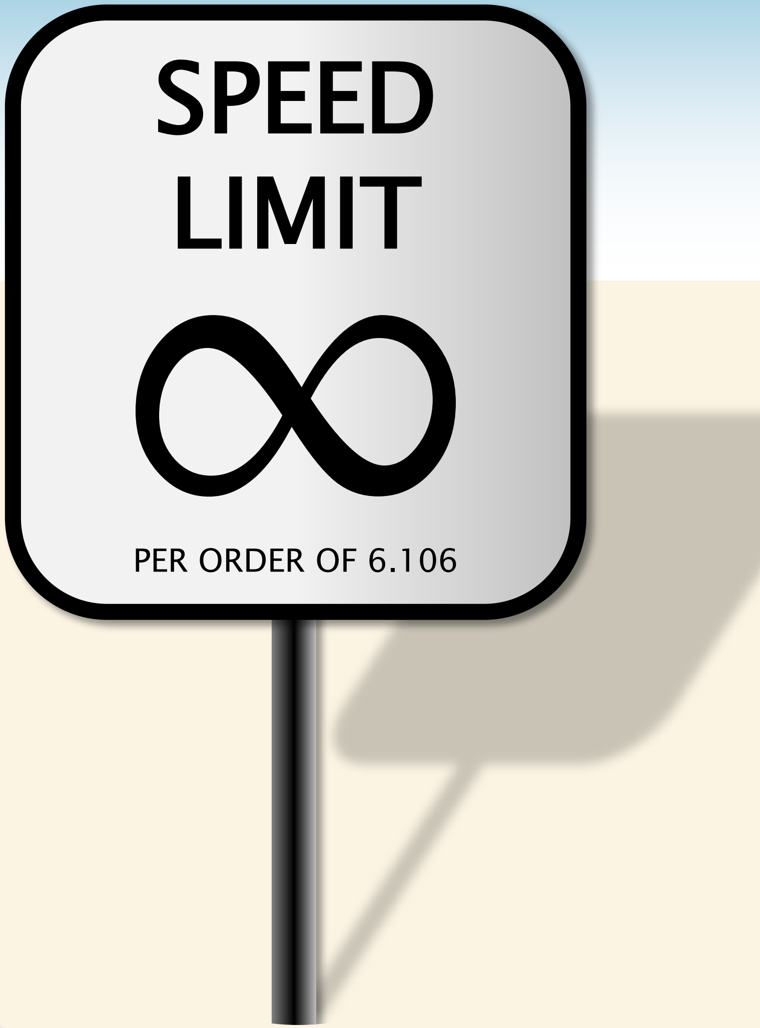
Block 9

```
9:                             ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
```

Adopt the value %8 if control comes from block 3.

Adopt the value %0 if control comes from block 1.

The phi instruction does not correspond to a real instruction in assembly.



Putting It Together: From fib.c to fib.ll

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                                    ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                    ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From `fib.c` to `fib.ll`

C code `fib.c`

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

LLVM IR `fib.ll`

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From fib.c to fib.ll

C code fib.c

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

LLVM IR fib.ll

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

3:                                 ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                 ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

From `fib.c` to `fib.ll`

C code `fib.c`

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

LLVM IR `fib.ll`

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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

3:                              ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                              ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Summary: C to LLVM IR

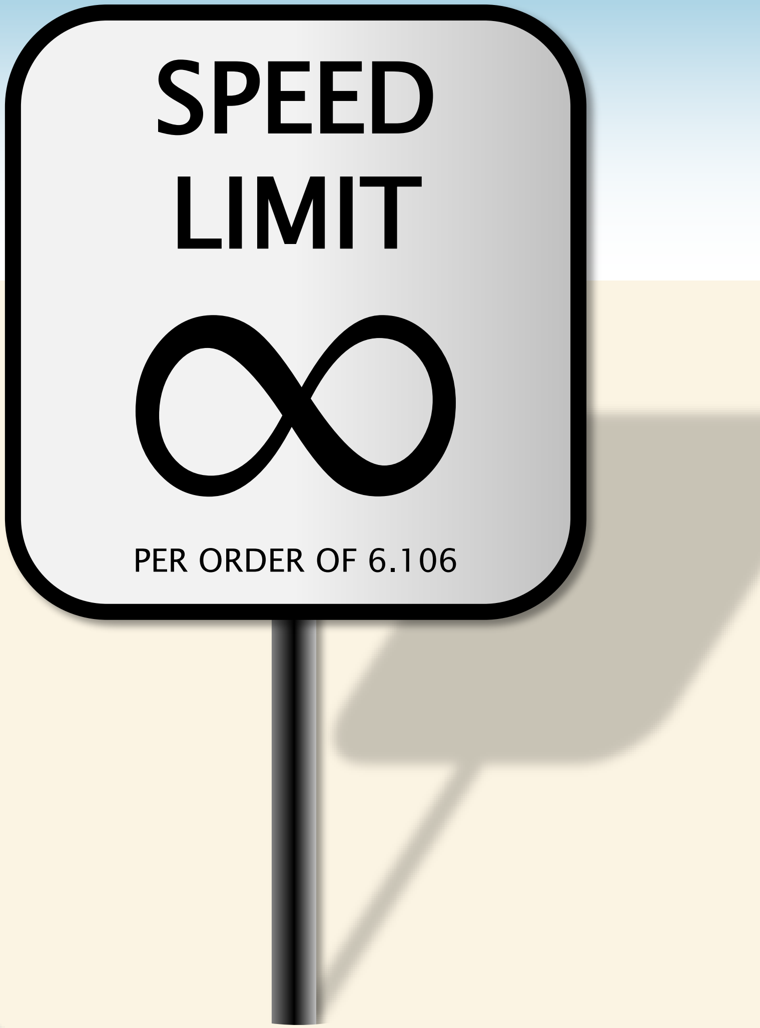
To transform C into LLVM IR:

- Each function is broken down into ***basic blocks***.
- Control-flow constructs (i.e., conditionals and loops) are implemented using ***branches*** between basic blocks.
- Other statements and expressions are implemented using primitive operations that may store values in ***LLVM registers***.

We'll explore more LLVM IR if time permits.

- Feel free to explore LLVM IR yourself, e.g., with Compiler Explorer.

LLVM IR to Assembly



Comparing LLVM IR and Assembly

LLVM IR is structurally similar to assembly.

