Meliora!

LLVM Tutorial

John Criswell
University of Rochester

Introduction

History of LLVM

- Developed by Chris Lattner and Vikram Adve at the University of Illinois at Urbana-Champaign (UIUC)
- Released open-source in October 2003
- Default compiler for Mac OS X, iOS, and FreeBSD
- Used by many companies and research groups
- Contributions by *many* people!

LLVM is a *compiler infrastructure*!

Tools Built Using LLVM

Tools Built Using LLVM

Compilers!

Tools Built Using LLVM

Compilers!

JITs!

Tools Built Using LLVM

Compilers!

JITs!

Formal Verification!

Tools Built Using LLVM

Compilers!

JITs!

Security Hardening Tools!

Formal Verification!

Tools Built Using LLVM

Compilers!

JITs!

Security Hardening Tools!

Formal Verification!

Bug Finding Tools!

Tools Built Using LLVM

Compilers!

JITs!

Security Hardening Tools!

Formal Verification!

Bug Finding Tools!

Profiling Tools!

Things to Do in the Compiler Zoo

- Add a security check to every load and store
- Create a memory access trace
- Check pointer arithmetic on certain types of variables
- Trace atomic modifications to a memory location
- Change order of local variables in stack frame

What do you want to do with LLVM?

LLVM Source Code Structure

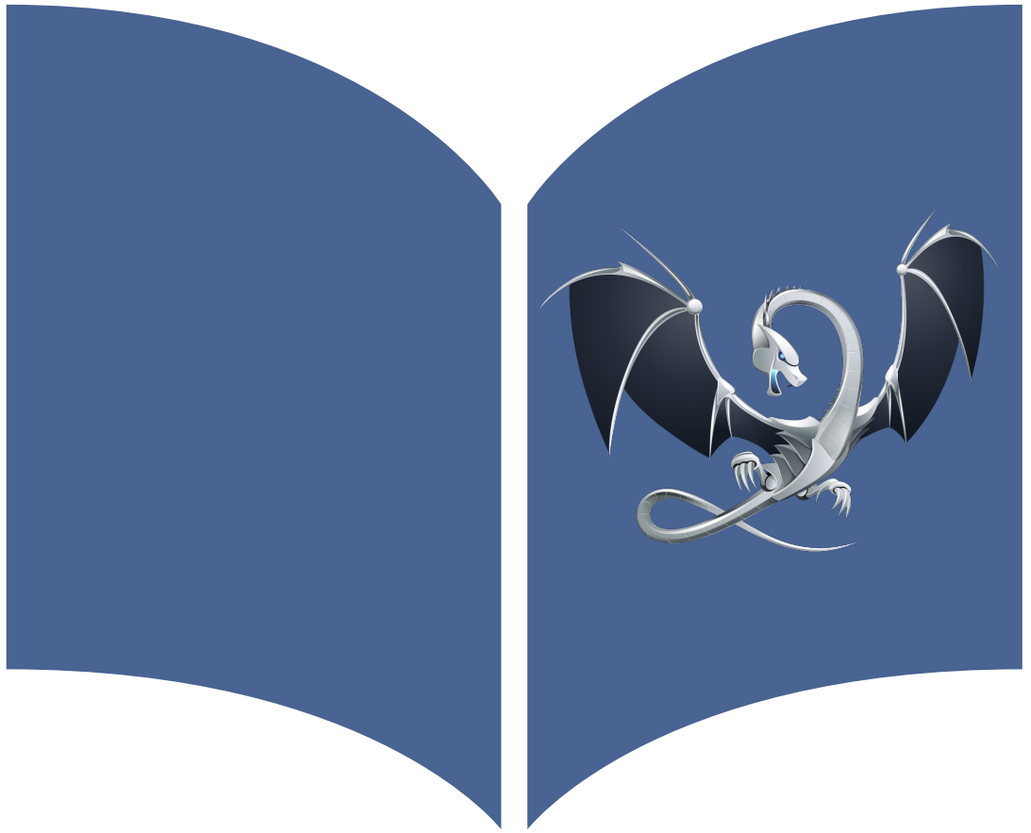
- LLVM is primarily a set of libraries
- We use the libraries to create LLVM-based tools

Programming Background

- C++
  - Other language bindings exist, but C++ is “native”
- Know how to use classes, pointers, and references
- Know how to use C++ iterators
- Know how to use Standard Template Library (STL)

Helpful Documents

- LLVM Language Reference Manual
- LLVM Programmer's Manual
- How to Write an LLVM Pass
- Online LLVM Doxygen documents



