Lecture 7

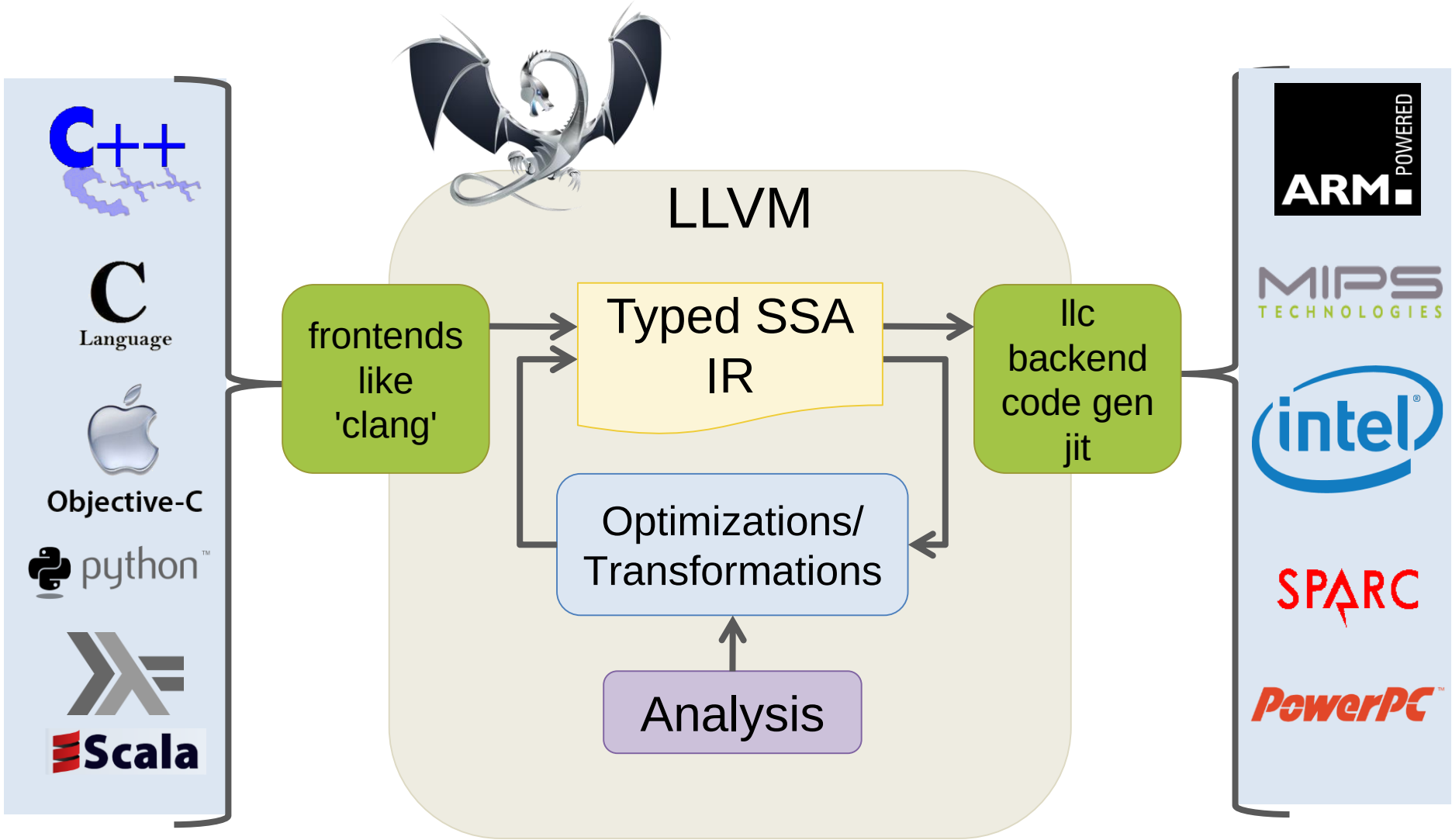
COMPILER DESIGN

Low-Level Virtual Machine LLVM

- Open-Source Compiler Infrastructure
  - see llvm.org for full documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Multi-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
  - clang: C, objective C, C++ compiler
  - various languages: Swift, ADA, Scala, Haskell, …
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects

LLVM Compiler Infrastructure

[Lattner et al.]



LLVM IR Overview

```
int s = 42;

long use(long a);

long foo(long a, long *b){
    long sum = a + 42;
    if (sum > 100) {
        use(sum);
        return sum;
    } else {
        *b = sum;
        return sum;
    }
}
```

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

Example from: 2019 LLVM Developers' Meeting: J. Paquette & F. Hahn "Getting Started With LLVM: Basics"

LLVM IR Overview

- Instructions

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return
  - Load/Store

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return
  - Load/Store
- Labeled Basic Blocks
  - The first BB label is optional

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return
  - Load/Store
- Labeled Basic Blocks
  - The first BB label is optional
  - Last instruction is called the terminator

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return
  - Load/Store
- Labeled Basic Blocks
  - The first BB label is optional
  - Last instruction is called the terminator
- Function Declarations/Definitions

