LLVM Tutorial

Gregory J. Duck

Overview

- Basic LLVM tutorial
  - LLVM architecture overview (frontend, optimizer, linker)
  - LLVM *Intermediate Representation* (IR) overview
  - How go generate the IR
  - Programming with LLVM IR

- Basic LLVM knowledge is required for Assignment 2

LLVM Compiler System

- LLVM = ~~*Low Level Virtual Machine*~~ LLVM = LLVM

- The LLVM *Compiler Infrastructure Project*
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...

- The LLVM *Compiler Framework*
  - End-to-end compilers using the LLVM infrastructure
  - C and C++ are robust
  - Emit C code or native code for X86, Sparc, PowerPC, etc.

Three primary LLVM components

- The LLVM *Virtual Instruction Set* (Main focus of today's tutorial)
  - The common language- and target-independent *Intermediate Representation* (IR)
- A collection of *integrated libraries*
  - Analyses, optimizations, code generators, JIT compiler, profiling, etc.
- A collection of *tools* built from the libraries
  - Assemblers, debugger, linker, code generator, compiler driver, optimizer, etc.

The LLVM C/C++ Compiler

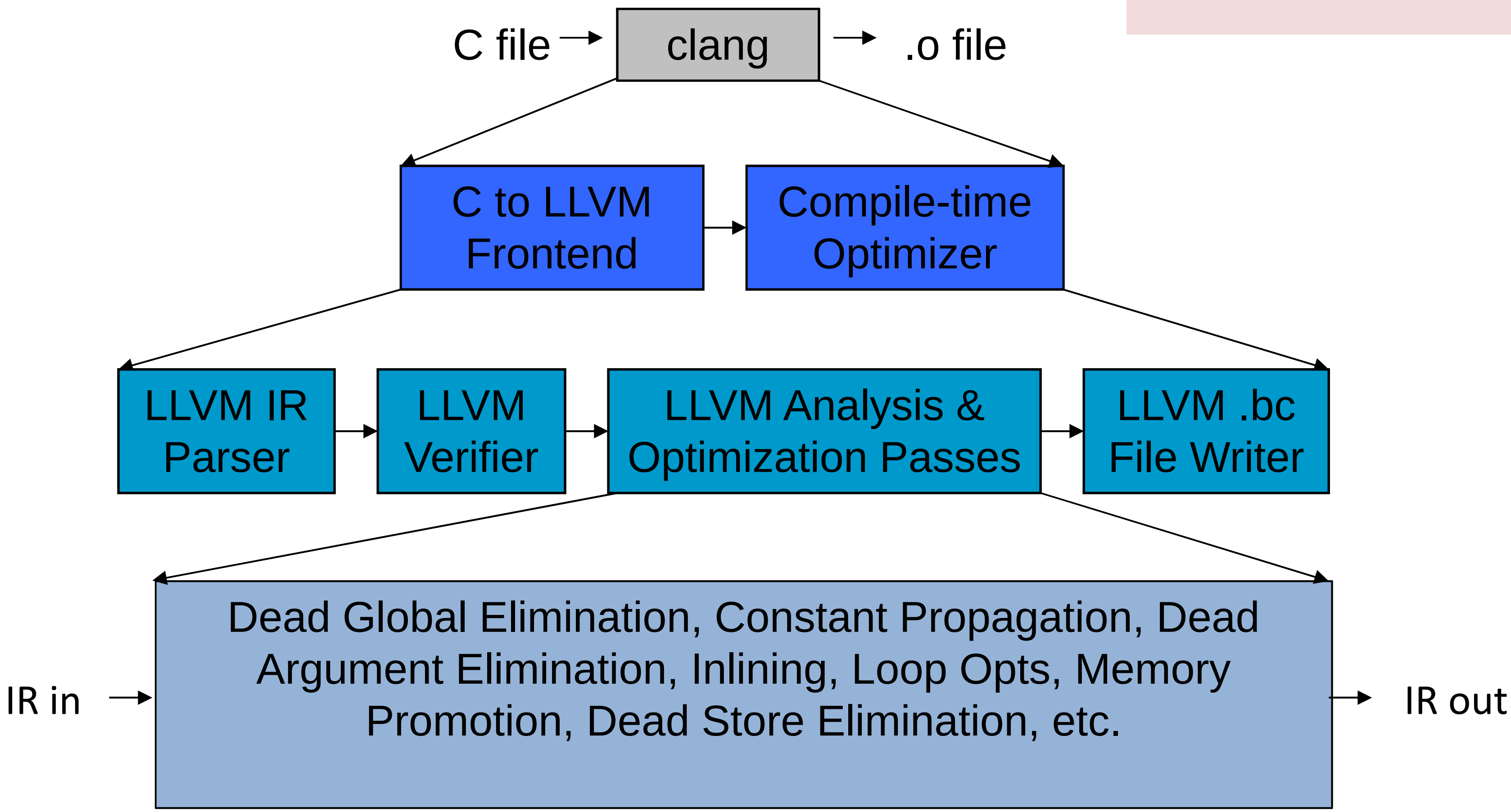
- LLVM C/C++ is a standard compiler:
  - Compatible with GCC options and extensions
  - Makefile compatible

```
$ clang -c hello.c
$ clang -o hello hello.o
$ ./hello
Hello World
```