LLVM IR code fib.ll

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

3:                        ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                        ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
          jl        LBB0_2
          leaq      -1(%rbx), %rdi
          callq     _fib
          movq      %rax, %r14
          addq      $-2, %rbx
          movq      %rbx, %rdi
          callq     _fib
          movq      %rax, %rbx
          addq      %r14, %rbx
LBB0_2:
          movq      %rbx, %rax
          popq      %rbx
          popq      %r14
          popq      %rbp
          retq
```

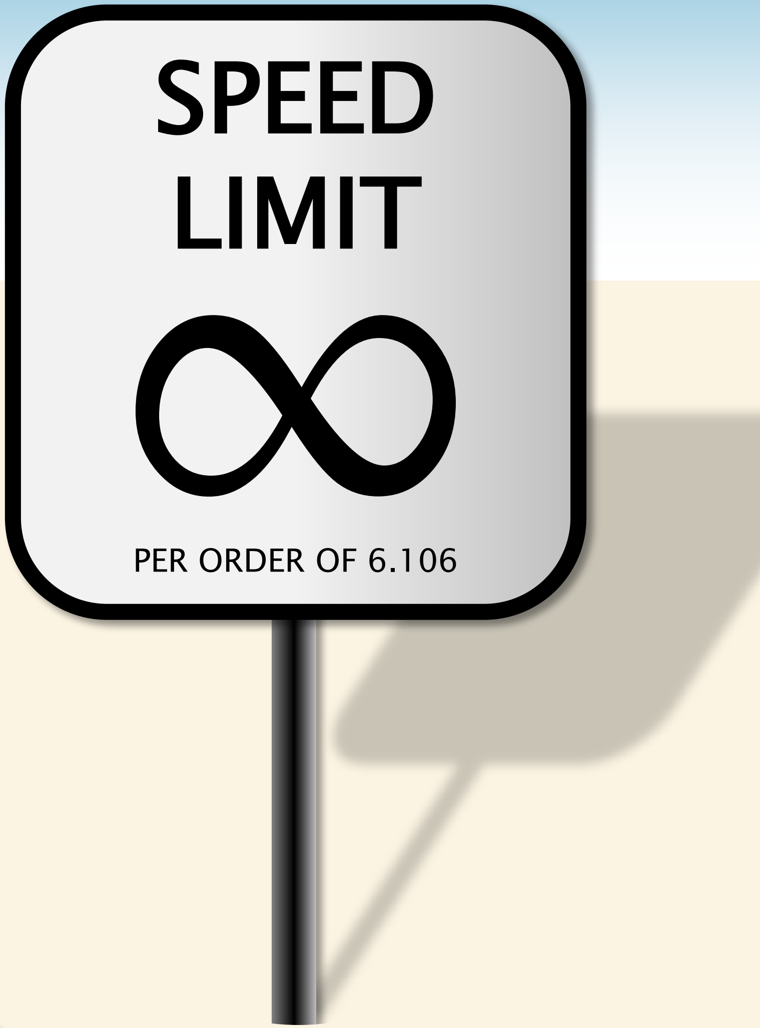
Translating LLVM IR to Assembly

The compiler must perform three main tasks to translate LLVM IR into assembly.

- Select assembly instructions to implement LLVM IR instructions.
- Allocate assembly registers to hold values in LLVM IR registers.
- Coordinate function calls.

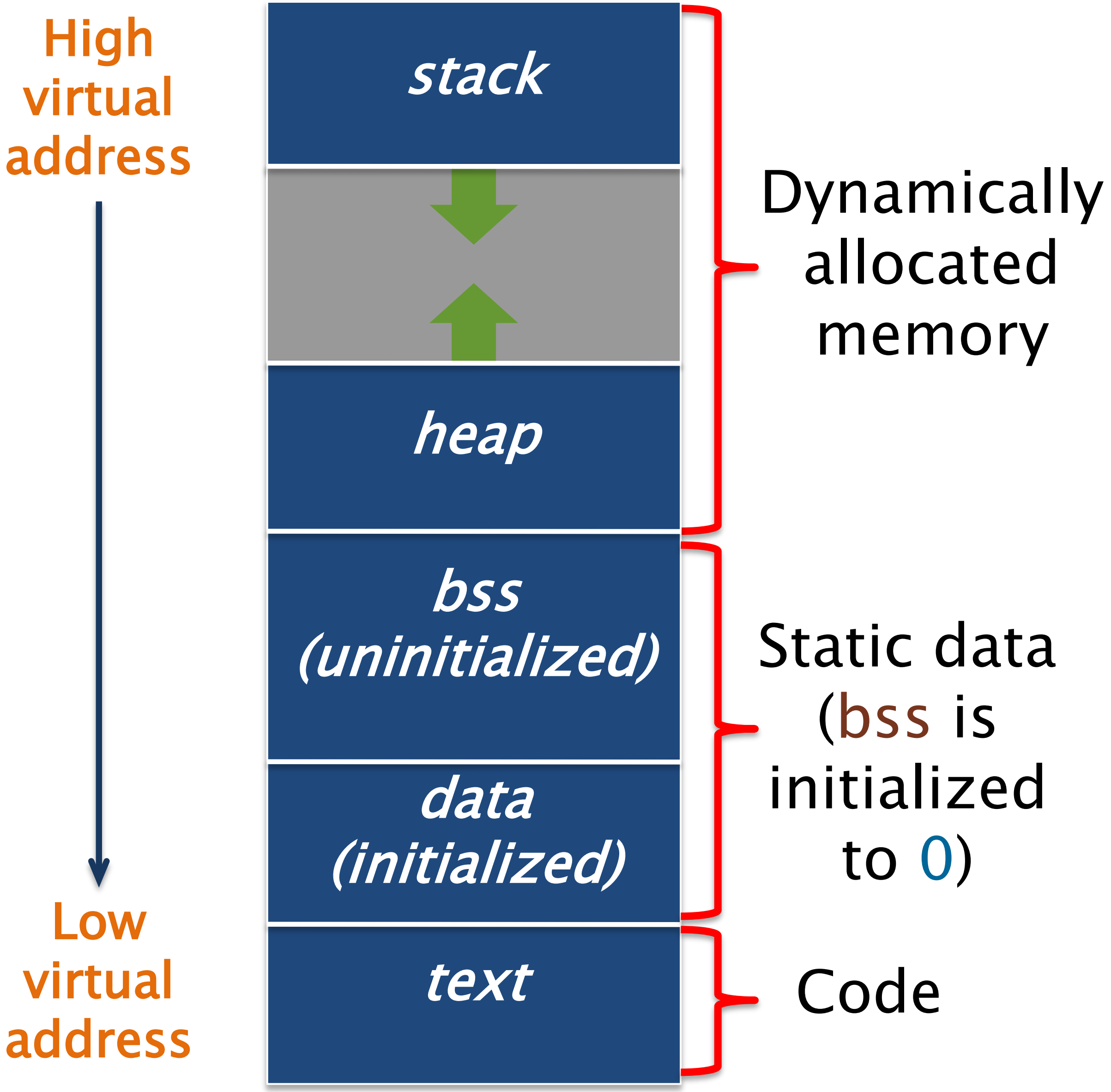
Our main focus

The Linux x86-64 Calling Convention



Layout of a Program in Memory

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

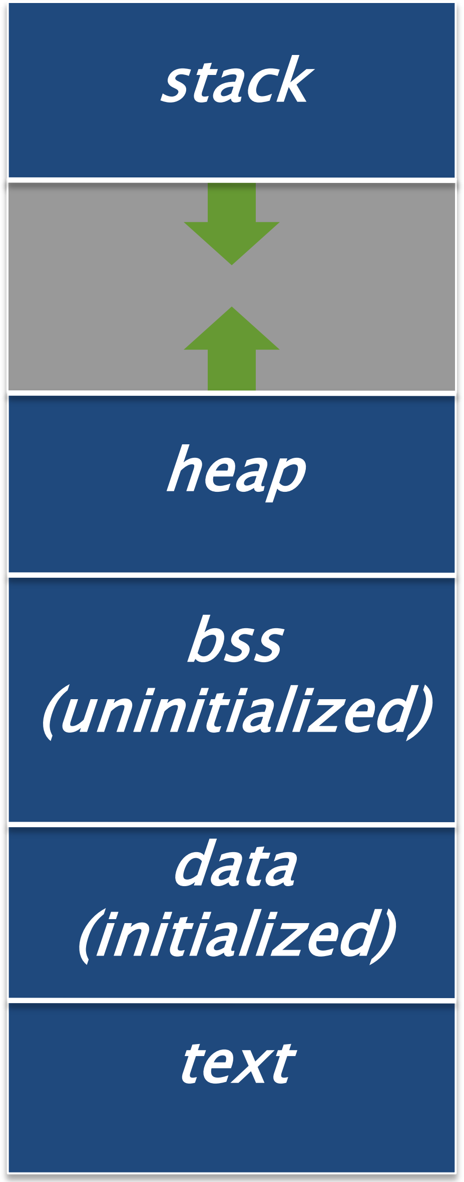


The Call Stack

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

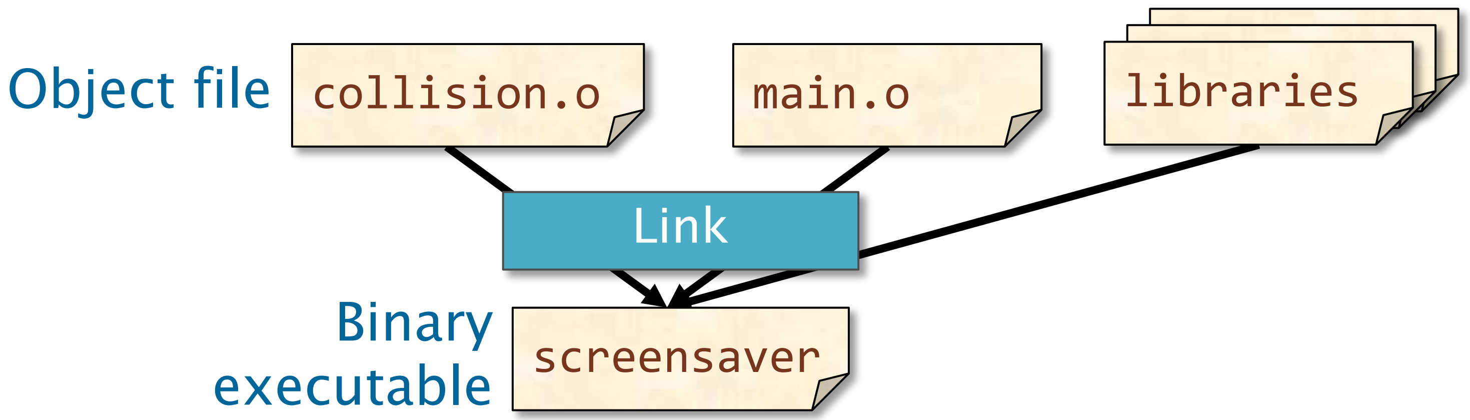
More specifically, what data is stored on the stack?

- Return addresses of function calls.
- 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*.

Organizing the Stack

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