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

LLVM IR Overview

- Instructions
  - Opcode
  - One or Zero SSA Return Values
  - Operands
  - Explicitly typed
- Important Instruction Classes
  - Arithmetic
  - Comparison
  - Control flow
  - Call/Return
  - Load/Store
- Labeled Basic Blocks
  - The first BB label is optional
  - Last instruction is called the terminator
- Function Declarations/Definitions
- Globals

```
@s = global i32 42

declare void @use(i64)

define i64 @foo(i64 %a, i64* %b) {
    %sum = add nsw i64 %a, 42
    %cond = icmp sgt i64 %sum, 58
    br i1 %cond, label %then, label %else

then:
    call void @use(i64 %sum)
    ret i64 %sum

else:
    store i64 %sum, i64* %b
    ret i64 %sum
}
```

Example LLVM Code

```
#include <stdint.h>

int64_t factorial(int64_t n) {
  int64_t acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

```
define i64 @factorial(i64 %0)    {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %2
  store i64 1, i64* %3
  br label %4

4:
  %5 = load i64, i64* %2
  %6 = icmp sgt i64 %5, 0
  br i1 %6, label %7, label %13

7:
  %8 = load i64, i64* %3
  %9 = load i64, i64* %2
  %10 = mul nsw i64 %8, %9
  store i64 %10, i64* %3
  %11 = load i64, i64* %2
  %12 = sub nsw i64 %11, 1
  store i64 %12, i64* %2
  br label %4

13:
  %14 = load i64, i64* %3
  ret i64 %14
}
```

Real LLVM

```
#include <stdint.h>

int64_t factorial(int64_t n) {
  int64_t acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

```
define dso_local i64 @factorial(i64 %0) #0 {
  %2 = alloca i64, align 8
  %3 = alloca i64, align 8
  store i64 %0, i64* %2, align 8
  store i64 1, i64* %3, align 8
  br label %4

4:                                                ; preds = %7, %1
  %5 = load i64, i64* %2, align 8
  %6 = icmp sgt i64 %5, 0
  br i1 %6, label %7, label %13

7:                                                ; preds = %4
  %8 = load i64, i64* %3, align 8
  %9 = load i64, i64* %2, align 8
  %10 = mul nsw i64 %8, %9
  store i64 %10, i64* %3, align 8
  %11 = load i64, i64* %2, align 8
  %12 = sub nsw i64 %11, 1
  store i64 %12, i64* %2, align 8
  br label %4

13:                                               ; preds = %4
  %14 = load i64, i64* %3, align 8
  ret i64 %14
}

attributes #0 = { noinline nounwind optnone uwtable … }
```

Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the “link”
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

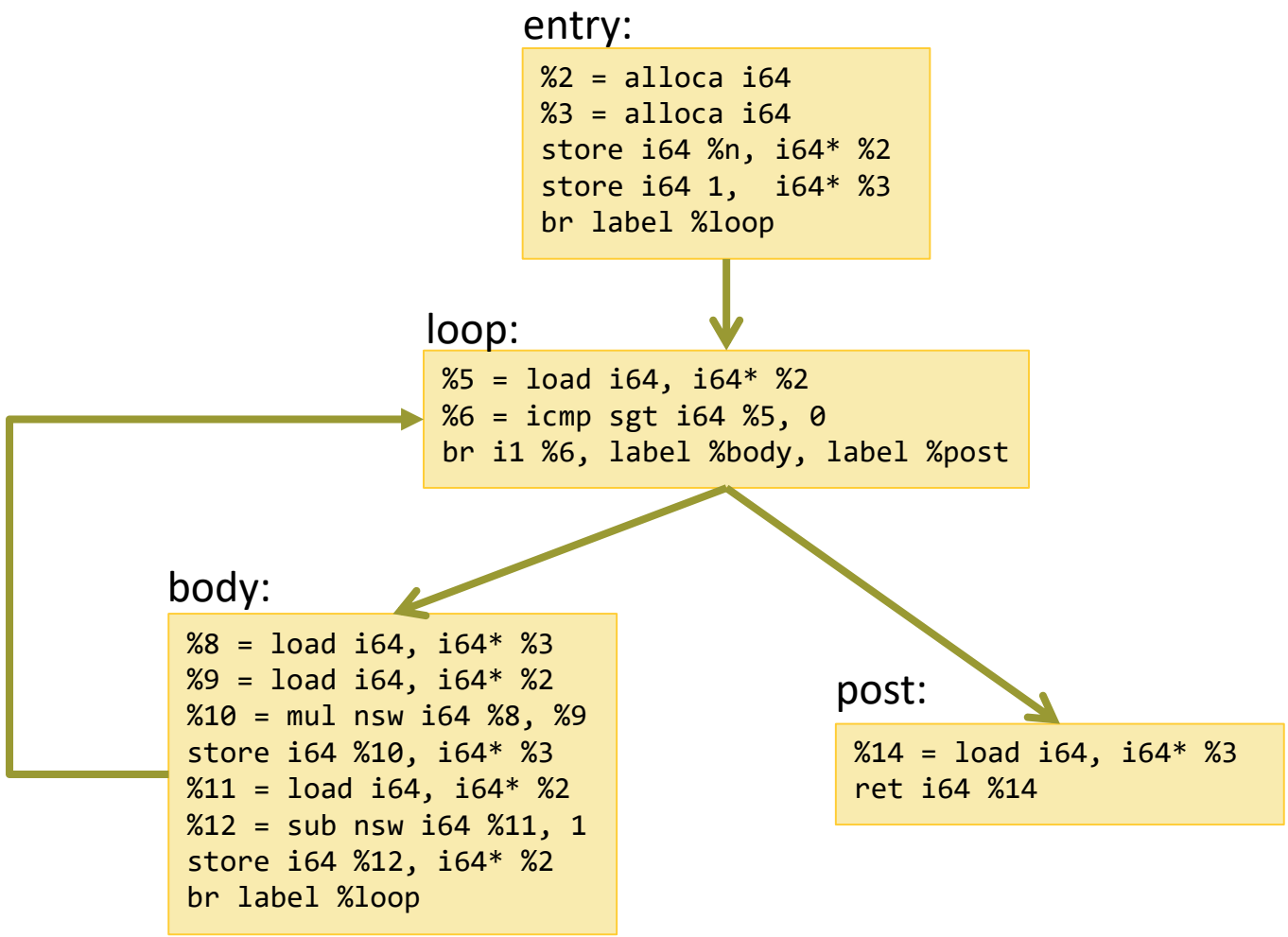
- Representation in OCaml:

```
type block = {
    insns : (uid * insn) list;
    term  : (uid * terminator)
}
```

```
define i64 @factorial(i64 %0)    {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %2
  store i64 1, i64* %3
  br label %4

4:
  %5 = load i64, i64* %2
  %6 = icmp sgt i64 %5, 0
  br i1 %6, label %7, label %13

7:
  %8 = load i64, i64* %3
  %9 = load i64, i64* %2
  %10 = mul nsw i64 %8, %9
  store i64 %10, i64* %3
  %11 = load i64, i64* %2
  %12 = sub nsw i64 %11, 1
  store i64 %12, i64* %2
  br label %4

13:
  %14 = load i64, i64* %3
  ret i64 %14
}
```