- Standard compiler organization, which uses LLVM as midlevel IR:
  - Language specific front-end lowers code to LLVM IR
  - Language/target independent optimizers improve code
  - Target specific back-end (e.g. x86) generate native code

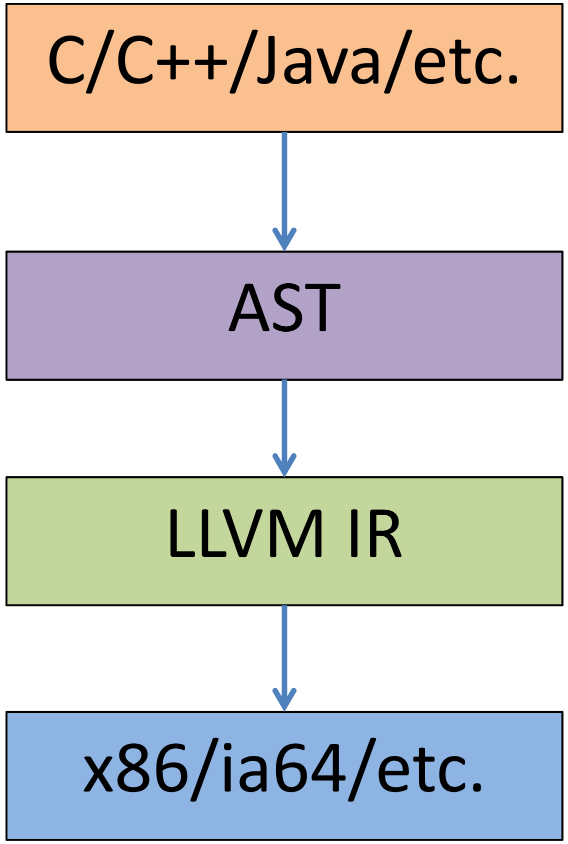
LLVM Compilation Overview

clang -c hello.c



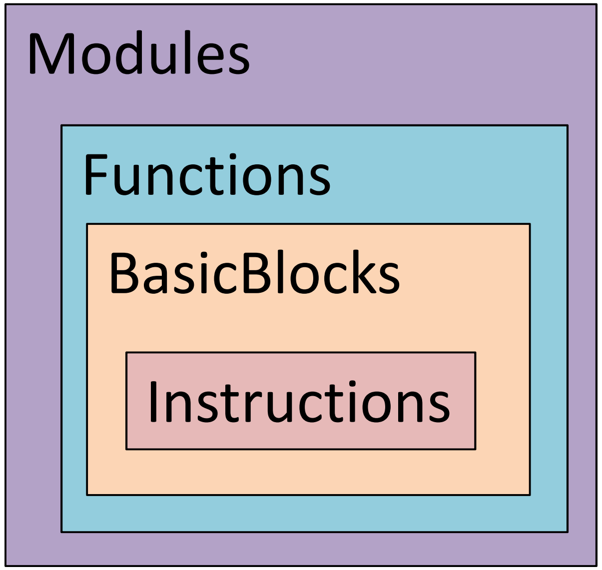
Goals of LLVM IR

- Language- and Target-Independent
  - AST-level IR is not language independent
- One IR for analysis **and** optimization
  - IR must be able to support aggressive IPO, loop opts, scalar opts, … high- ***and*** low-level optimization!
- Optimize early as possible
  - No lowering in the IR!



LLVM IR Program Structure

- Module contains Functions/GVs
  - Module is unit of compilation
- Function contains BasicBlocks
  - Functions correspond to C functions
- BasicBlock contains instructions
  - Each block ends in a control flow instruction
- Instruction is opcode + operands



My slides go “bottom up”

DEMO: Generating LLVM IR

LLVM IR Instruction Basics

- LLVM IR Instructions are of the general form:

  retval = opcode type operand_1, . . ., operand_n

  (There can be other forms!)

- For example, addition:

```
%z = add i32 %x, %y
```

Registers (output) | Opcode | Type | Registers (input)

- C/ASM Equiv.:

```
int x, y, z; . . .; z = x + y;
```

```
add %rbx, %rax
```

LLVM IR Instruction Components

```
%z = add i32 %x, %y
```

- Registers:
  - Register names begin with a `%’, e.g. %x, %y, %z, etc.
  - Registers are *unlimited* (unlike real machines, e.g. x86).
- Types:
  - Integer types, e.g. i32 = 32bit integer.