- The ***base pointer*** (or ***frame pointer***), `%rbp`, points to the top of the current stack frame.
- The ***stack pointer***, `%rsp`, points to the bottom of the current stack frame.

Managing Return Addresses

The `call` and `ret` instructions manage return addresses using the stack and the instruction pointer, `%rip`.

- A `call` instruction pushes `%rip` onto the stack and jumps to the specified function.
- To return to the caller, a `ret` instruction pops `%rip` from the stack.

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

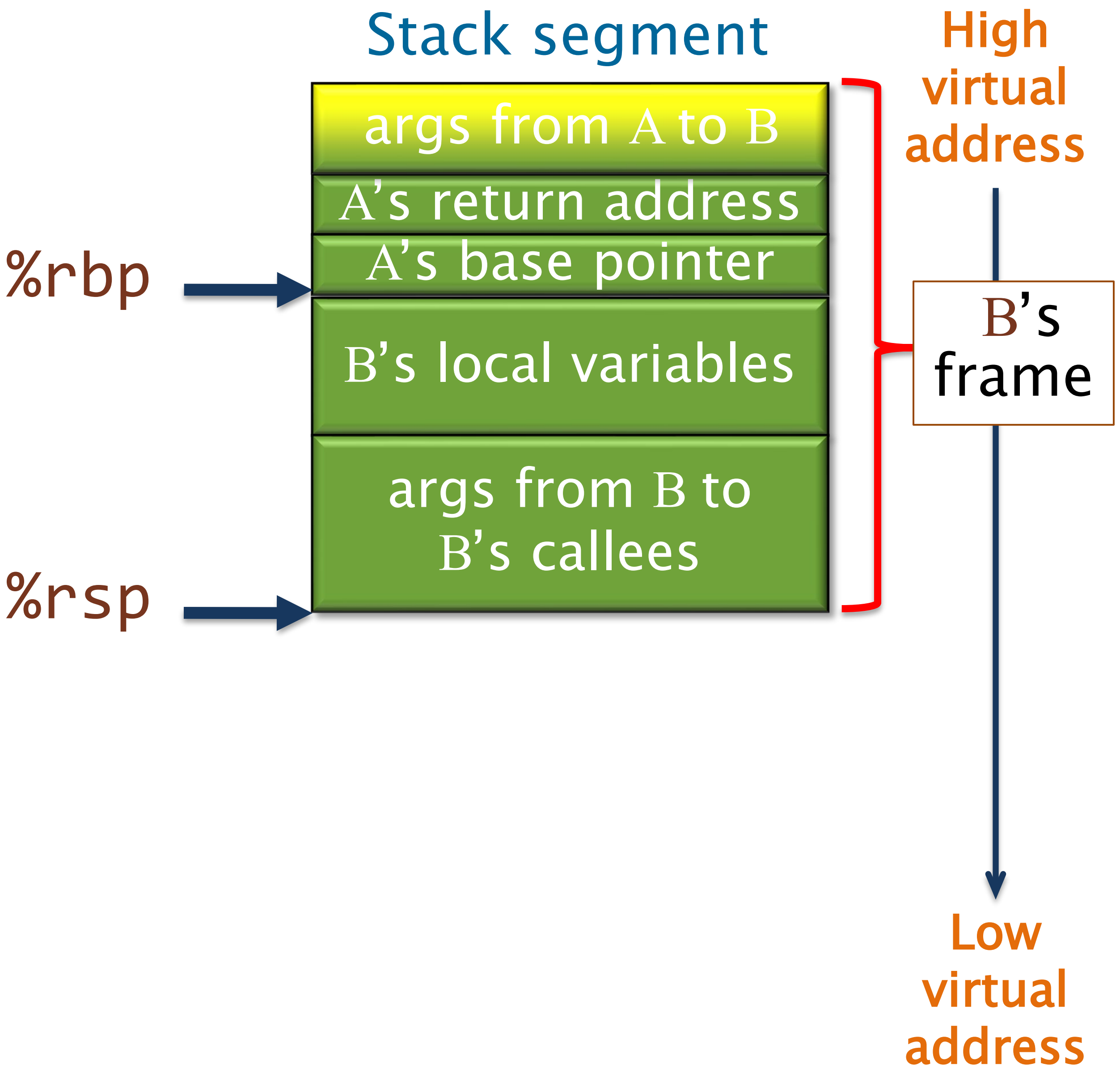
C linkage	64-bit name	32-bit name	16-bit name	8-bit name(s)
Return value	%rax	%eax	%ax	%ah, %al
Callee saved	%rbx	%ebx	%bx	%bh, %bl
4th argument	%rcx	%ecx	%cx	%ch, %cl
3rd argument	%rdx	%edx	%dx	%dh, %dl
2nd argument	%rsi	%esi	%si	%sil
1st argument	%rdi	%edi	%di	%dil
Base pointer	%rbp	%ebp	%bp	%bpl
Stack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

The %xmm0–%xmm7 registers are used for floating-point arguments.

Example: Linux C Subroutine Linkage

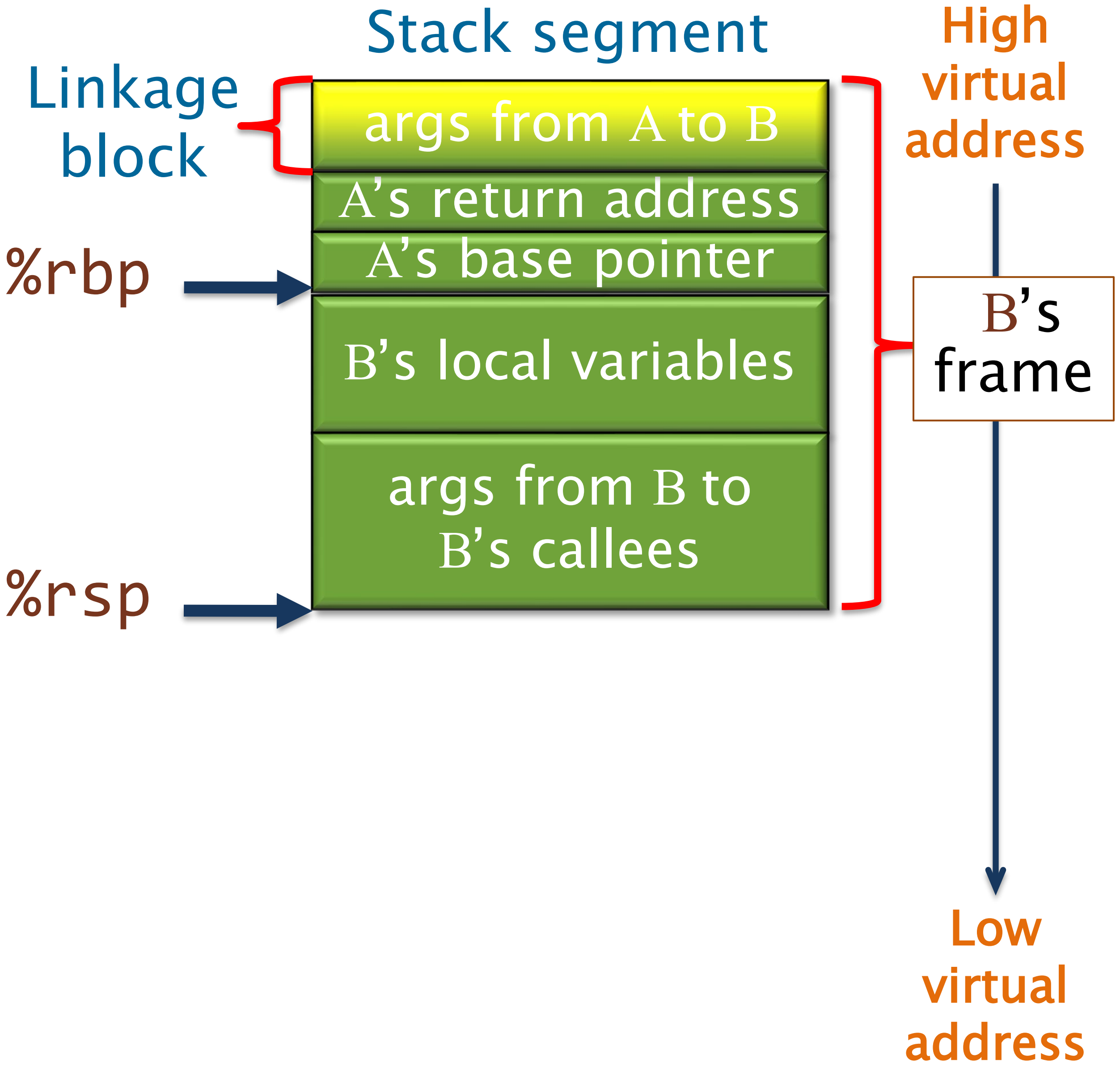
Let's work through an example function call in action.

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



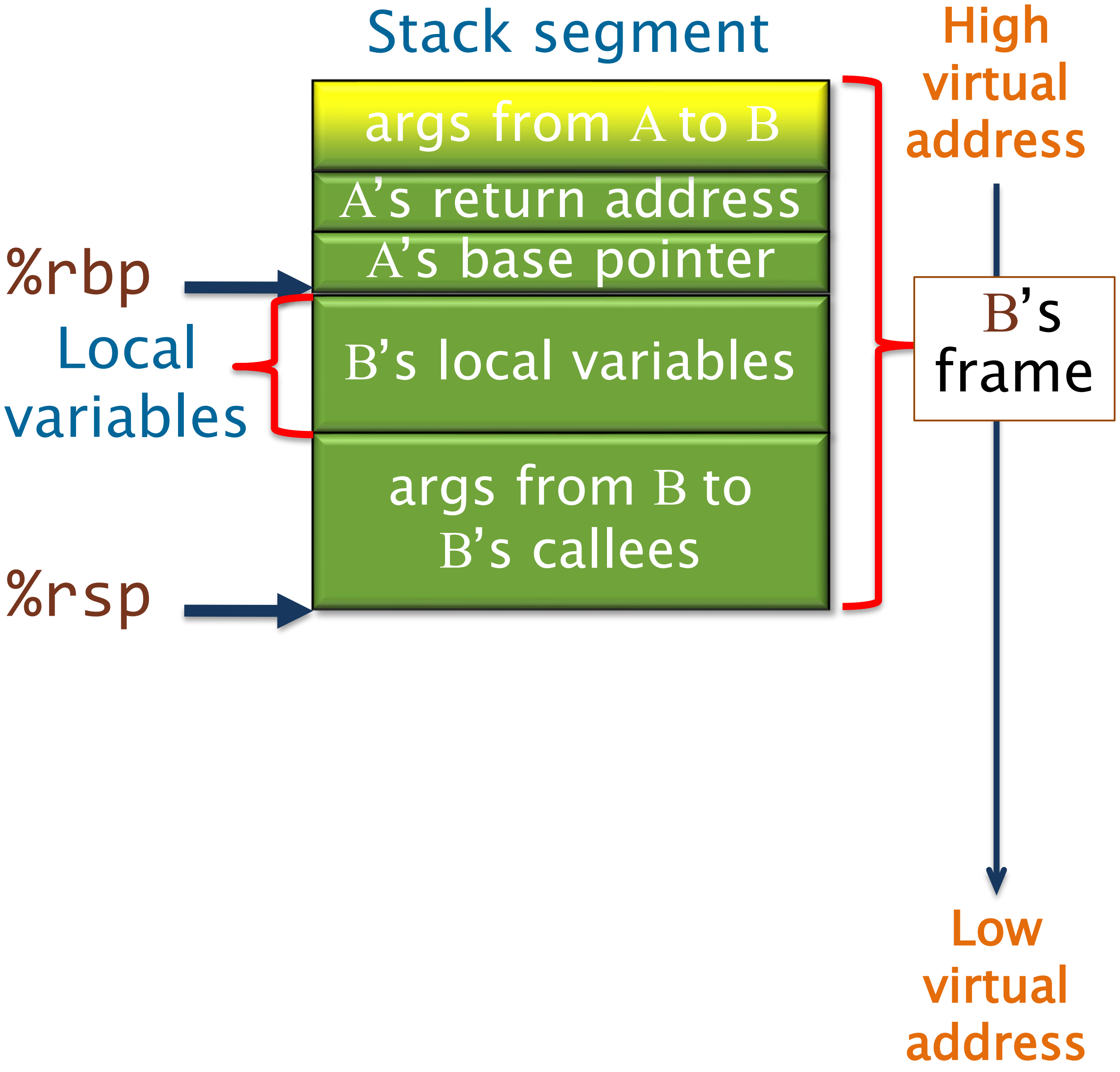
Example: Linux C Subroutine Linkage

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



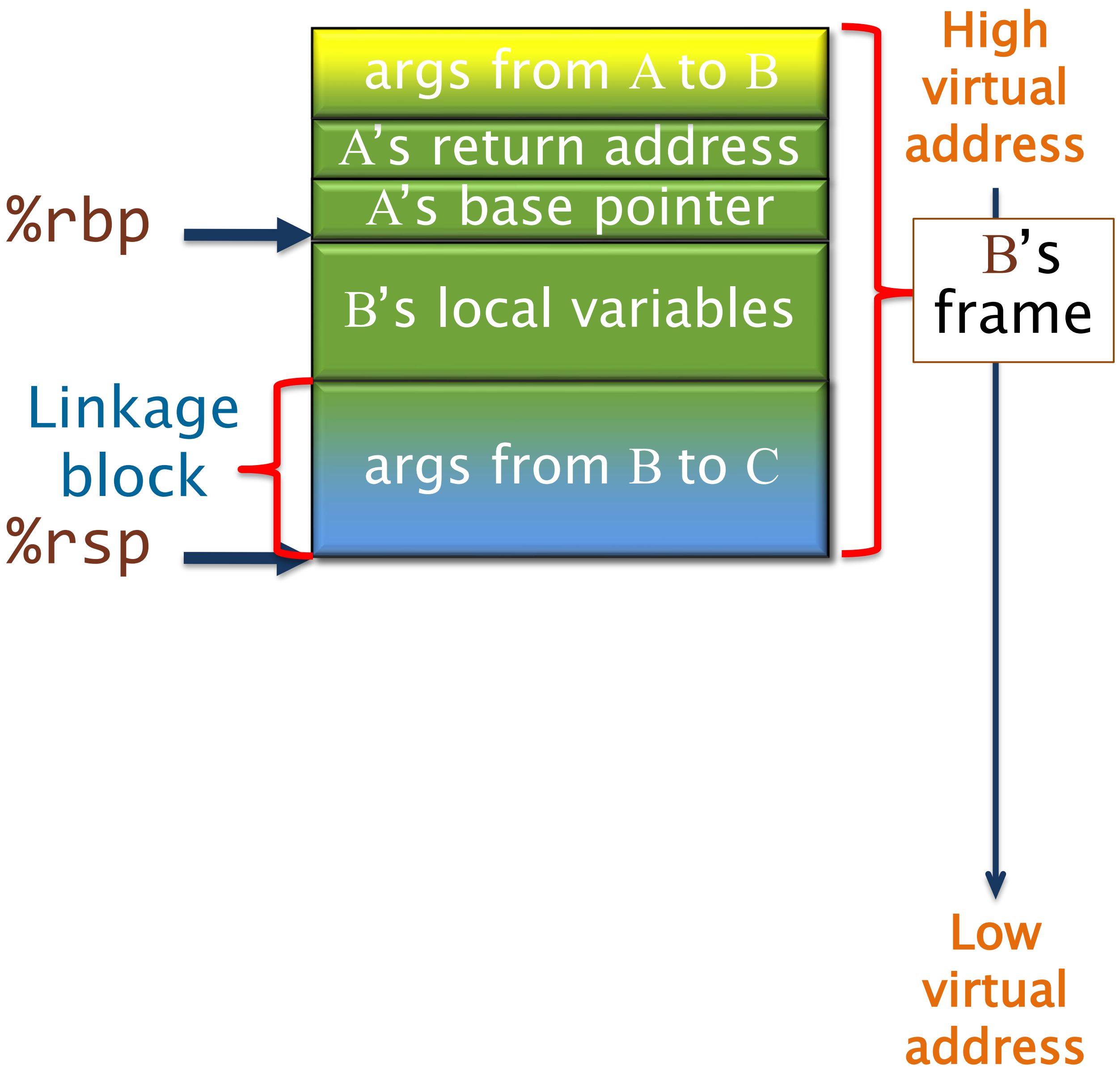
Example: Linux C Subroutine Linkage

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.

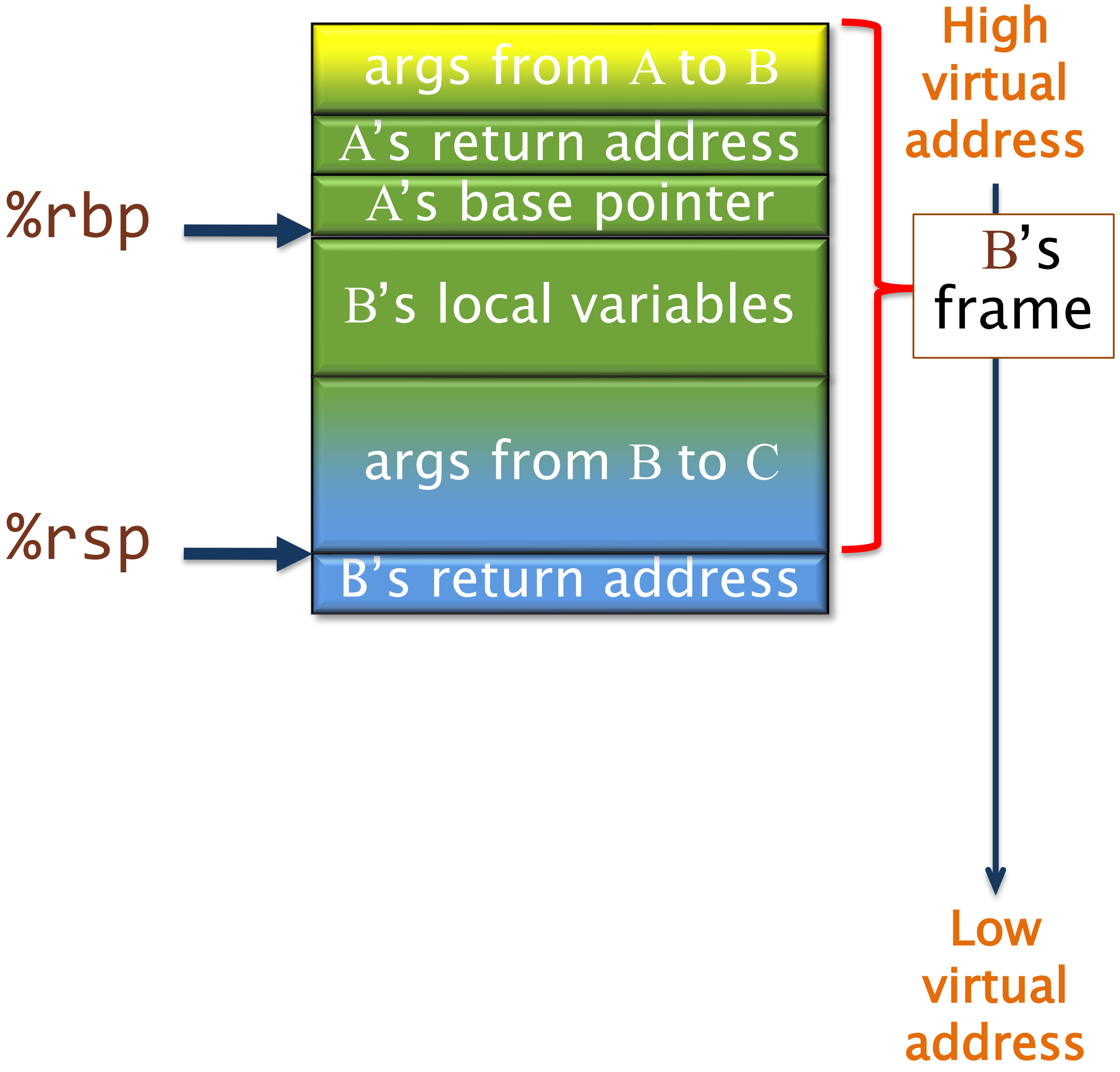


Example: Linux C Subroutine Linkage

Function B executes