Getting Involved with LLVM

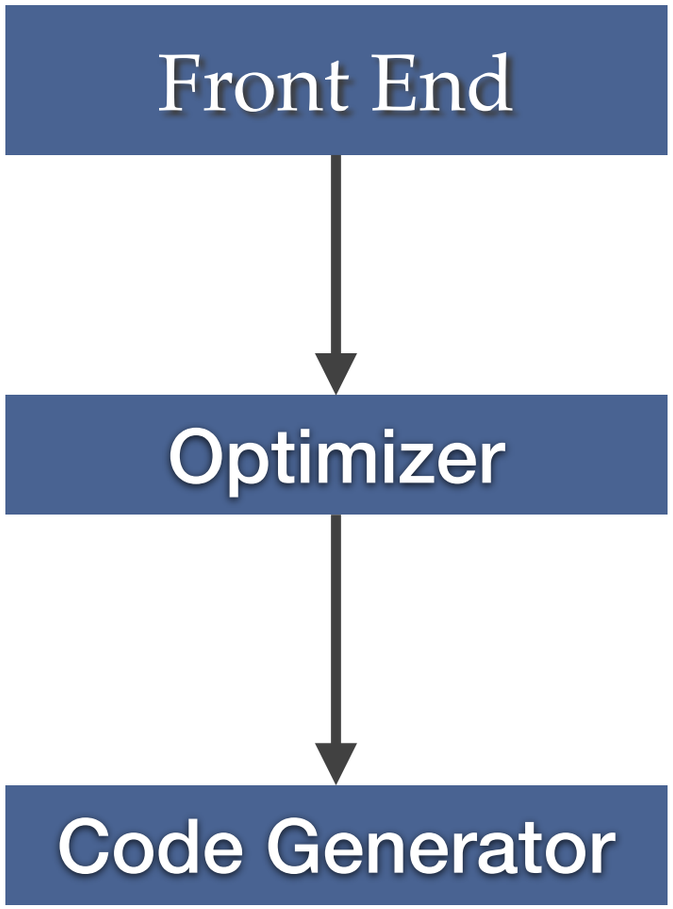
- Research on program analysis (NSF REUs)
- Google Summer of Code projects
- Apple, Samsung, Google, Facebook build LLVM tools
- LLVM Developer's Meeting
  - One in California; one in Europe
  - Can present talks, posters, BoFs, etc.

Please fill out feedback form:

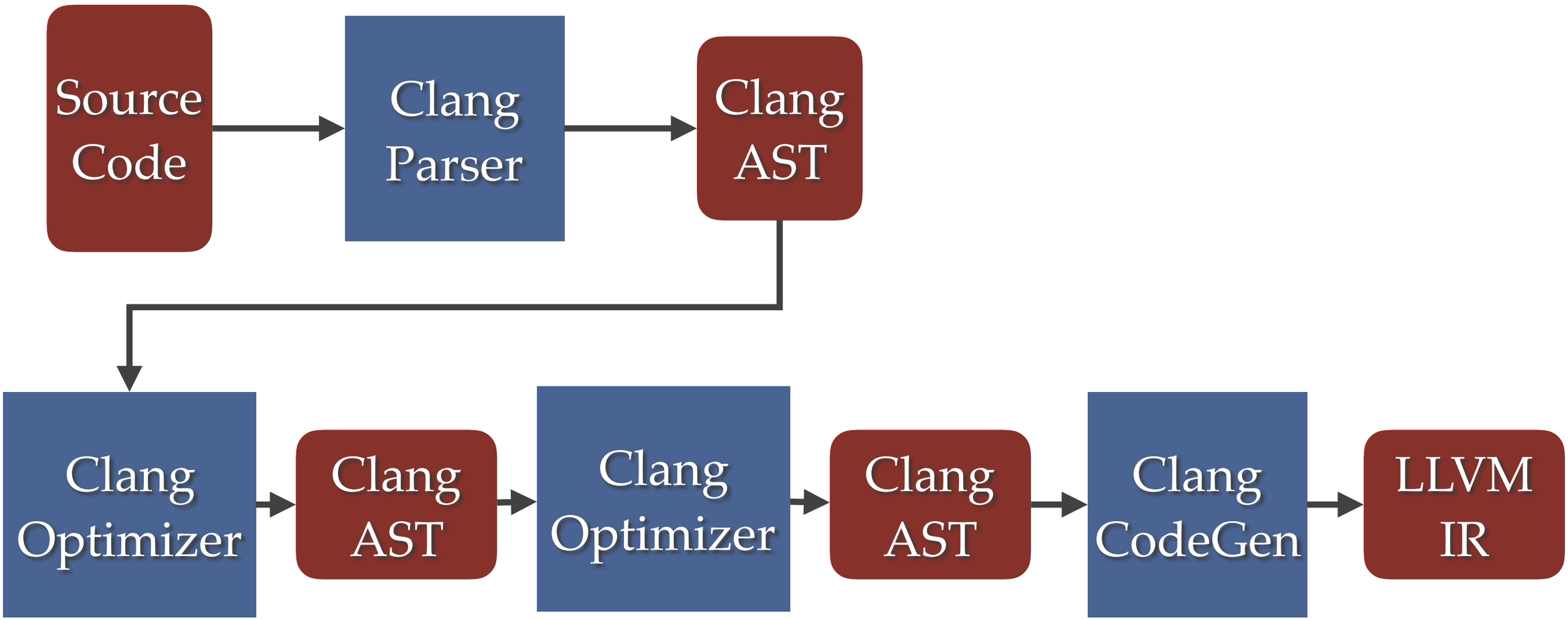
https://forms.gle/ib3Ng6osSFqNoQGD7

LLVM Compiler Structure

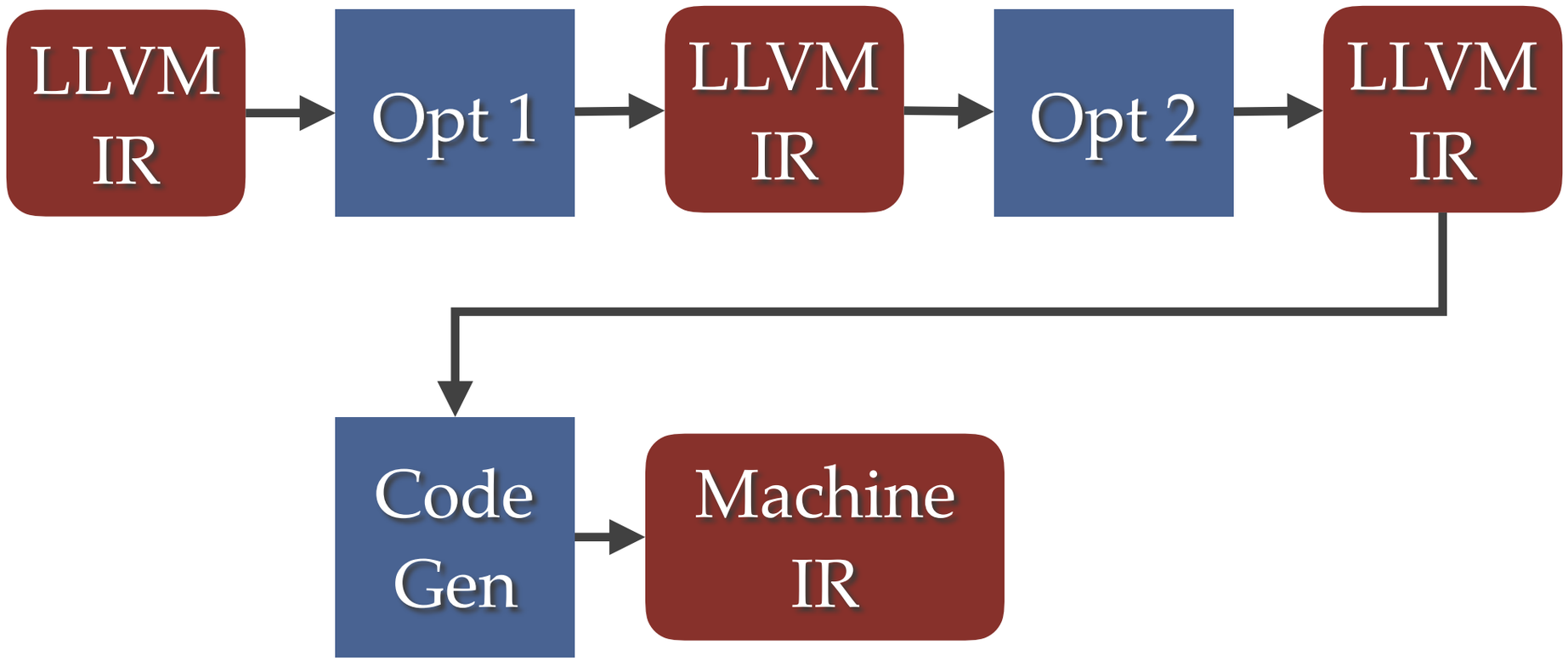
Ahead of Time (AOT) Compiler



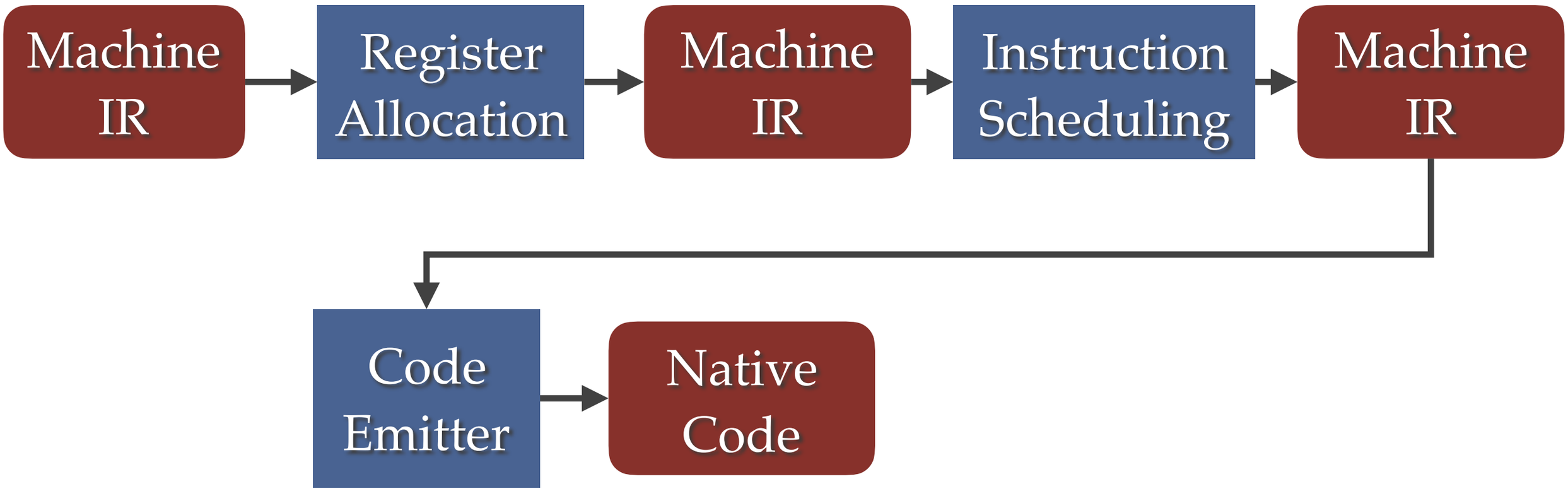
Front End Structure



Optimizer Structure



Code Generator Structure



LLVM Toolchain Overview

Intermediate Representation	Description
Clang AST	Describes structure of source code (if-statements, while-loops)
LLVM IR	Architecture independent code in SSA form
Machine IR	Native code (machine registers; native code instructions)

LLVM Toolchain Overview

Intermediate Representation	Description
Clang AST	Describes structure of source code (if-statements, while-loops)
LLVM IR	Architecture independent code in SSA form
Machine IR	Native code (machine registers; native code instructions)

LLVM

Intermediate Representation

LLVM IR is a *language* into which programs are translated for analysis and transformation (optimization)

LLVM IR Forms

- LLVM Assembly Language
  - Text form saved on disk for humans to read
- LLVM Bitcode
  - Binary form saved on disk for programs to read
- LLVM In-Memory IR
  - Data structures used for analysis and optimization