  - Other types: float (floating point), i32* (pointers), %struct.node (structures), [100 x i32] (arrays), etc.

LLVM IR Instruction Components

```
%z = add i32 %x, %y
```

- Opcodes (various):
  - *Arithmetic*: add, sub, mul, div, etc. *Bitwise*: lshr, asll, and, or, xor, etc. *Comparison*: cmp *Control*: ret, call, br, switch, etc. *Casts*: bitcast, zext, sext, trunc, ptrtoint, inttoptr, etc. *Memory*: alloca, load, store, getelementptr
- Return value is sometimes optional.

```
ret i32 %x
```

```
call @func(i32 %x, i32 %y)
```

- Some opcodes have special syntax.

```
%z = bitcast i32* %y to i16*
```

LLVM IR Instruction Components

- Instruction operands can also be *constants*:

```
%z = add i32 %x, 1
```

z = x + 1

- Also *global variables*:

z = (short *)inc ; // Global

```
%z = bitcast i32* @inc to i16*
```

  – Local (registers) start with a %
  – Globals start with a @, e.g. @inc
  – Note that functions are typically globals, e.g. @main

LLVM IR Key Properties

- Low-level and target-independent semantics
  - RISC-like three address code / SSA form
  - Infinite virtual register set
  - Strongly typed
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointer

Properties: LLVM IR is in SSA Form

- LLVM is in *Single Static Assignment* form:
  - Values are created once and can never change.
  - Motivation: simplifies analysis and optimization.
  - LLVM has infinite registers, so not a problem.
  - E.g. this instruction is **not** in SSA form (%x redefined):

```
%x = add i32 %x, %y
ret i32 %x
```

Error

```
%1 = add i32 %x, %y
ret i32 %1
```

OK!

(SSA is standard practice)

(Register names can be numbers, e.g. %1, %2, etc.)