```
call C
```

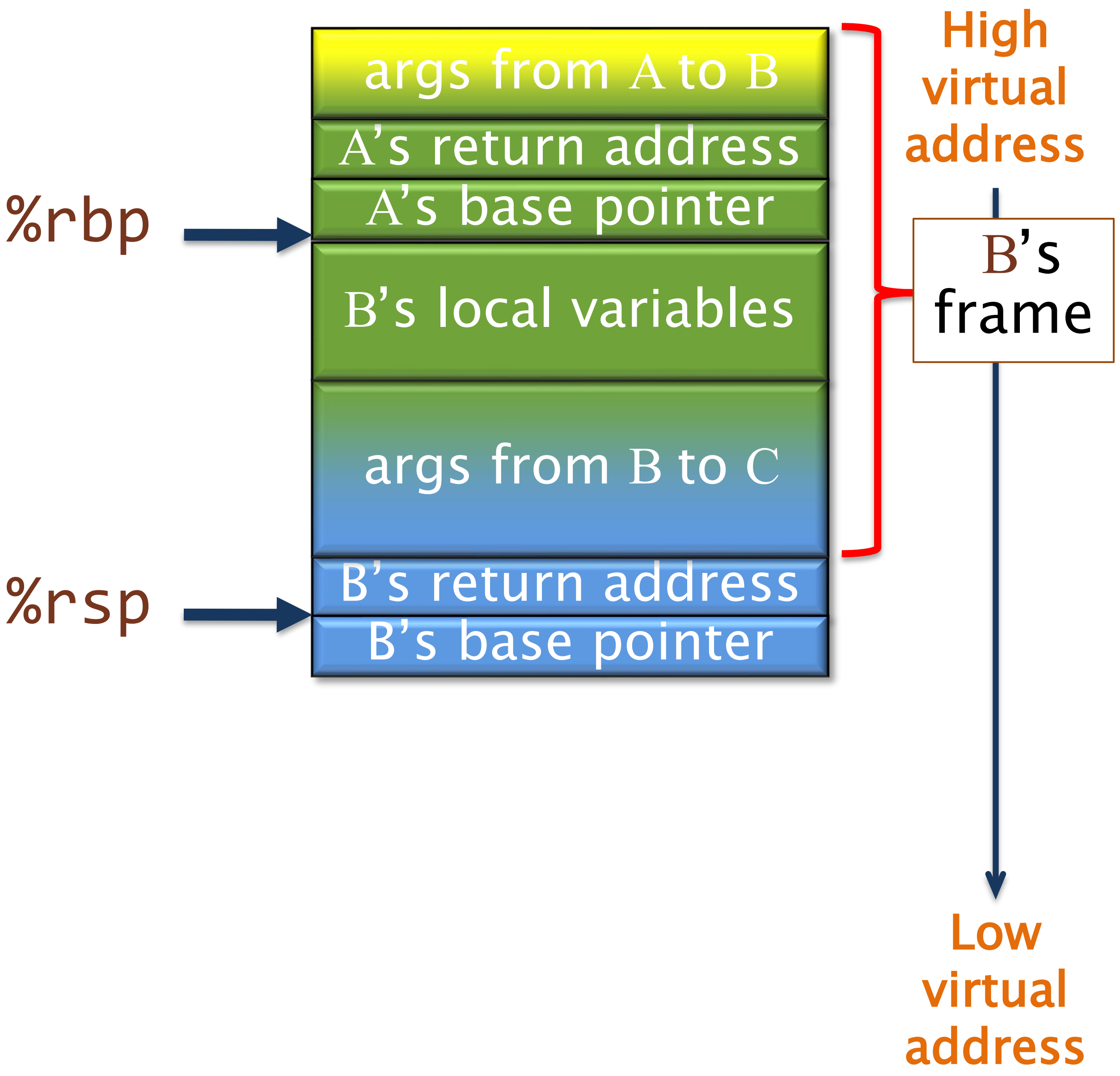
which pushes the return address for B on the stack and transfers control to C.



Example: Linux C Subroutine Linkage

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

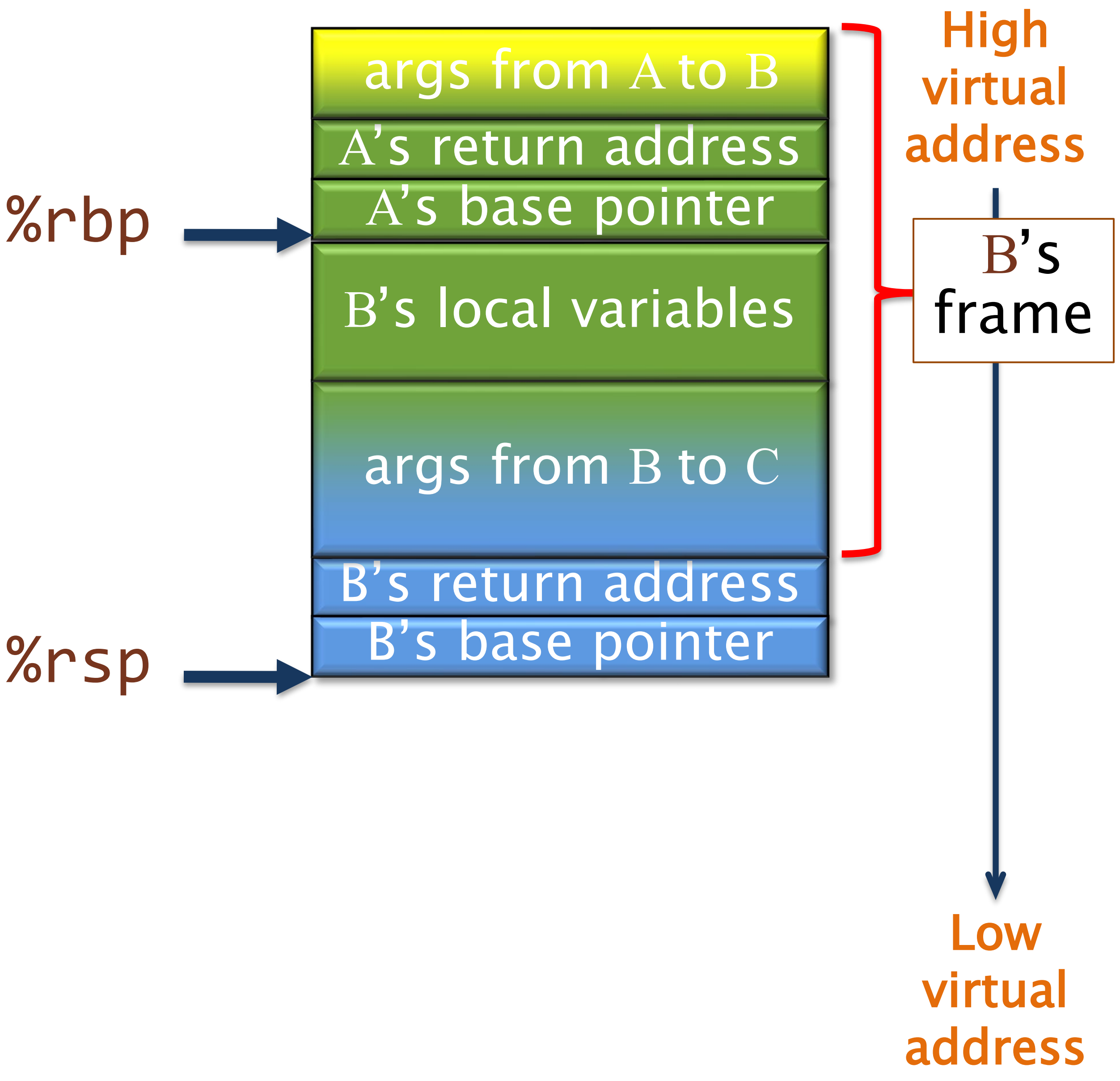
1. Push %rbp — B's base pointer — onto the stack,



Example: Linux C Subroutine Linkage

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

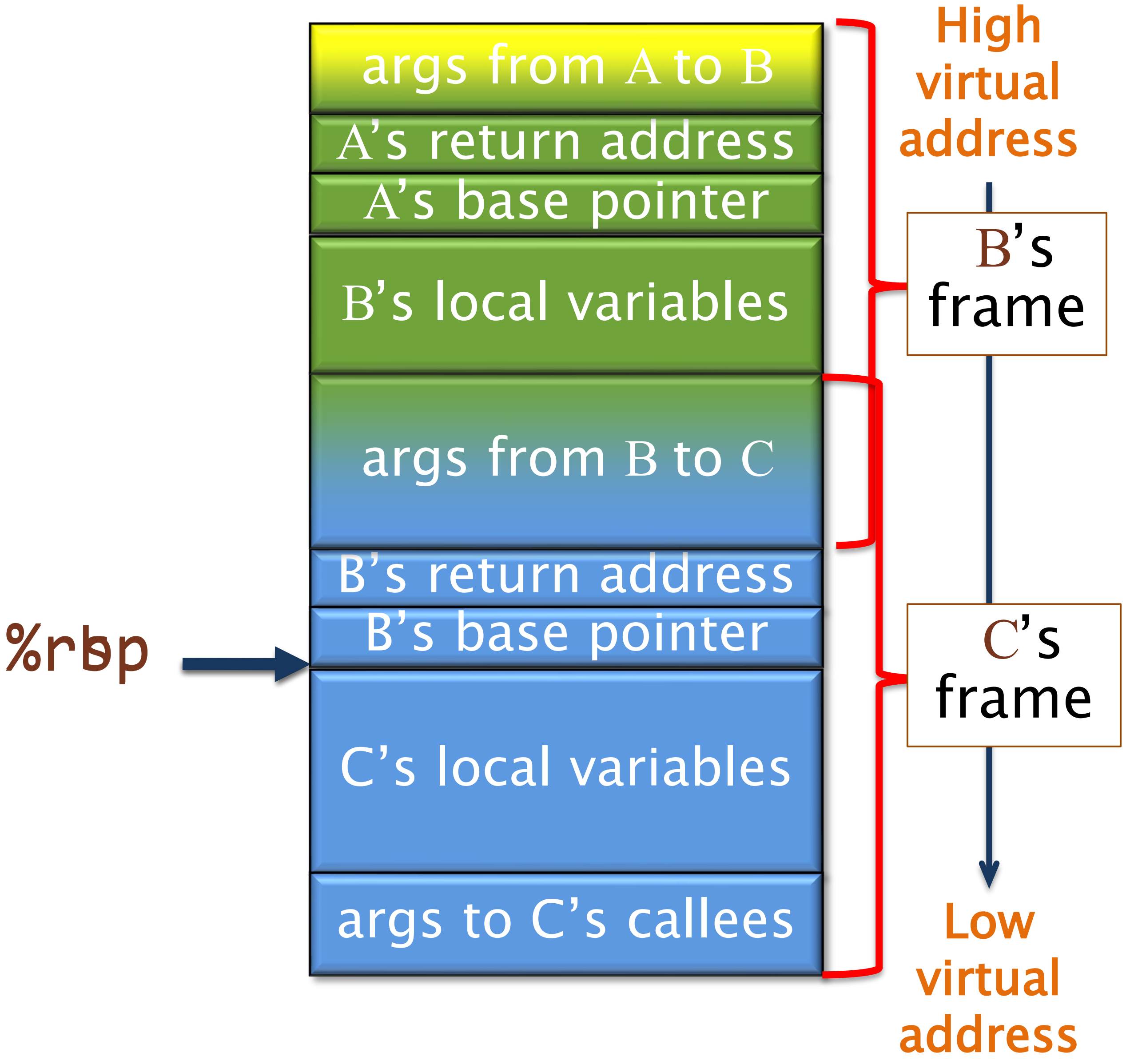
1. Push %rbp — B's base pointer — onto the stack,
2. Set %rbp=%rsp,



Example: Linux C Subroutine Linkage

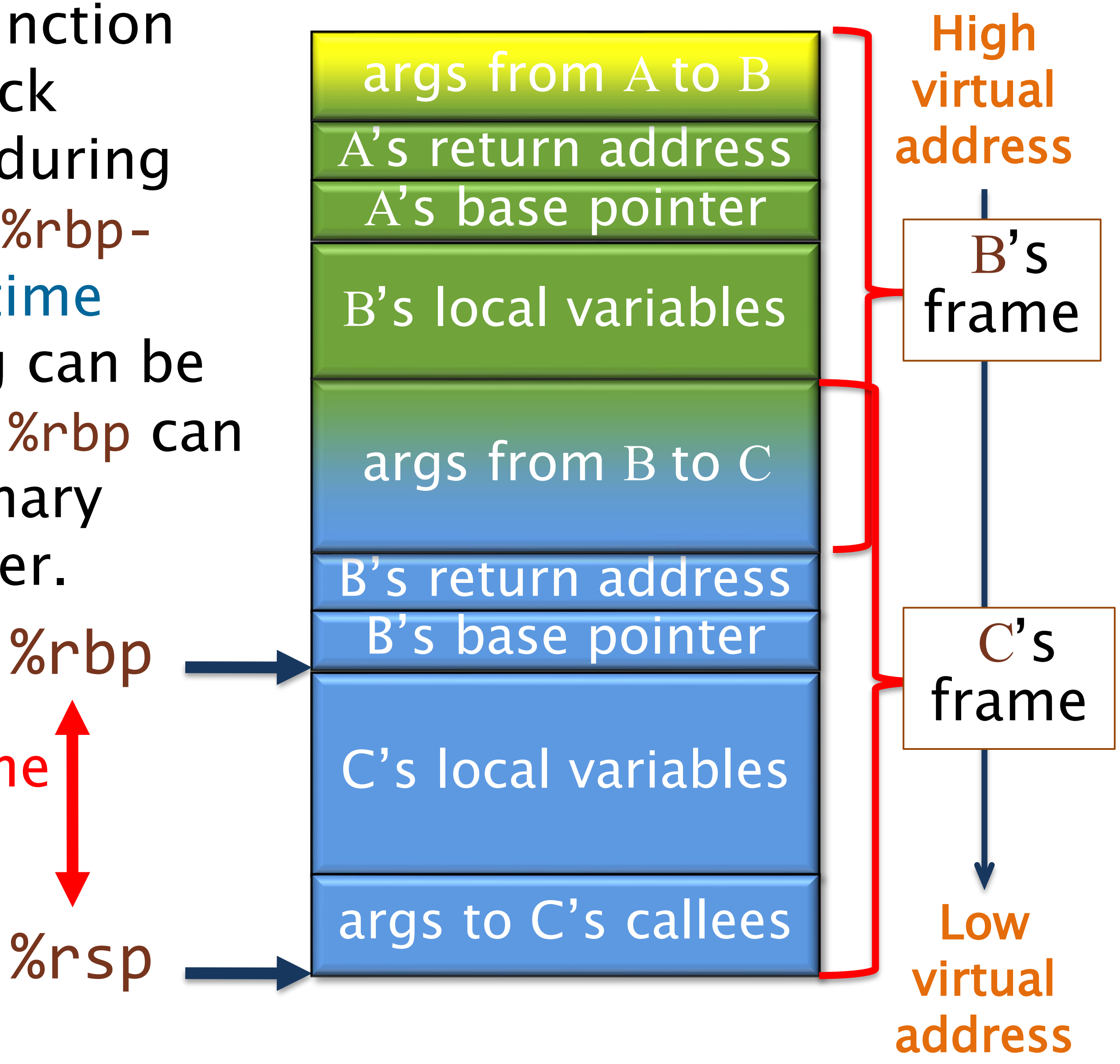
When function C starts, it executes a ***function prologue***:

1. Push %rbp — B's base pointer — onto the stack,
2. Set %rbp=%rsp,
3. Advance %rsp to allocate space for C's local variables and linkage block.