Example Control-flow Graph

```
define @factorial(%n) {
```

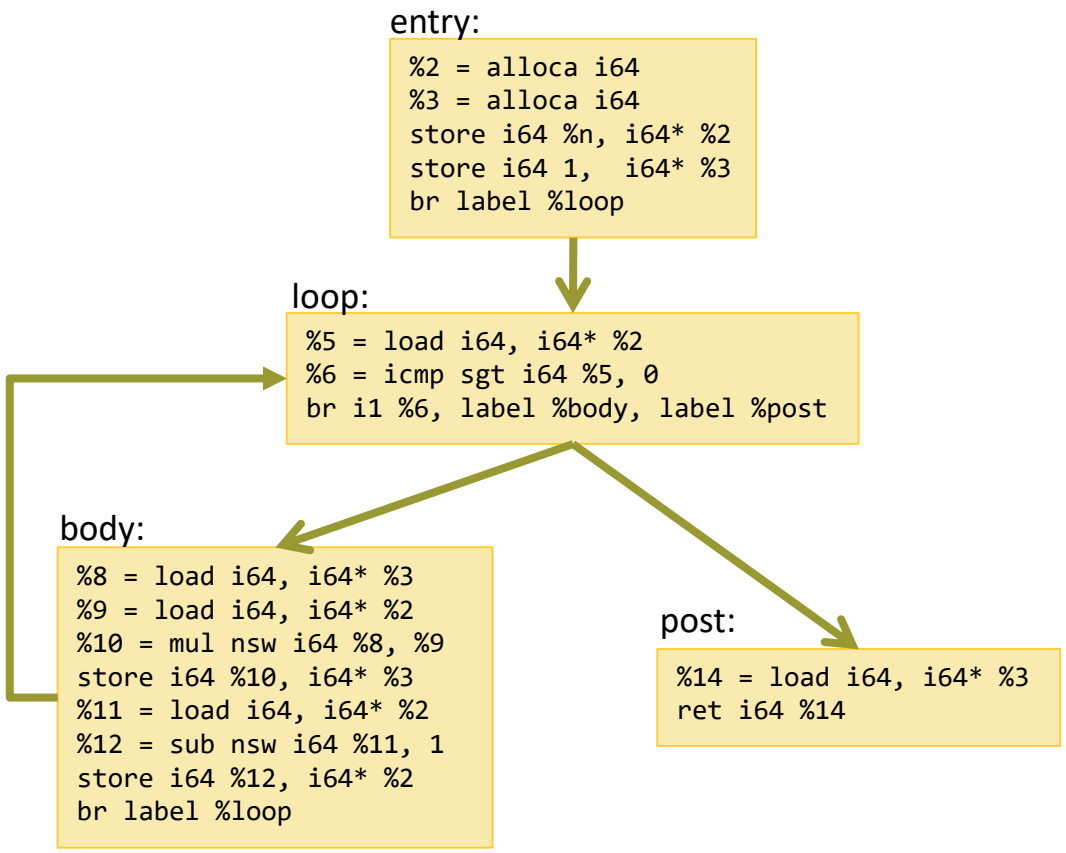



```
}
```

Control-flow Graphs

A *control flow graph* is represented as a list of labeled basic blocks with these invariants:

- No two blocks have the same label
- All terminators mention only labels that are defined among the set of basic blocks
- There is a distinguished, potentially unlabeled, entry block




```
type cfg = block * (lbl * block) list
```

LLVM IR

- Language Reference: https://llvm.org/docs/LangRef.html

- LLVMLite Spec on Moodle
  - Subset of LLVM IR
  - Used in the course (project)

COMPILING C TO LLVMLITE

(WITH CLANG'S HELP)

```
long bar(long n);

long foo(long n) {
    return bar(n);
}
```

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

Functions can be defined and/or declared.