Properties: LLVM IR is Strongly Typed

- LLVM is *strongly typed*:

```
%z = add i32 %x, %y
%w = add i64 %z, %z
```

Type error

(%z is int32 and passed to int64 instruction)

- Types must be explicitly converted via instructions:

```
%y = zext i32 %x to i64
```

int32 to int64 (zero extend)

```
%y = trunc i64 %x to i32
```

int64 to int32 (truncate)

```
%y = bitcast i32* %x to i64*
```

int32* to int64*

(others for int2ptr, ptr2int, int2float, etc., etc.)

LLVM Instruction Types

- LLVM has many instruction types:
  - Arithmetic & Bitwise
  - Casts
  - Memory
  - Function calls
  - Pointer arithmetic
  - Comparison and Control Flow
  - PHI-nodes

- All C/C++ compile to these basic instructions.

LLVM Instruction Types: Memory

- Memory is accessed via two instructions:

```
%x = load i32 %y
```

x = *y

```
store i32 %y, i32* %x
```

*x = y

  – All other instructions must work on registers (unlike x86)

- Stack memory allocated via *alloca*

```
%x = alloca [100xi32]
```

int x[100]

LLVM Instruction Types: GEP

- GEP = *Get Element Pointer*
- GEP is for *pointer arithmetic*

y = x + 10

y = x + 10

```
%y = getelementptr i32* %x, i32 10
```

y = &x->next   a.k.a.   y = &x[0].next

y = x + 0 + offsetof(next)

```
%y = getelementptr %struct.node* %x, 0, 1
```

- The often misunderstood GEP instruction
  - http://llvm.org/docs/GetElementPtr.html
  - GEP = pointer arithmetic, nothing else.

LLVM Basic Blocks & Control Flow

- Instructions are arranged into *Basic Blocks*:

```
return x * y + z;
```

```
label:
%1 = mul i32 %x, %y
%2 = add i32 %1, %z
ret i32 %2
```

- Each Basic Block:
  - Has a label (e.g. %label)
  - A sequence of *branch free* instructions.
  - Ends with a *terminator* instruction (e.g. ret)
  - (LLVM) Has exactly *one* entry and *one* exit (no "fall through" e.g. x86).
- Together forms the *Control Flow Graph* (CFG)

LLVM Instruction Types: Control Flow

- LLVM *branch instructions* connect Basic Blocks.

```
br i1 %x, label %true, label %false
```

Condition Var (Boolean) True branch False branch

```
br label %label
```

(unconditional branch)

- Return instruction:

```
ret i32 %x
```

- Others: invoke, unreachable

LLVM Instruction Types: Comparison

- Conditional branches take a Boolean input:

```
br i1 %b, label %true, label %false
```

- Typically the results of a *comparison instruction*:

```
%b = icmp ult i32 %x, %y
br i1 %b, label %true, label %false
```

- Multiple comparison operations supported: eq, ne, gt, ge, lt, le (both *signed* and *unsigned*)

LLVM Control Flow Example 1

```
if (x + 1 < y)
   return x * y + z;
return 0;
```

```
entry:
%1 = add i32 %x, 1
%2 = cmp lt i32 %x, %y
br i1 %2 label %T, label %F
```