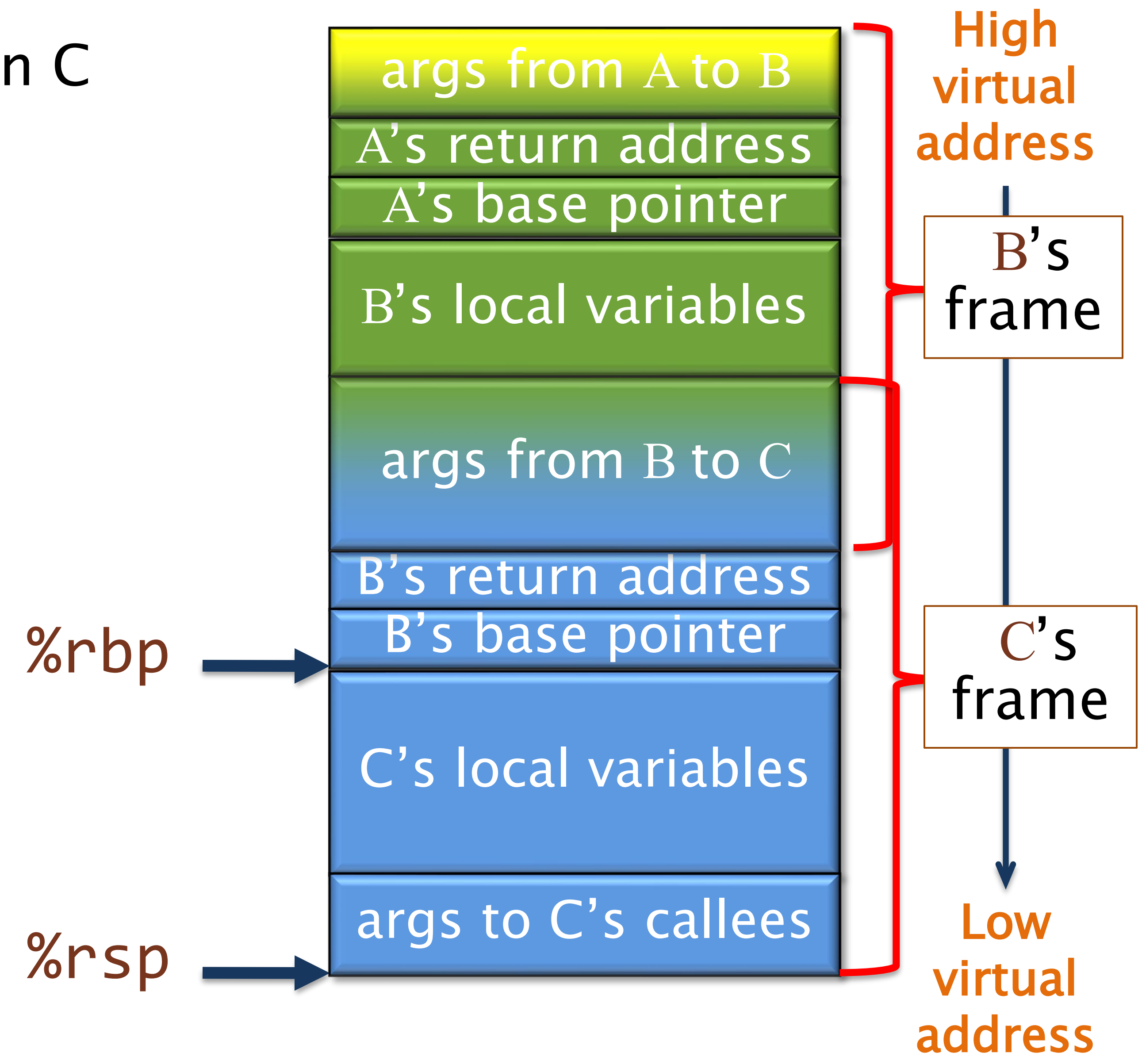
Example: Linux C Subroutine Linkage

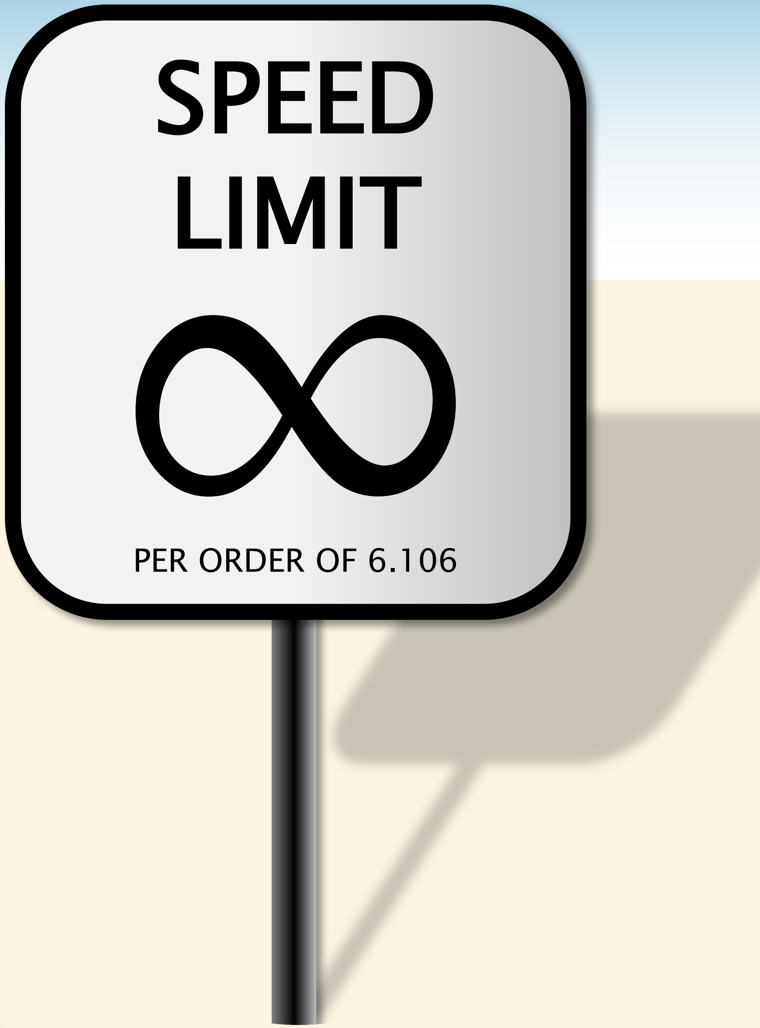
OPTIMIZATION: If a function never performs stack allocations except during function calls (i.e., %rbp-%rsp is a compile-time constant), indexing can be done off %rsp, and %rbp can be used as an ordinary callee-saved register.



Example: Linux C Subroutine Linkage

For more details on C linkage, see the System V ABI.





Putting It Together: From fib.ll to fib.s

Compiling LLVM IR To Assembly

LLVM IR code fib.ll

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

3:                                    ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                    ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i
}
```