A function type includes the return type (can be void) and the types of its arguments.

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

The `call` instruction calls a function with the provided arguments.

Calls to void functions have no return value.

A function returns via the `ret` keyword (`ret void` for void return types).

```
define i64 @foo(i64 %0) {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
ptr = alloca Type
```

- Allocates a sizeof(Type) “stack slot”.
- The slot is automatically freed when the function returns.
- The return value is a pointer to that slot and can be used to read from or write data to it.

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
store Typ %val, Typ* addr
```

Stores the value to wherever address points to.

```
%val = load Typ, Typ* %addr
```

Loads a Typ from the provided address.

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
long foo(long n) {
    long a = n + 1;
    return a * n;
}
```

```
define i64 @foo(i64 %0) {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %2
  %4 = load i64, i64* %2
  %5 = add nsw i64 %4, 1
  store i64 %5, i64* %3
  %6 = load i64, i64* %3
  %7 = load i64, i64* %2
  %8 = mul nsw i64 %6, %7
  ret i64 %8
}
```

LLVM values are stored on "registers".

- There is no limit to the number of registers.
- They are SSA
  - Defined once
  - Never changed
- They model "real" registers

```
define i64 @foo(i64 %0) {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %2
  %4 = load i64, i64* %2
  %5 = add nsw i64 %4, 1
  store i64 %5, i64* %3
  %6 = load i64, i64* %3
  %7 = load i64, i64* %2
  %8 = mul nsw i64 %6, %7
  ret i64 %8
}
```

```
long g = 5;


long* foo() {
    long* ptr = malloc(8);
    *ptr = g;
    return ptr;
}
```

```
@g =  global i64 5

define i64* @foo()  {
  %1 = alloca i64*
  %2 = call i8* @malloc(i64 8)
  %3 = bitcast i8* %2 to i64*
  store i64* %3, i64** %1
  %4 = load i64, i64* @g
  %5 = load i64*, i64** %1
  store i64 %4, i64* %5
  %6 = load i64*, i64** %1
  ret i64* %6
}
```

No direct support for dynamic memory allocation.

OS/Runtime facilities, such as malloc, must be used.

```
@g =  global i64 5

define i64* @foo()  {
  %1 = alloca i64*
  %2 = call i8* @malloc(i64 8)
  %3 = bitcast i8* %2 to i64*
  store i64* %3, i64** %1
  %4 = load i64, i64* @g
  %5 = load i64*, i64** %1
  store i64 %4, i64* %5
  %6 = load i64*, i64** %1
  ret i64* %6
}
```

Globals are “pointers”.

Can be manipulated via load/store

```
@g =  global i64 5

define i64* @foo()  {
  %1 = alloca i64*
  %2 = call i8* @malloc(i64 8)
  %3 = bitcast i8* %2 to i64*
  store i64* %3, i64** %1
  %4 = load i64, i64* @g
  %5 = load i64*, i64** %1
  store i64 %4, i64* %5
  %6 = load i64*, i64** %1
  ret i64* %6
}
```

```
struct Point {
    long x;
    long y;
};

void foo(){
    struct Point p;
    p.x = 1;
    p.y = 2;
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
          %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
          %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

```
struct Point {
    long x;
    long y;
};

void foo(){
    struct Point p;
    p.x = 1;
    p.y = 2;
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
          %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
          %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

Datatypes in LLVM

- LLVM's IR uses types to describe the structure of data.
- <#elts> is an integer constant >= 0
- Structure types can be named at the top level:
  - Such structure types can be recursive

```
t ::=
    void
    i1 | i8 | i64          N-bit integers
    [<#elts> x t]          arrays
    fty                    function types
    {t1, t2, … , tn}       structures
    t*                     pointers
    %Tident                named (identified) type

fty ::=                    Function Types
    t (t1, .., tn)         return, argument types

%T1 = type {t1, t2, … , tn}  named type
```

Example LL Types

C Code

```
struct Node {
    long a;
    struct Node* next;
};

struct List {
  struct Node head;
  long length;
};

struct ListsOfLists{
    struct List *lists1;
    struct List *lists2;
};
```

```
void foo() {
    long a[4];
    long b[3][4];
    struct Node c;
    struct List d;
    struct ListsOfLists f;
    long(*g)(long,long);
}
```

LLVM IR

```
%struct.Node = type { i64, %struct.Node* }

%struct.List = type { %struct.Node, i64 }

%struct.ListsOfLists =
      type { %struct.List*, %struct.List* }

Define void @foo() #0 {
  %1 = alloca [4 x i64]                ;a
  %2 = alloca [3 x [4 x i64]]          ;b
  %3 = alloca %struct.Node             ;c
  %4 = alloca %struct.List             ;d
  %5 = alloca %struct.ListsOfLists     ;f
  %6 = alloca i64 (i64, i64)*          ;g
  ret void
}
```

```
struct Point {
    long x;
    long y;
};

void foo(){
    struct Point p;
    p.x = 1;
    p.y = 2;
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
          %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
          %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

getelementptr

LLVM provides the `getelementptr` instruction to compute pointer values

- Given a pointer and a “path” through the structured data pointed to by that pointer, `getelementptr` computes an address
- This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
- It is a “type indexed” operation, since the size computations depend on the type

<result> = getelementptr <ty>* <ptrval>{,<ty> <idx>}*

GEP Example*

```
struct RT {
     int A;
     int B[10][20];
     int C;
}
struct ST {
     struct RT X;
     int Y;
     struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR

2. Compute the index of the 1st element by adding `size_ty(%ST)`.

3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.