true

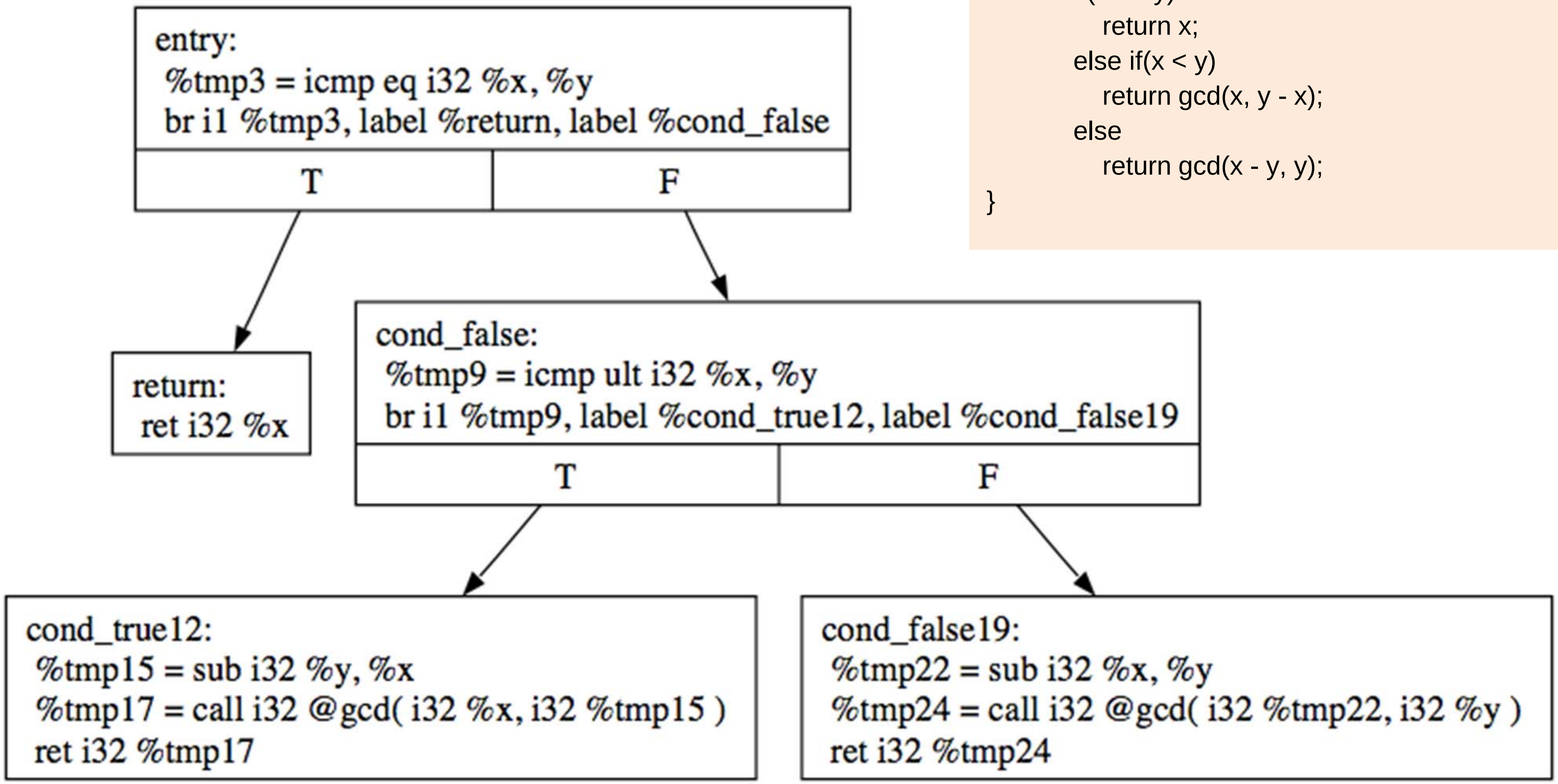
false

```
T:
%3 = mul i32 %x, %y
%4 = add i32 %3, %z
ret i32 %2
```

```
F:
ret i32 0
```

LLVM Control Flow Example 2

```
unsigned gcd(unsigned x, unsigned y)
{
        if(x == y)
            return x;
        else if(x < y)
            return gcd(x, y - x);
        else
            return gcd(x - y, y);
}
```