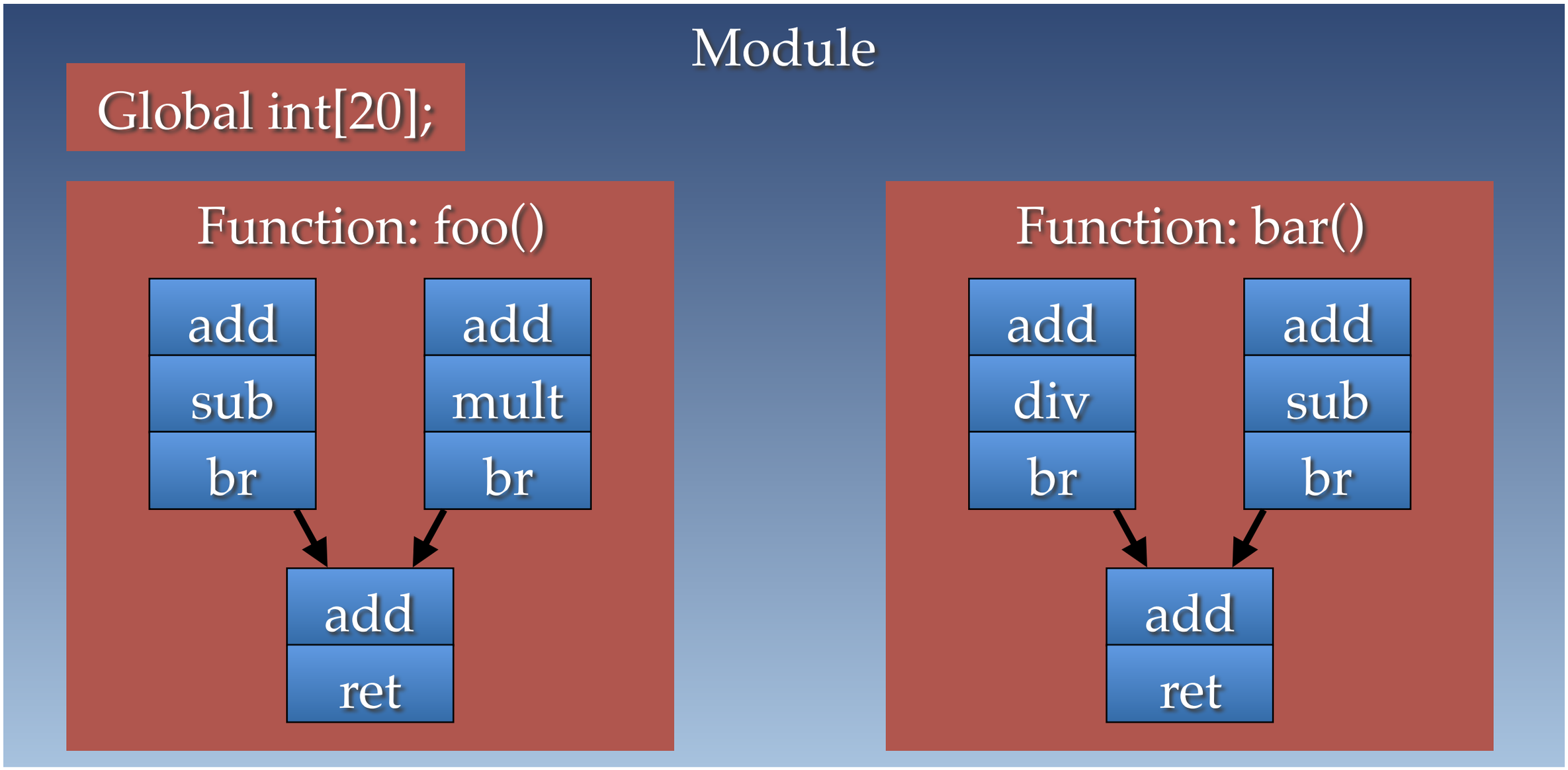
LLVM Assembly Language

```
define i32 @foo(i32, i32) local_unnamed_addr #0 {
  %3 = tail call i32 @bar(i32 %0) #2
  %4 = add nsw i32 %1, %0
  %5 = sub i32 %4, %3
  ret i32 %5
}

declare i32 @bar(i32) local_unnamed_addr #1
```

Overview of LLVM IR

- Each assembly/bitcode file is a Module
- Each Module is comprised of
  - Global variables
  - A set of Functions which are comprised of
    - A set of basic blocks which are comprised of
      - A set of instructions

LLVM Bitcode File



LLVM Module with One Function

```
define i32 @foo(i32, i32) local_unnamed_addr #0 {
  %3 = icmp ult i32 %0, %1
  br i1 %3, label %4, label %6


 %5 = tail call i32 @bar(i8* getelementptr inbounds ([7 x i8], [7 x i8]* @.str, i64 0, i64 0)) #2
  br label %8


 %7 = tail call i32 @bar(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.1, i64 0, i64 0)) #2
  br label %8


 %9 = add i32 %1, %0
  ret i32 %9
}
```

LLVM Instruction Set

- RISC-like architecture
  - Virtual registers in SSA form
  - Load/store instructions to read/write memory objects
  - All other instructions read or write virtual registers

LLVM Memory Objects

- Global Variables
- Memory allocated on the stack
- Memory allocated on the heap

Instructions for Computation

- Arithmetic and binary operators
  - Two's complement arithmetic (add, sub, multiply, etc)
  - Bit-shifting and bit-masking
- Pointer arithmetic (getelementptr or "GEP")
- Comparison instructions (icmp, fcmp)
  - Generates a boolean result

Memory Access Instructions

- Load instruction reads memory
- Store instruction writes to memory
- Atomic compare and exchange
- Atomic read / modify / write

Control Flow Instructions

- Terminator instructions
  - Indicate which basic block to jump to next
  - conditional branch, unconditional branch, switch
  - Return instruction to return to caller
  - Unwind instruction for exception handling
- Call instruction calls a function
  - It can occur in the middle of a basic block