4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.

5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
     %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
     ret i32* %arrayidx
}
```

Final answer: `ADDR + size_ty(%ST) + size_ty(%RT) + size_ty(i32) + size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

getelementptr

- GEP *never* dereferences the address it's calculating:
  - GEP only produces pointers by doing arithmetic
  - It doesn't actually traverse the links of a datastructure

- To index into a deeply nested structure, need to "follow the pointer" by loading from the computed pointer

```
struct Point {
    long x;
    long y;
};

void foo(struct Point *ps,
                        long n){
    ps[n].y = 42;
}
```

```
%struct.Point = type { i64, i64 }


define  void @foo(%struct.Point*  %0, i64 %1)
{
  %3 = getelementptr,
            %struct.Point* %0, i64 %1, i32 1
  store i64 42, i64* %3
  ret void
}
```

Arrays are indexed with getelementptr

```
%struct.Point = type { i64, i64 }


define  void @foo(%struct.Point*  %0, i64 %1)
{
  %3 = getelementptr,
            %struct.Point* %0, i64 %1, i32 1
  store i64 42, i64* %3
  ret void
}
```

```
struct Point {
    long x;
    long y;
};


long foo(struct Point p){
    return p.x + p.y;
}
```

```
%struct.Point = type { i64, i64 }


define i64 @foo(i64 %0, i64 %1)   {
  %3 = add nsw i64 %1, %0
  ret i64 %3
}
```

Struct arguments are unpacked.

- Calling convention depended.

```
%struct.Point = type { i64, i64 }


define i64 @foo(i64 %0, i64 %1)   {
  %3 = add nsw i64 %1, %0
  ret i64 %3
}
```

```
struct Point {
    long x;
    long y;
    long z;
};


struct Point foo(long x, long y, long z){
    struct Point p;
    p.x = x;
    p.y = y;
    p.z = z;
    return p;
}
```

```
%struct.Point = type { i64, i64, i64 }

define void @foo(%struct.Point*  %0,
                 i64 %1, i64 %2, i64 %3)   {
  %5 = getelementptr,
               %struct.Point* %0, i64 0, i32 0
  store i64 %1, i64* %5, align 8
  %6 = getelementptr,
               %struct.Point* %0, i64 0, i32 1
  store i64 %2, i64* %6, align 8
  %7 = getelementptr,
               %struct.Point* %0, i64 0, i32 2
  store i64 %3, i64* %7, align 8
  ret void
}
```

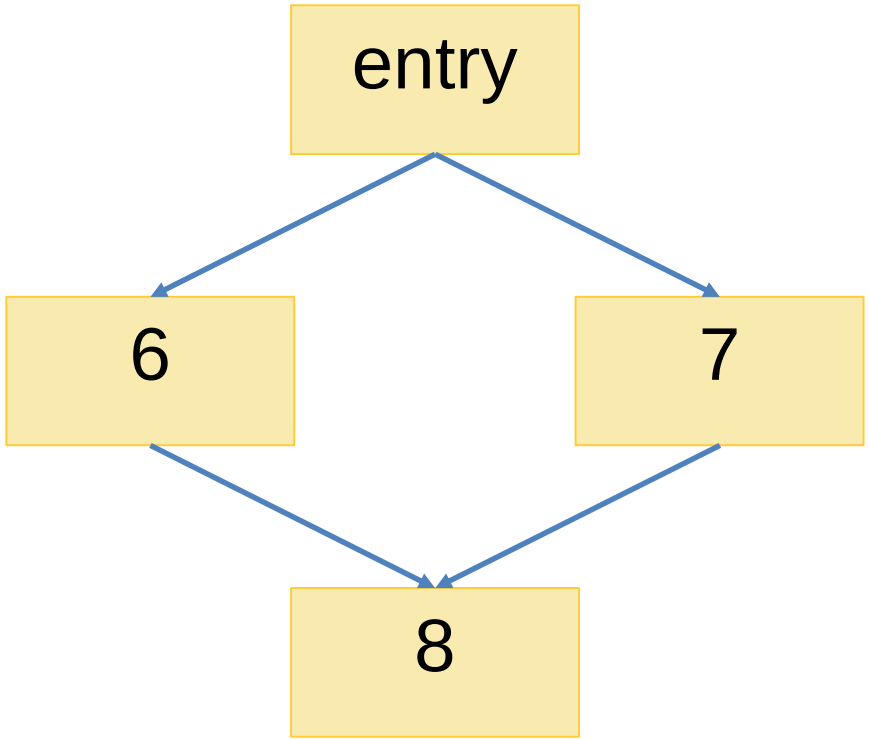
Return struct passed as pointer argument.

- Allocated by the caller
- Calling convention dependent