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, %rdi
              jl         LBB0_2
              leaq       -1(%rbx), %rdi
              callq      _fib
              movq       %rax, %r14
              addq       $-2, %rbx
              movq       %rbx, %rdi
              callq      _fib
              movq       %rax, %rbx
              addq       %r14, %rbx
LBB0_2:
              movq       %rbx, %rax
              popq       %rbx
              popq       %r14
              popq       %rbp
              retq
```

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

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl        LBB0_2
        leaq      -1(%rbx), %rdi
        callq     _fib
        movq      %rax, %r14
        addq      $-2, %rbx
        movq      %rbx, %rdi
        callq     _fib
        movq      %rax, %rbx
        addq      %r14, %rbx
LBB0_2:
        movq      %rbx, %rax
        popq      %rbx
        popq      %r14
```


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

3:                      ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                      ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl        LBB0_2
        leaq      -1(%rbx), %rdi
        callq     _fib
        movq      %rax, %r14
        addq      $-2, %rbx
        movq      %rbx, %rdi
        callq     _fib
        movq      %rax, %rbx
        addq      %r14, %rbx
LBB0_2:
        movq      %rbx, %rax
        popq      %rbx
        popq      %r14
```

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

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

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                        ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                        ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
```

```
pushq   %r14
pushq   %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 %0) {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
         jl         LBB0_2
         leaq       -1(%rbx), %rdi
         callq      _fib
         movq       %rax, %r14
         addq       $-2, %rbx
         ...
         popq       %r14
```

```
movq    %rdi, %rbx
```

Register %rdi stores the function argument n.

Copy the incoming argument n into %rbx.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

```
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, %rdi
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
```

Compare n against 2.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl         LBB0_2
        leaq       -1(%rbx), %rdi
        callq      _fib
        movq       %rax, %r14
        addq       $-2, %rbx
        movq       %rbx, %rdi
        callq      _fib
        movq       %rax, %rbx
        addq       %r14, %rbx
LBB0_2:
        movq       %rbx, %rax
        popq       %rbx
        popq       %r14
```

```
jl      LBB0_2
```

True side of LLVM branch: If `n < 2`, jump to label `LBB0_2`.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        ...
        popq        %r14
        popq        %rbp
```

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

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

The lea opcode evaluates the given address stores the result in the destination register. It does not access memory.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
            jl          LBB0_2
            leaq        -1(%rbx), %rdi
            callq       _fib
            movq        %rax, %r14
            addq        $-2, %rbx
            movq        %rbx, %rdi
            callq       _fib
            movq        %rax, %rbx
            addq        %r14, %rbx
LBB0_2:
            movq        %rbx, %rax
```

```
callq   _fib
```

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Assembly code fib.s

```
        .globl      _fib
        ...
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
```

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

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

Save the result of calling fib into %r14.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
```

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

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

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

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, %rdi
        jl          LBB0_2
        leaq        -1(%rbx), %rdi
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
```

```
callq   _fib
movq    %rax, %rbx
```

Call fib. The argument n-2 is in %rdi. Store the result in %rbx.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Assembly code fib.s

```
        .globl      _fib
        .p2align    4, 0x90
        ...
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
        popq        %rbp
        retq
```

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

```
addq   %r14, %rbx
```

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Assembly code fib.s

```
        .globl  _fib
        .p2align 4, 0x90
        ...
        callq   _fib
        movq    %rax, %r14
        addq    $-2, %rbx
        movq    %rbx, %rdi
        callq   _fib
        movq    %rax, %rbx
        addq    %r14, %rbx
LBB0_2:
        movq    %rbx, %rax
        popq    %rbx
        popq    %r14
        popq    %rbp
        retq
```

Label for the true side of the LLVM branch.

LBB0_2:

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds =
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Assembly code fib.s

```
        .globl      _fib
        .p2align    4, 0x90
_fib:   ## @fib
        pushq       %rbp
        movq        %rsp, %rbp
        pushq       %r14
                    , %rbx
                    %rdi
                    _2
                    rbx), %rdi

                    , %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
        popq        %rbp
        retq
```

Wait! What happened to this unconditional branch?

In the assembly, LBB0_2 ends up on the fall-through path, so no explicit jmp instruction is required.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                      ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                      ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

```
movq  %rbx, %rax
```

Assembly code fib.s

```
        .globl    _fib
        callq     _fib
        movq      %rax, %r14
        addq      $-2, %rbx
        movq      %rbx, %rdi
        callq     _fib
        movq      %rax, %rbx
        addq      %r14, %rbx
LBB0_2:
        movq      %rbx, %rax
        popq      %rbx
        popq      %r14
        popq      %rbp
        retq
```

Move the result from %rbx into %rax.

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Assembly code fib.s

```
        .globl      _fib
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
        popq        %rbp
        retq
```

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

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

From fib.ll to fib.s

LLVM IR code fib.ll

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

3:                          ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                          ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

```
retq
```

Assembly code fib.s

```
        .globl      _fib
        .p2align    4, 0x90
        ...
        callq       _fib
        movq        %rax, %r14
        addq        $-2, %rbx
        movq        %rbx, %rdi
        callq       _fib
        movq        %rax, %rbx
        addq        %r14, %rbx
LBB0_2:
        movq        %rbx, %rax
        popq        %rbx
        popq        %r14
        popq        %rbp
        retq
```

Return from the function.

Summary: LLVM IR to Assembly

To transform LLVM IR into assembly:

- LLVM IR is translated approximately *line by line* into assembly.
- ISA *instructions* are *chosen* to implement primitive LLVM operations.
- ISA *registers* (and stack space) are *allocated* to store LLVM IR registers.
- Functions and function calls are implemented according to a *calling convention*.

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 with straight-line code in C.
  - C control-flow constructs, such as 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

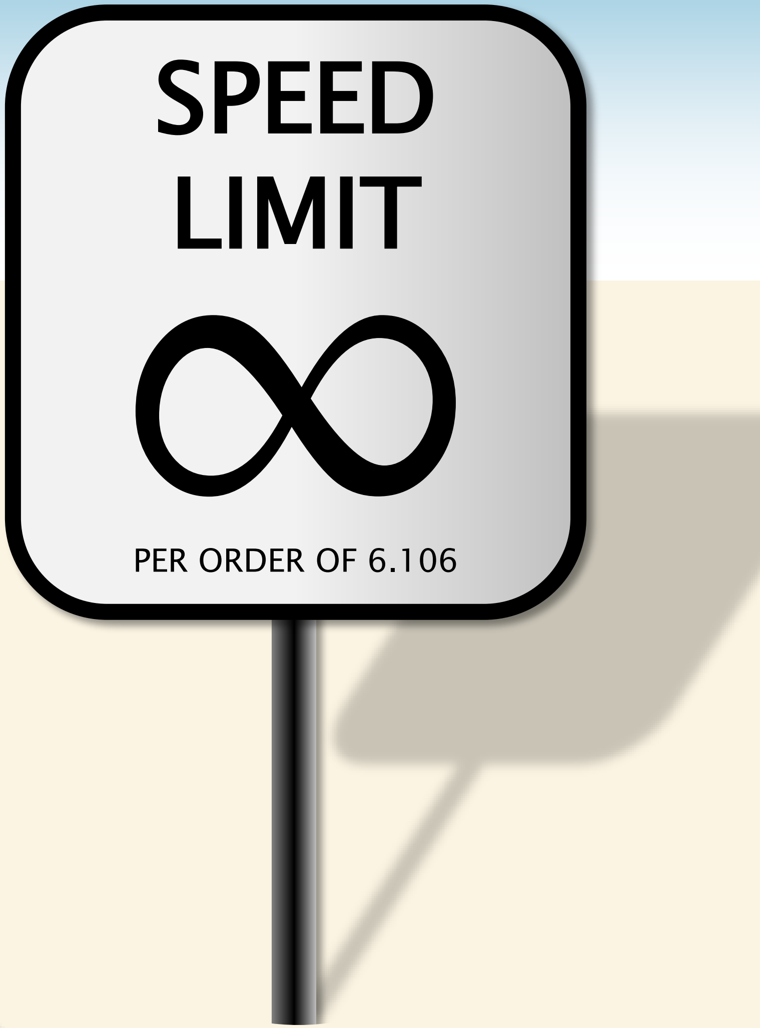
Full details:
Intel Software Developer Manuals (on course website)

C subroutine linkage:
System V Application Binary Interface (on course website)

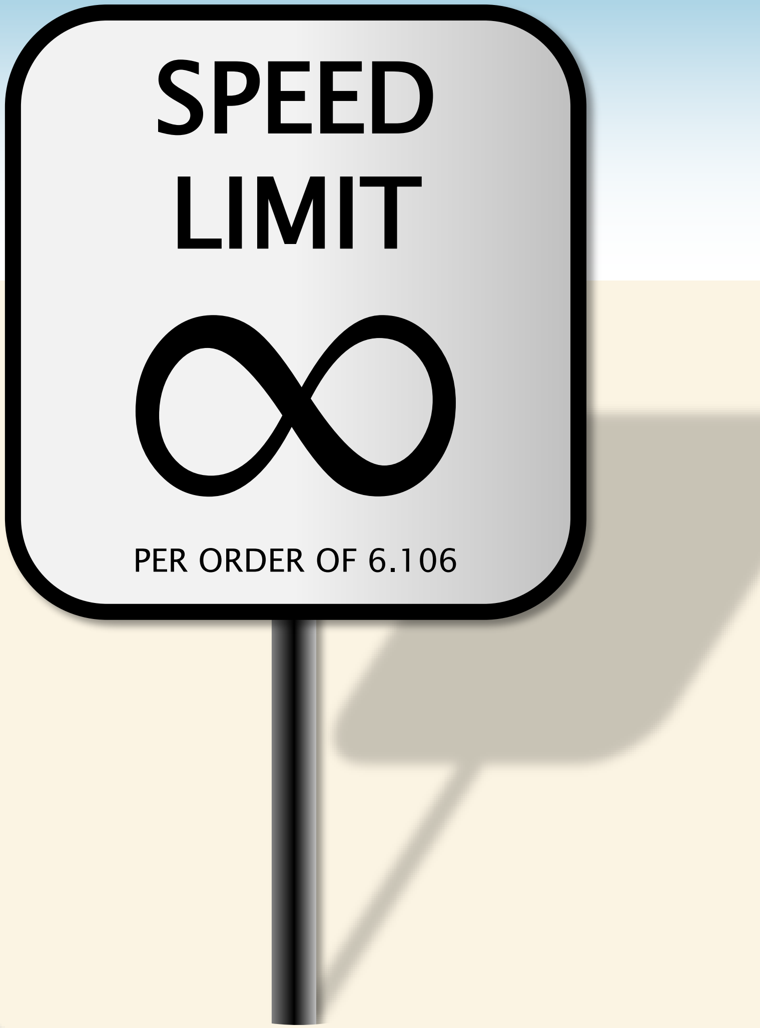
LLVM IR language reference
https://llvm.org/docs/LangRef.html

Compiler intrinsic for inline assembly:
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Optional Slides



Handling LLVM IR Manually



Viewing LLVM IR Manually

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

Source code `fib.c`