Memory Allocation Instructions

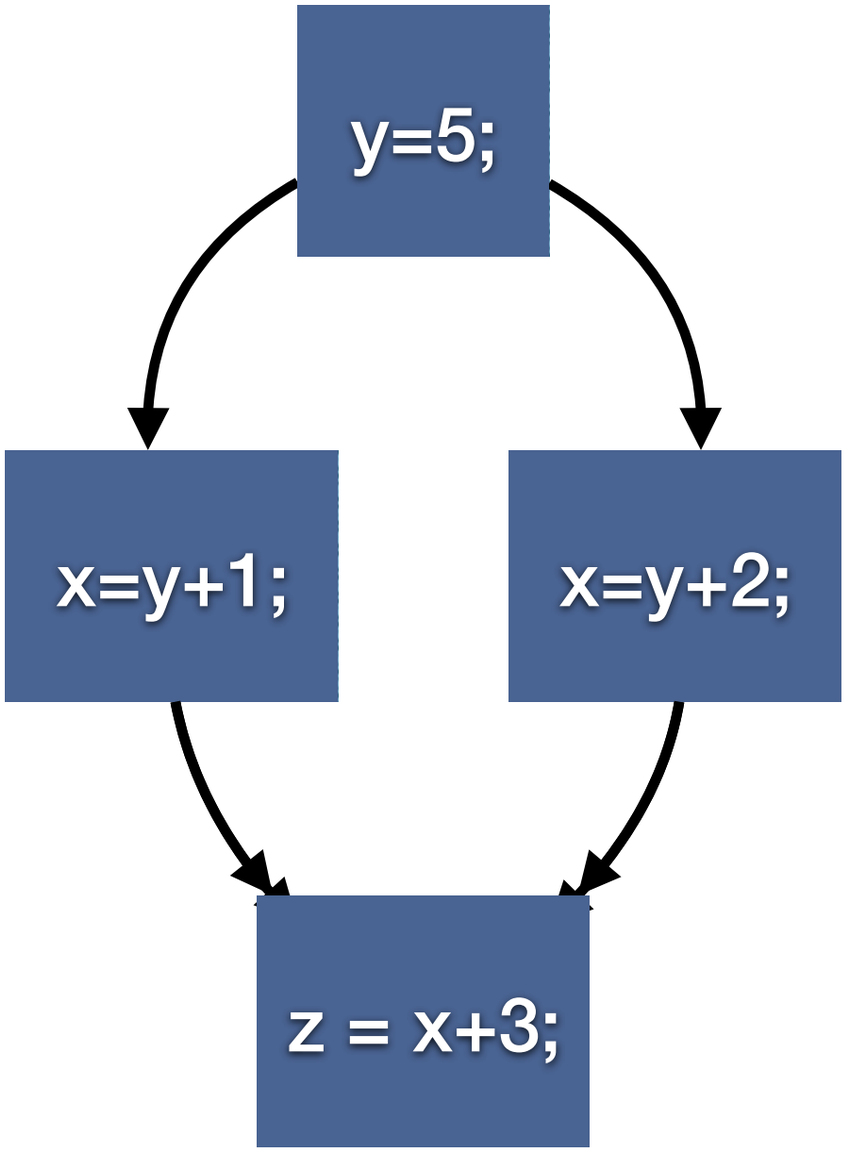
- Stack allocation (alloca)
  - Allocates memory on the stack
- Calls to heap-allocation functions (e.g., malloc())
  - Not a special instruction; just uses a call instruction
- Global variable declarations
  - Not really instructions, but allocate memory
  - All globals are pointers to memory objects

Single Static Assignment (SSA)