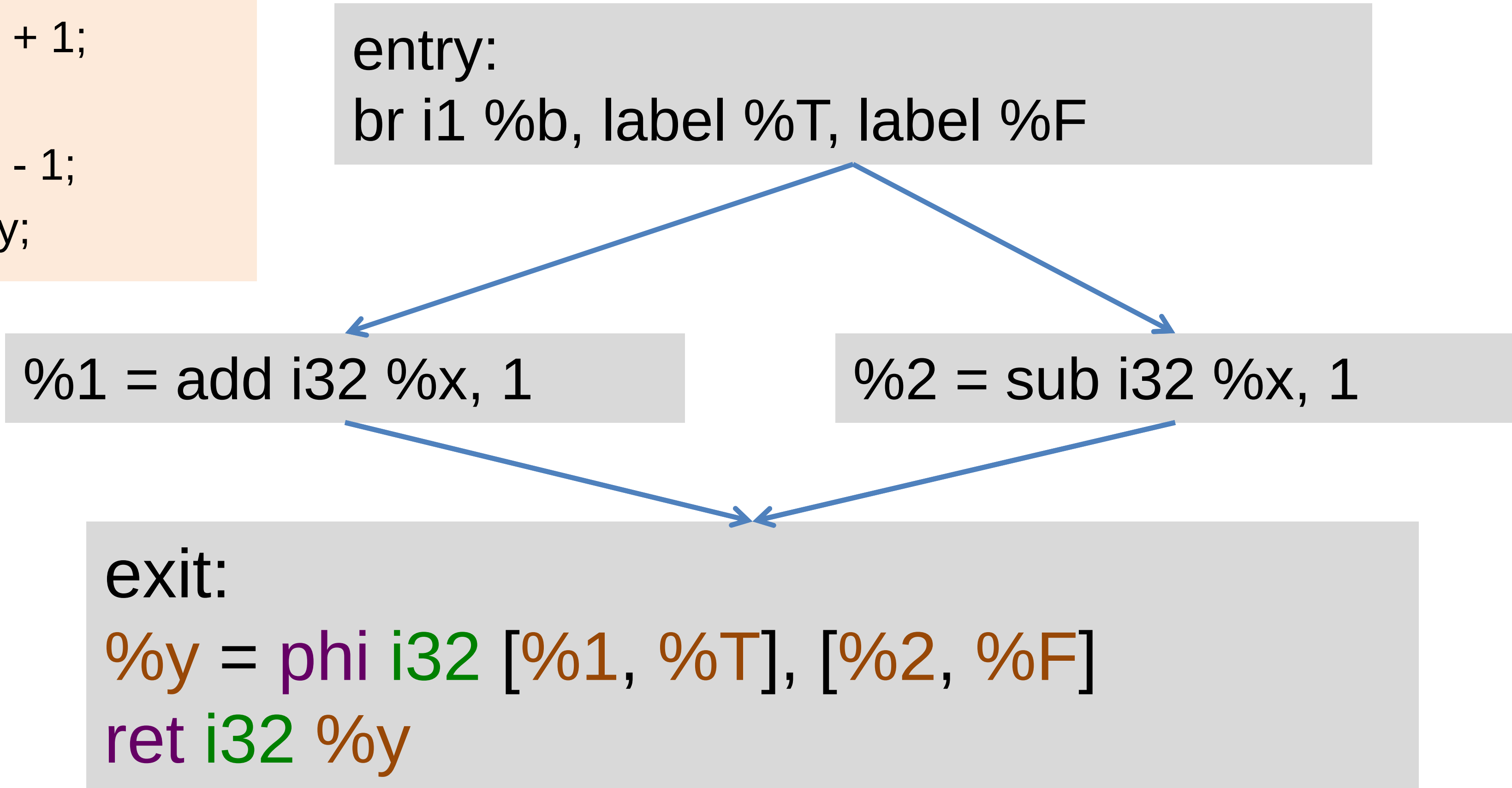


CFG for 'gcd' function

LLVM IR Instructions: PHI Nodes

- PHI Nodes are special instructions that “join” values:

```
if (b)
   y = x + 1;
else
   y = x - 1;
return y;
```