```
%struct.Point = type { i64, i64, i64 }

define void @foo(%struct.Point*  %0,
                 i64 %1, i64 %2, i64 %3)   {
  %5 = getelementptr,
               %struct.Point* %0, i64 0, i32 0
  store i64 %1, i64* %5, align 8
  %6 = getelementptr,
               %struct.Point* %0, i64 0, i32 1
  store i64 %2, i64* %6, align 8
  %7 = getelementptr,
               %struct.Point* %0, i64 0, i32 2
  store i64 %3, i64* %7, align 8
  ret void
}
```

```
long sign(long n) {
    long s;
    if ( n < 0)
        s =  - 1;
    else
        s =  1;
    return s;
}
```

```
define  i64 @sign(i64 %0)  {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %3
  %4 = load i64, i64* %3
  %5 = icmp slt i64 %4, 0
  br i1 %5, label %6, label %7
6:
  store i64 -1, i64* %2
  br label %8
7:
  store i64 1, i64* %2
  br label %8
8:
  %9 = load i64, i64* %2
  ret i64 %9
}
```

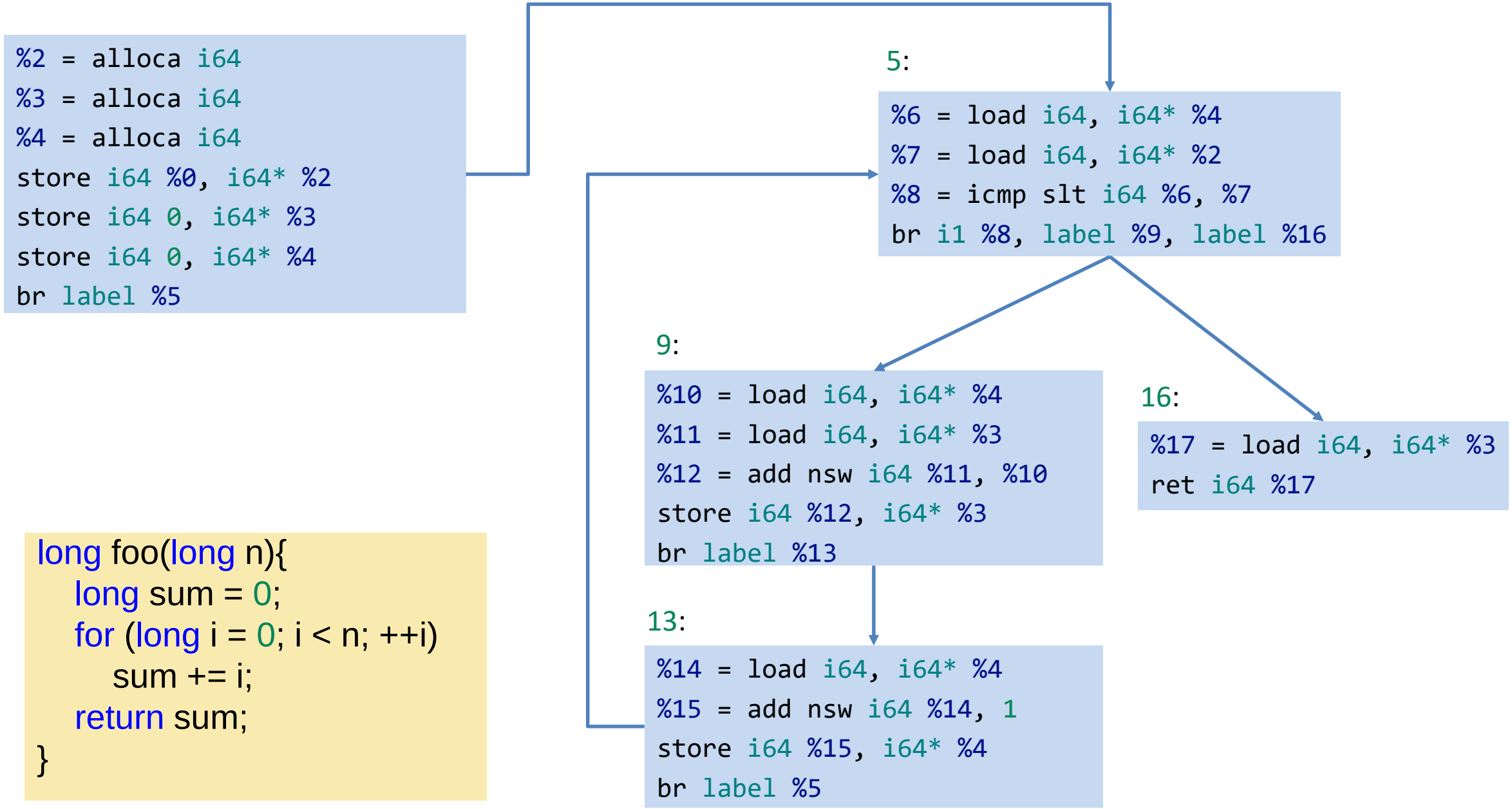


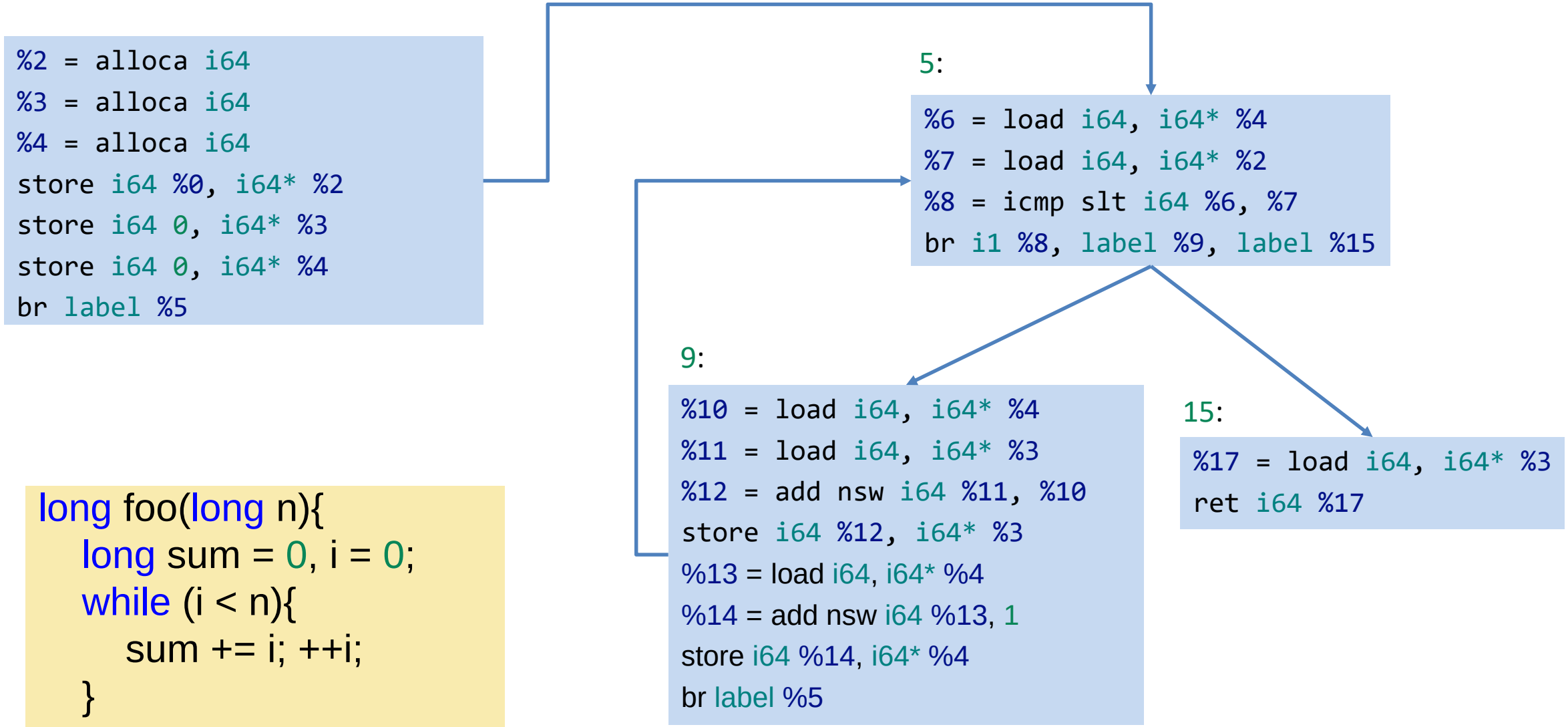