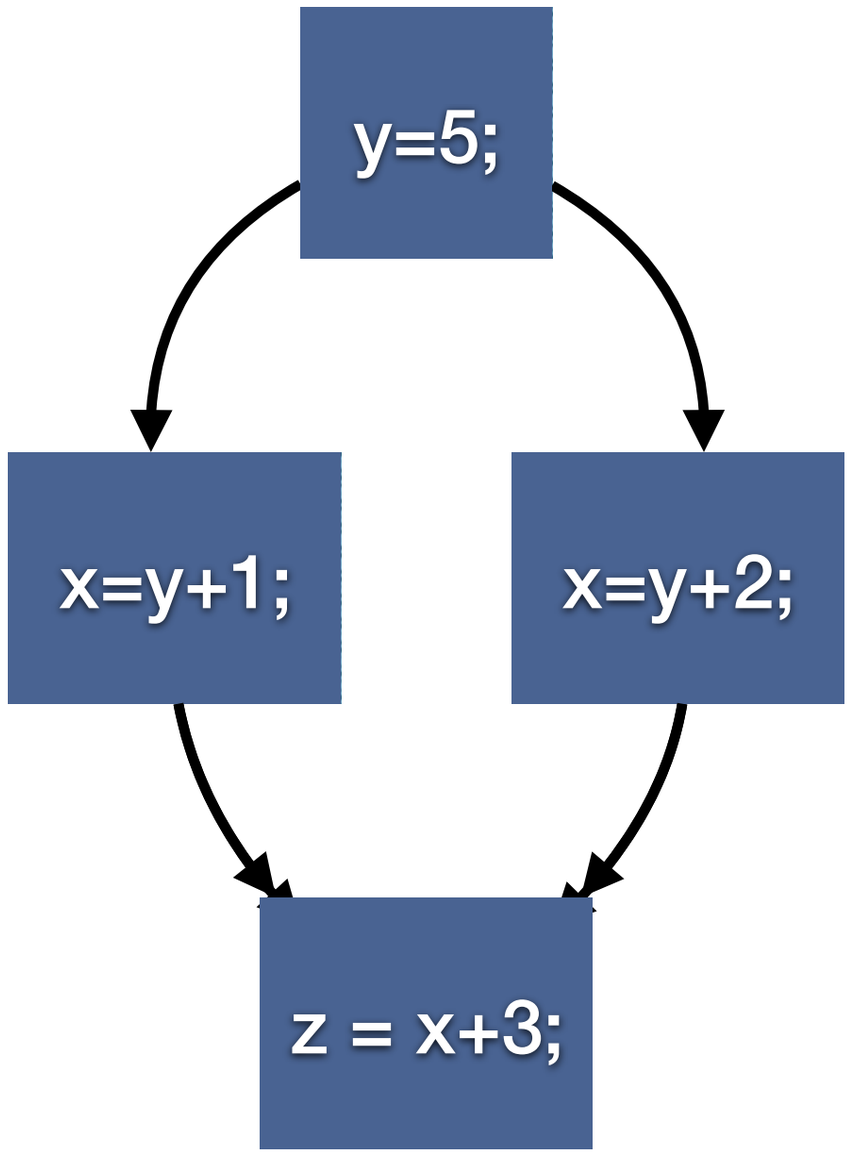
- Each function has infinite set of ***virtual registers***
- Only ***one*** instruction assigns a value to a virtual register (called the ***definition*** of the register)
- An instruction and the register it defines are ***synonymous***