```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`

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

3:                                    ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                    ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

Compiling LLVM IR Manually

LLVM IR can be translated directly into assembly.

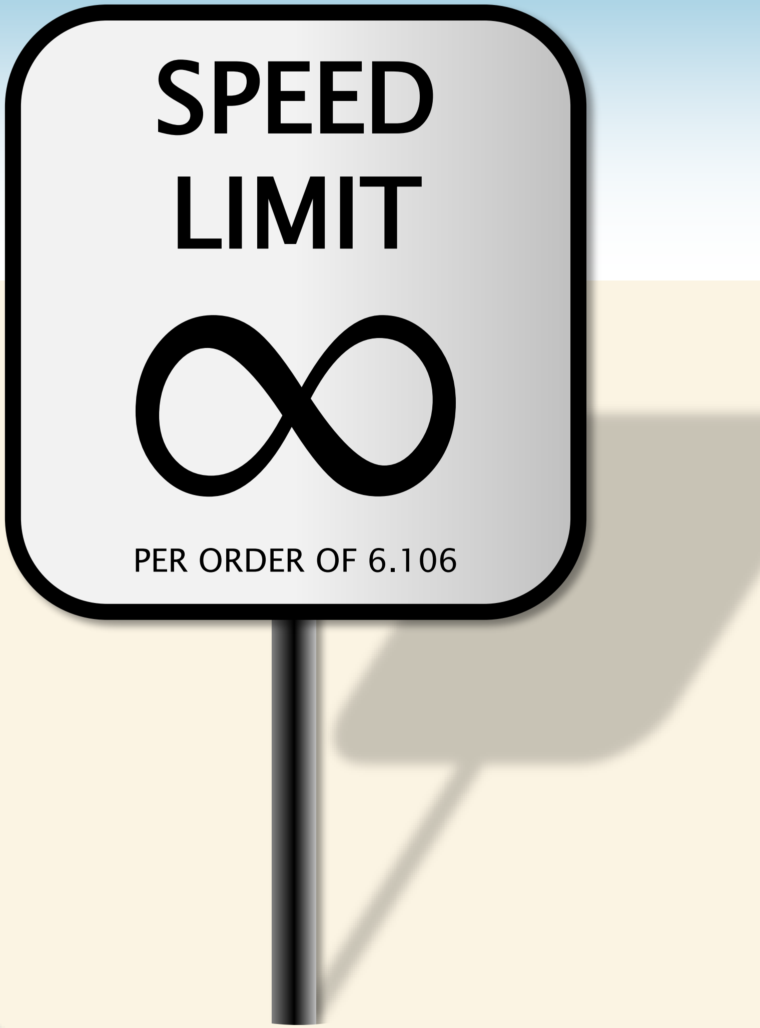
LLVM IR code fib.ll

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

3:                                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

```
$ 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, %rdi
          jl          LBB0_2
          leaq        -1(%rbx), %rdi
          callq       _fib
          movq        %rax, %r14
          addq        $-2, %rbx
          movq        %rbx, %rdi
          callq       _fib
          movq        %rax, %rbx
          addq        %r14, %rbx
LBB0_2:
          movq        %rbx, %rax
          popq        %rbx
          popq        %r14
          popq        %rbp
          retq
```



LLVM IR Overview

LLVM IR Registers

LLVM IR stores values in ***registers***.

- Syntax: %<name>
- LLVM registers are like local variables in C: 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 = call i64 @fib(i64 %4)
%6 = add nsw i64 %0, -2
%7 = call i64 @fib(i64 %6)
%8 = add nsw i64 %7, %5
br label %9
```

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> [flags] <operands>
```

- Syntax for other instructions:

```
<opcode> <operands>
```

- Operands are registers, constants, or “basic blocks,” often with their data type.

Instruction that produces a value.

Instruction that does not produce a value.

```
%4 = add nsw i64 %0, -1
%5 = call i64 @fib(i64 %4)
%6 = add nsw i64 %0, -2
%7 = call i64 @fib(i64 %6)
%8 = add nsw i64 %7, %5
br label %9
```

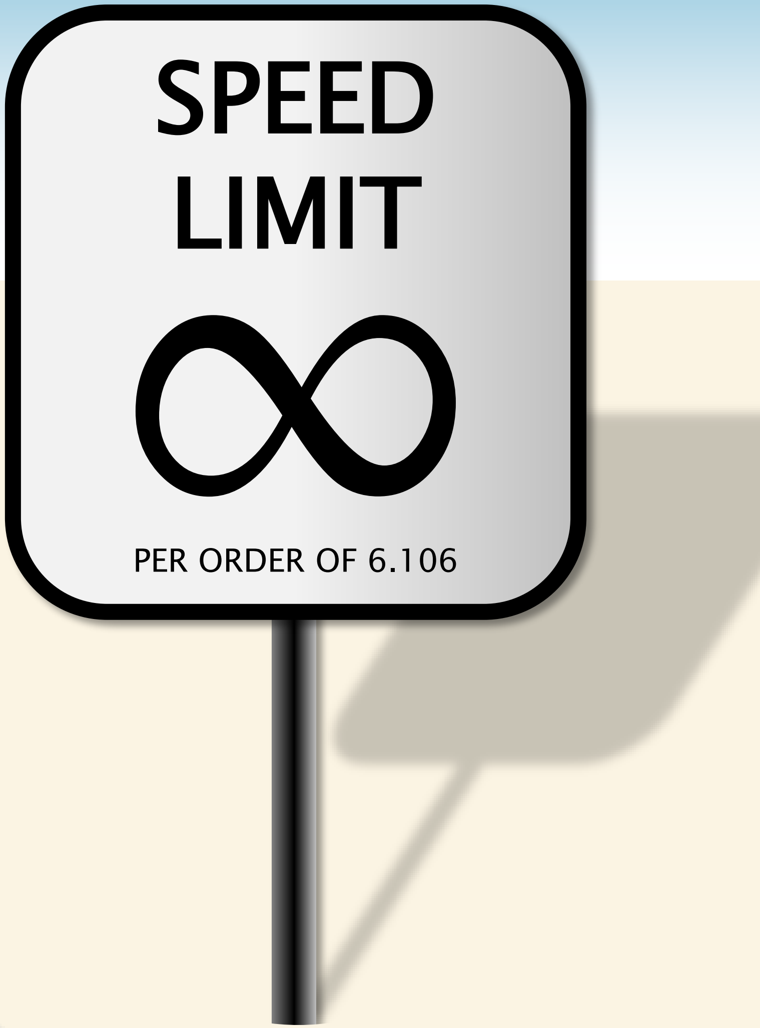
Common LLVM IR Instructions

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

LLVM IR Data Types

LLVM IR uses a simple ***type system***.

- 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



C Loops to LLVM IR

Components of a C Loop

```c
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

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

Loop body

Loop control

LLVM IR snippet

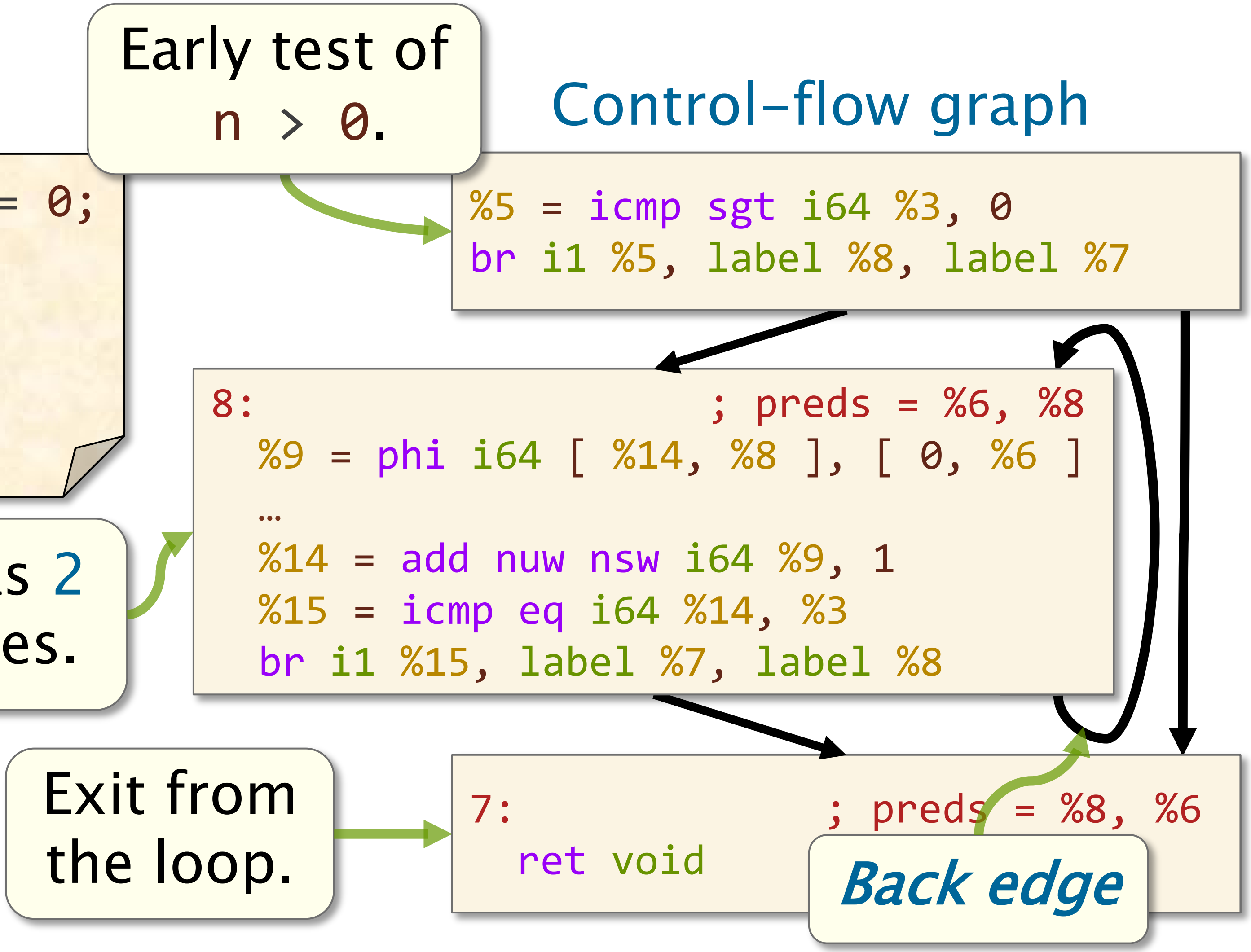
```
8:                                ; preds = %6, %8
%9 = phi i64 [ %14, %8 ], [ 0, %6 ]
%10 = getelementptr inbounds double,
      double* %2, i64 %9
%11 = load double, double* %10, align 8
%12 = fmul double %11, %1
%13 = getelementptr inbounds double,
      double* %0, i64 %9
store double %12, double* %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

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



Loop Control

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

Register %3 holds the value of n.