LLVM Functions

- LLVM functions roughly correspond to C functions:

```
int mul_add(int x,
            int y,
            int z)
{
    return x * y + z;
}
```

```
define i32 @mul_add(i32 %x,
                    i32 %y,
                    i32 %z)
{
  entry:
  %1 = mul i32 %x, %y
  %2 = add i32 %1, %z
  ret i32 %2
}
```

  - Each LLVM function has exactly one entry point

Part 2: Programming with LLVM IR

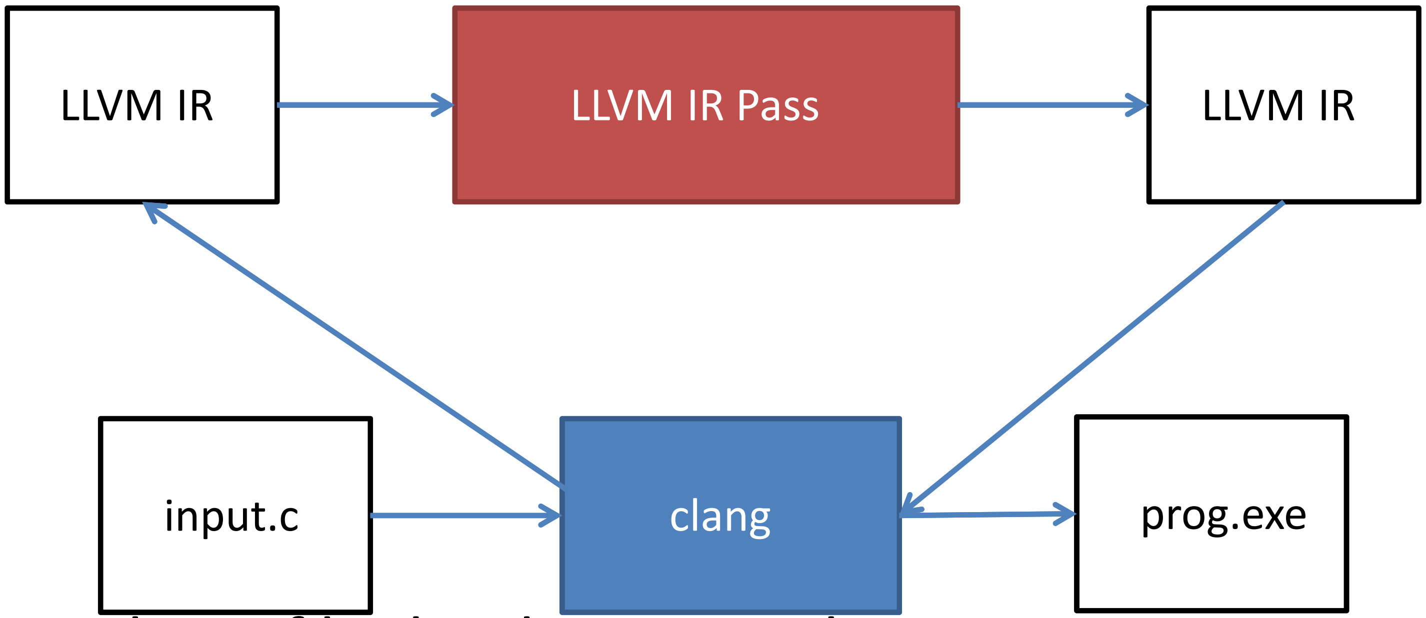
- LLVM is implemented in C++. (optional for assignment 2)
- To install:

```
sudo apt-get install llvm-dev
```

  - Installs the LLVM API (or the IR) header files and libraries.
- The LLVM API can be used to:
  - Create new LLVM compiler passes (e.g. optimizations, analysis).
  - Create standalone programs that manipulate LLVM IR.

Option 1: Compiler Pass

- A compiler pass transforms LLVM IR:



- Con: lots of boilerplate to implement.