```
define  i64 @sign(i64 %0)  {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %3
  %4 = load i64, i64* %3
  %5 = icmp slt i64 %4, 0
  br i1 %5, label %6, label %7
6:
  store i64 -1, i64* %2
  br label %8
7:
  store i64 1, i64* %2
  br label %8
8:
  %9 = load i64, i64* %2
  ret i64 %9
}
```

For each branch of an if-statement:

- Generate a separate basic block
  - Additional basic blocks might be necessary for nested control flow.
- Connect the basic blocks with (conditional) branches
- Stores/loads can be used for assignments

```
define  i64 @sign(i64 %0)  {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %3
  %4 = load i64, i64* %3
  %5 = icmp slt i64 %4, 0
  br i1 %5, label %6, label %7
6:
  store i64 -1, i64* %2
  br label %8
7:
  store i64 1, i64* %2
  br label %8
8:
  %9 = load i64, i64* %2
  ret i64 %9
}
```


%2 = alloca i64
%3 = alloca i64
%4 = alloca i64
store i64 %0, i64* %2
store i64 0, i64* %3
store i64 0, i64* %4
br label %5
5:
%6 = load i64, i64* %4
%7 = load i64, i64* %2
%8 = icmp slt i64 %6, %7
br i1 %8, label %9, label %16
9:
%10 = load i64, i64* %4
%11 = load i64, i64* %3
%12 = add nsw i64 %11, %10
store i64 %12, i64* %3
br label %13
13:
%14 = load i64, i64* %4
%15 = add nsw i64 %14, 1
store i64 %15, i64* %4
br label %5
16:
%17 = load i64, i64* %3
ret i64 %17
long foo(long n){
long sum = 0;
for (long i = 0; i < n; ++i)
sum += i;
return sum;
}

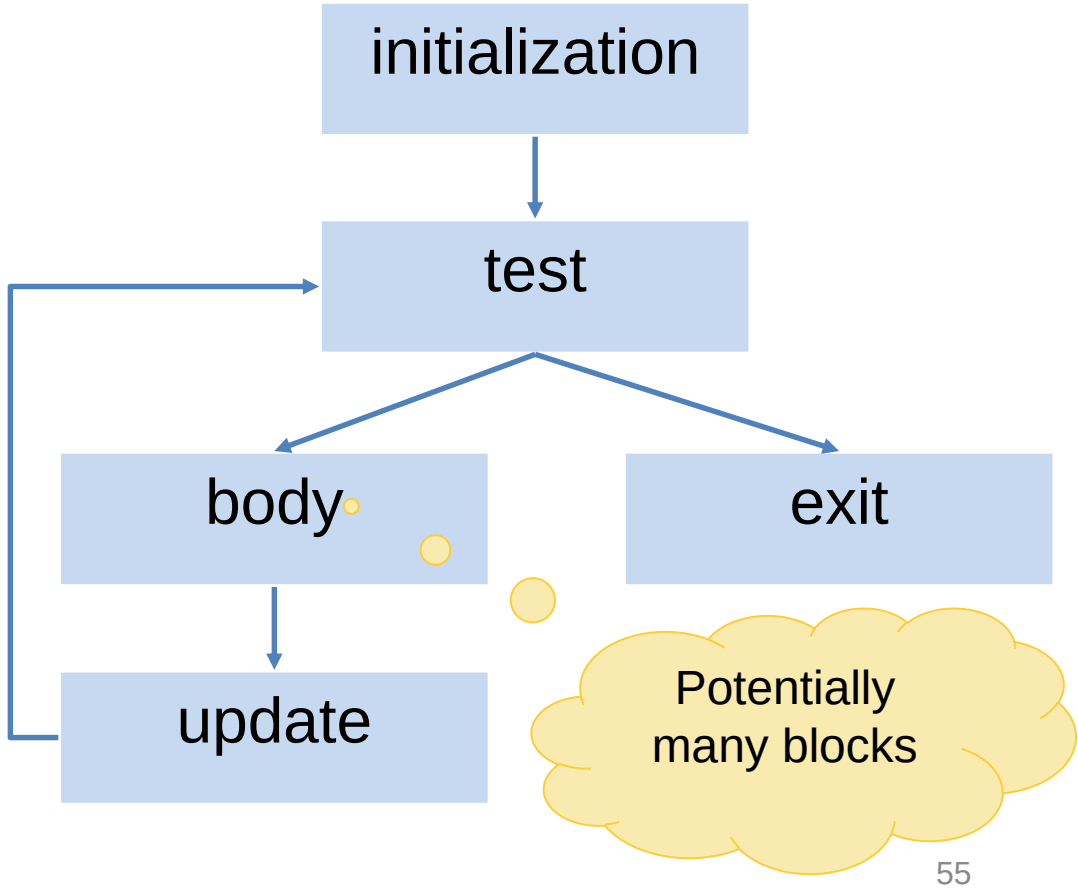



```
long foo(long n){
    long sum = 0, i = 0;
    while (i < n){
        sum += i; ++i;
    }
    return sum;
}
```

Generating Code for Loops

```
for (initializationStatement; testExpression; updateStatement) {
    // statements inside the body of loop
}
```

- Generate a basic block with the initialization
- Generate a basic block with the test expression
- Generate a basic block for the update statement
- Generate (a) basic block(s) for the body of the loop
- Connect them with (conditional branches)



LLVM Cheat Sheet

// Extract LLVM-IR from C code – with optimizations

clang –S –emit-llvm –O3 –o file.ll file.c

// Extract LLVM-IR from C code – no optimization

clang –S –emit-llvm –O0 –o file.ll cile.c -Xclang –disable-llvm-passes

// View the CFG of a file

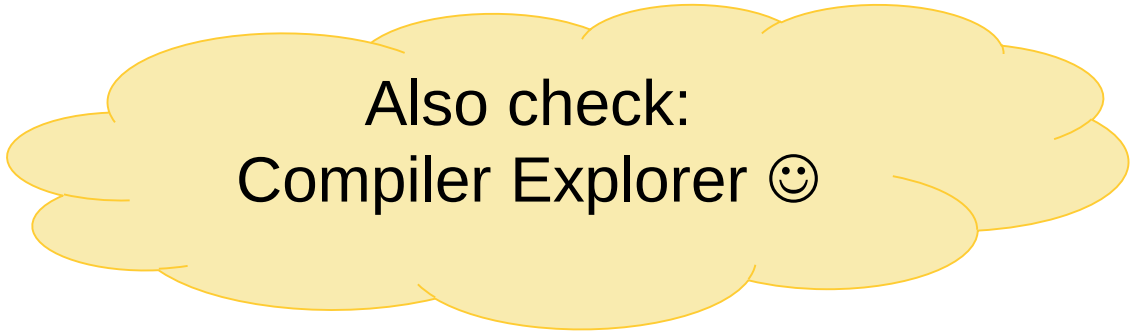
opt –view-cfg file.ll

// Compile .ll file to .o file

clang file.ll –c –o file.o

// Compile .ll file to executeable

clang file.ll –o file.exe



DEMO