C code

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

Initialization

Increment

Condition

LLVM IR

```
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
```

Because of SSA, the induction variable occupies multiple registers.

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

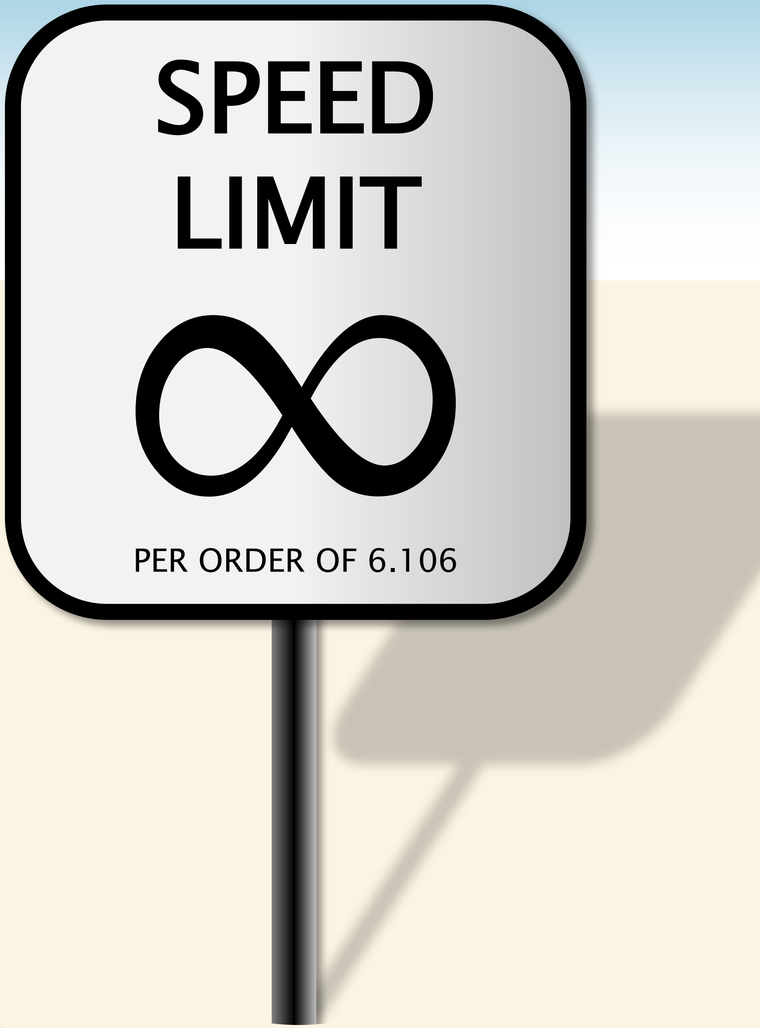
LLVM IR

```
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
```

The induction variable gets either the initial or incremented value.

The incremented value ends up in a new register.

Memory Operations



Dealing with Memory

Operations on pointers and aggregate types — arrays or structs — generally involve accessing memory.

A memory access in LLVM IR typically involves computing an address followed by reading or writing memory.

C code

```
int* A;
A[i];
```

LLVM IR

```
%3 = getelementptr inbounds i32,
     i32* %0, i64 %1
%4 = load i32, i32* %3, align 4
```

Compute an address and store it into register %5.

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.

C code

```
int* A;
A[i];
```

Example: Compute the address `i32* %0 + %1` using pointer arithmetic.

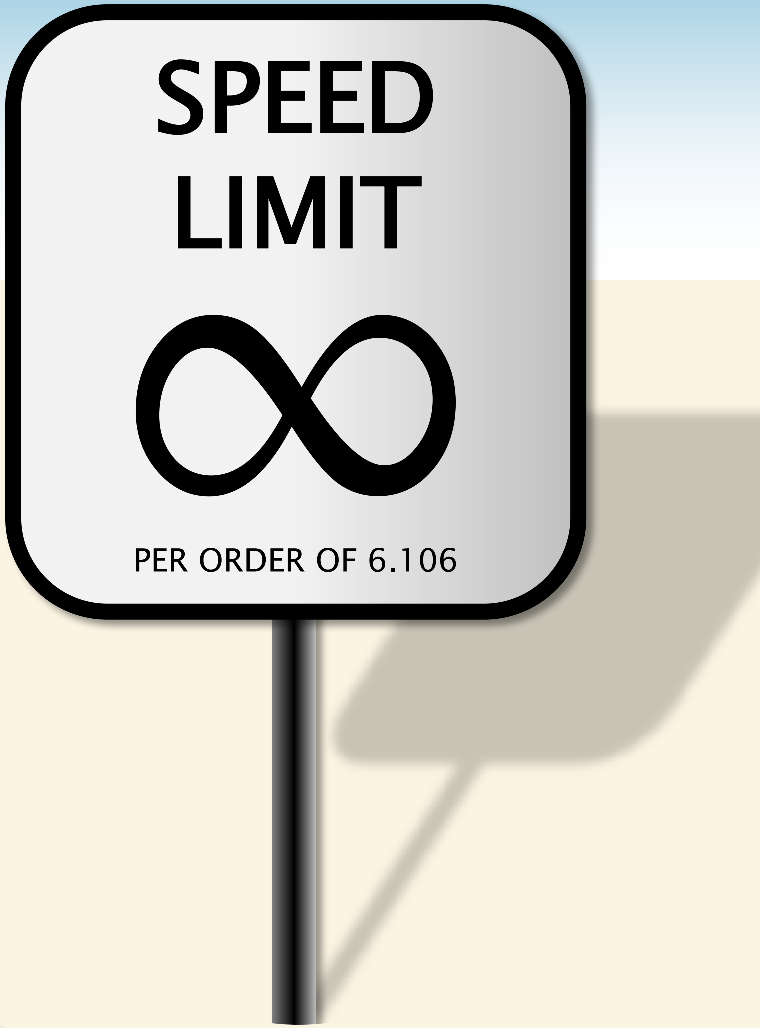
```
%3 = getelementptr inbounds i32,
     i32* %0, i64 %1
```

Pointer to the memory for A (`i32* %0`)

Index i (`i64 %1`)

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

LLVM IR Attributes



Attributes

LLVM IR constructs — including 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 daxpy.c

```
void daxpy(…
           const double *restrict x,
           …)
```

LLVM IR daxpy.ll

```
define void @daxpy(
   …
   double* noalias nocapture readonly,
   …)
```

Other attributes are determined by compiler analysis.

Analysis determined the alignment of this read.

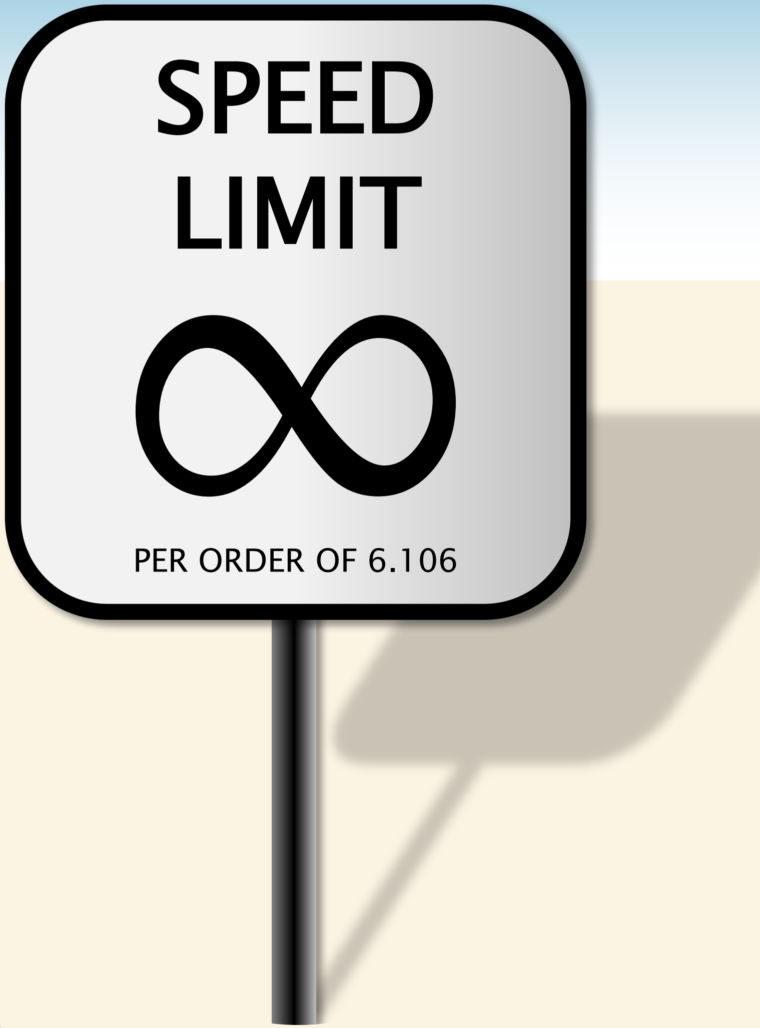
LLVM IR

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

Summary of LLVM IR

LLVM IR is similar to assembly, but simpler.

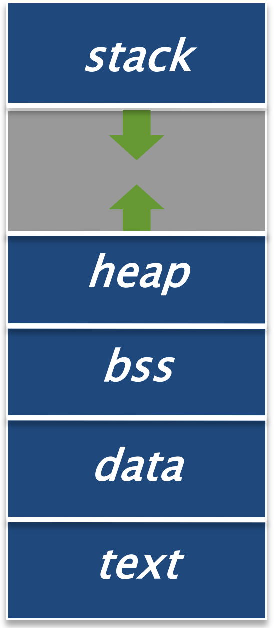
- All computed values are stored in *registers*.
- *Static single assignment:* Each register name is written on at most one line of the IR of a function.
- A function is modeled as a *control-flow graph*, whose nodes are *basic blocks*, and whose edges denote control flow between basic blocks.
- Compared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



Assembler Directives

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: .space 20`	Allocates 20 bytes at location x.
`y: .long 172`	Stores the constant 172L at location y.
`z: .asciz "6.172"`	Stores the string "6.172\0" at location z.
`.align 8`	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.