Option 2: Standalone C++

- An alternative to create a standalone C++ program:

```
#include "llvm/IRReader/IRReader.h"
...
int main(int argc, char **argv)
{
        // Step (1) Parse the given IR File
        LLVMContext &Context = getGlobalContext();
        SMDiagnostic Err;
        Module *M = ParseIRFile(argv[1], Err, Context);
        if (M == nullptr)
        {
                fprintf(stderr, "failed to read IR file %s\n", argv[1]);
                return 1;
        }
        ...
}
```

Standalone C++ (cont.)

- Reading in an IR file generates a *Module* object.
- Instructions can be traversed using C++ range loops:

```
int main(int argc, char **argv)
{
        ...
        // Step (2) Traverse all instructions
        for (auto &F: *M)                       // For each function F
                for (auto &BB: F)               // For each basic block BB
                        for (auto &I: BB)       // For each instruction I
                                I.dump();       // Dump the instruction!
        ...
}
```

Compiling the Standalone

- Use the magic command to compile:

```
$ clang++-3.4 -o Test Test.cpp `llvm-config-3.4 --cxxflags`
`llvm-config-3.4 --ldflags` `llvm-config-3.4 --libs` -lpthread
-lncurses -ldl
```

- Run the command to dump the instructions:

```
$ Test mul_add.ll
 %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 %y, i32* %2, align 4
 ...
```

Making it more Interesting

- The LLVM API makes heavy use of dynamic casting.
- E.g. to dump only call instruction names:

```
int main(int argc, char **argv)
{
	...
	// Step (2) Traverse all instructions
	for (auto &F: *M)			// For each function F
		for (auto &BB: F)		// For each basic block BB
			for (auto &I: BB) // For each instruction I
	{
		CallInst *Call = dyn_cast<CallInst>(&I);
		if (Call == nullptr) continue;
		Function *G = Call->getCalledFunction();
		if (G == nullptr)  continue;
		printf("Name = %s\n", G->getName().str().c_str());
	}
}
```

Standalone Version 2

- Use the magic command to re-compile:

```
$ clang++-3.4 -o Test Test.cpp `llvm-config-3.4 --cxxflags` `llvm-config-3.4 --ldflags` `llvm-config-3.4 --libs` -lpthread -lncurses -ldl
```

- Run the command to dump the instructions:

```
$ Test sha256.ll
llvm.memcpy.p0i8.p0i8.i64
sha256_init
sha256_update
sha256_final
printf
...
```

Questions?

Appendix: 2 Samples

```
#include <cstdio>
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IRReader/IRReader.h"
#include "llvm/Support/SourceMgr.h”
using namespace llvm;
int main(int argc, char **argv)
{
   // Read the IR file.
   LLVMContext &Context = getGlobalContext();
   SMDiagnostic Err;
   Module *M = ParseIRFile(argv[1], Err, Context);
   if (M == nullptr)
   {
      fprintf(stderr, "error: failed to load LLVM IR file \"%s\"", argv[1]);
      return EXIT_FAILURE;
   }

   // Dump all instructions.
   for (auto &F: *M)
      for (auto &BB: F)
         for (auto &I: BB)
            I.dump();
   return 0;
}
```

```
... // add include files

int main(int argc, char **argv)
{
   // Read the IR file.
   LLVMContext &Context = getGlobalContext();
   SMDiagnostic Err;
   Module *M = ParseIRFile(argv[1], Err, Context);
   if (M == nullptr)
   {
      fprintf(stderr, "error: failed to load LLVM IR file \"%s\"", argv[1]);
      return EXIT_FAILURE;
   }
   // Dump all instructions.
   for (auto &F: *M)
      for (auto &BB: F)
         for (auto &I: BB)
   {
      CallInst *Call = dyn_cast<CallInst>(&I);
      if (Call == nullptr)
         continue;
      Function *G = Call->getCalledFunction();
      if (G == nullptr)
         continue;
      printf("%s\n", G->getName().str().c_str());
   }
}
```