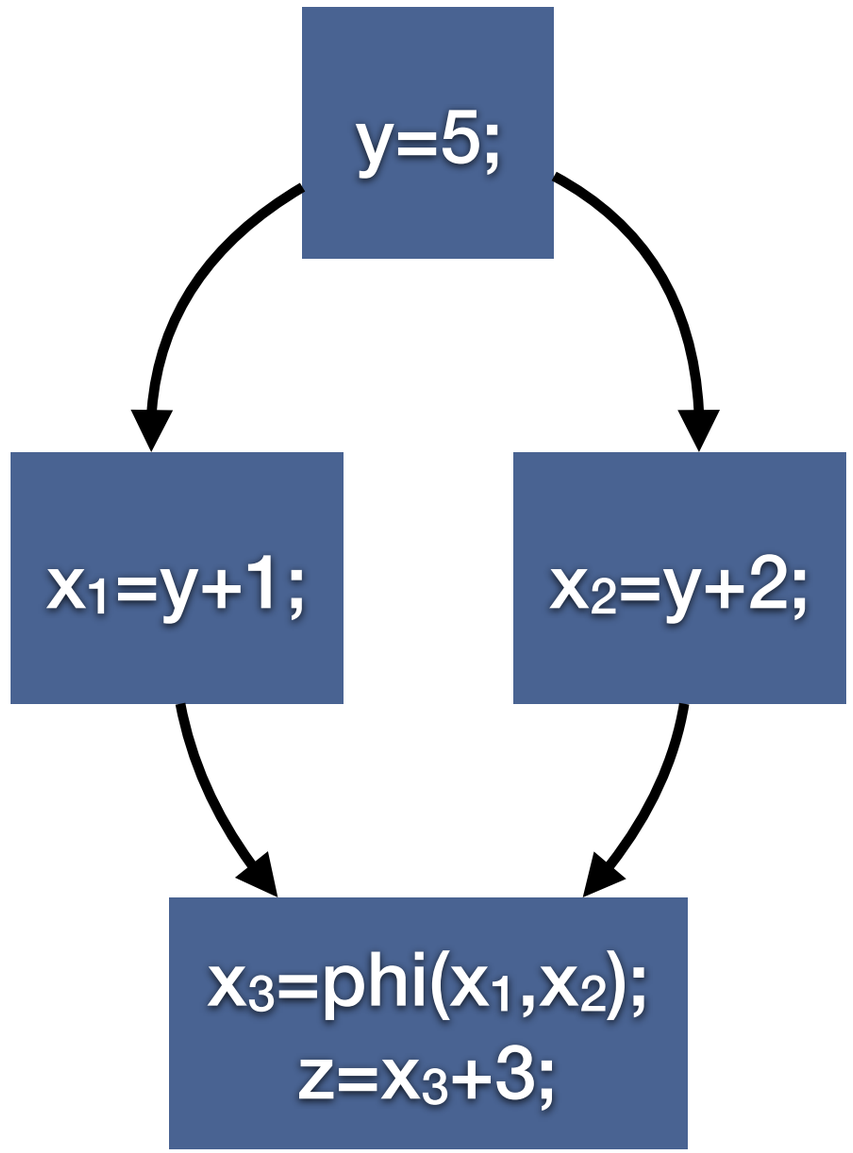
```
%z = add %x, %y
```

The Almighty Phi Node!



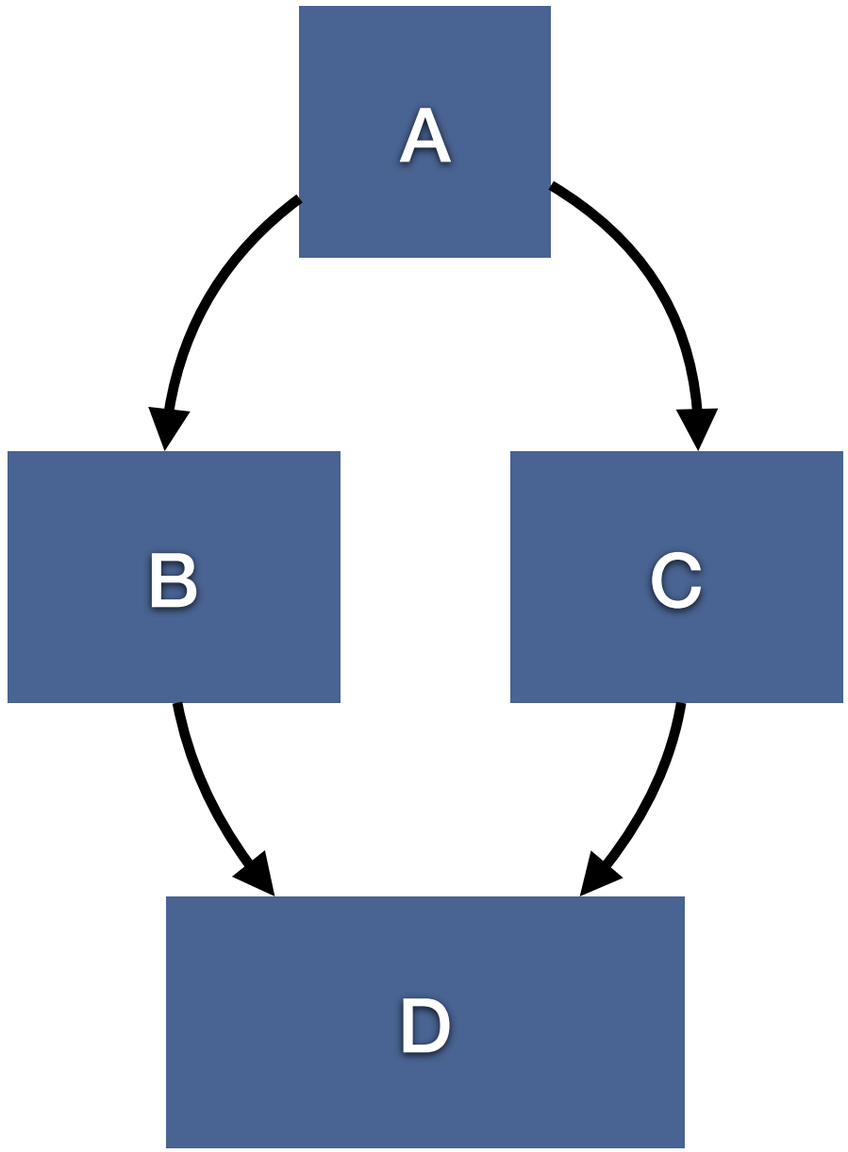
The Almighty Phi Node!





Domination

- The definition of a virtual register must dominate all of its uses
  - Except uses by phi-nodes
- A dominates B, C, and D

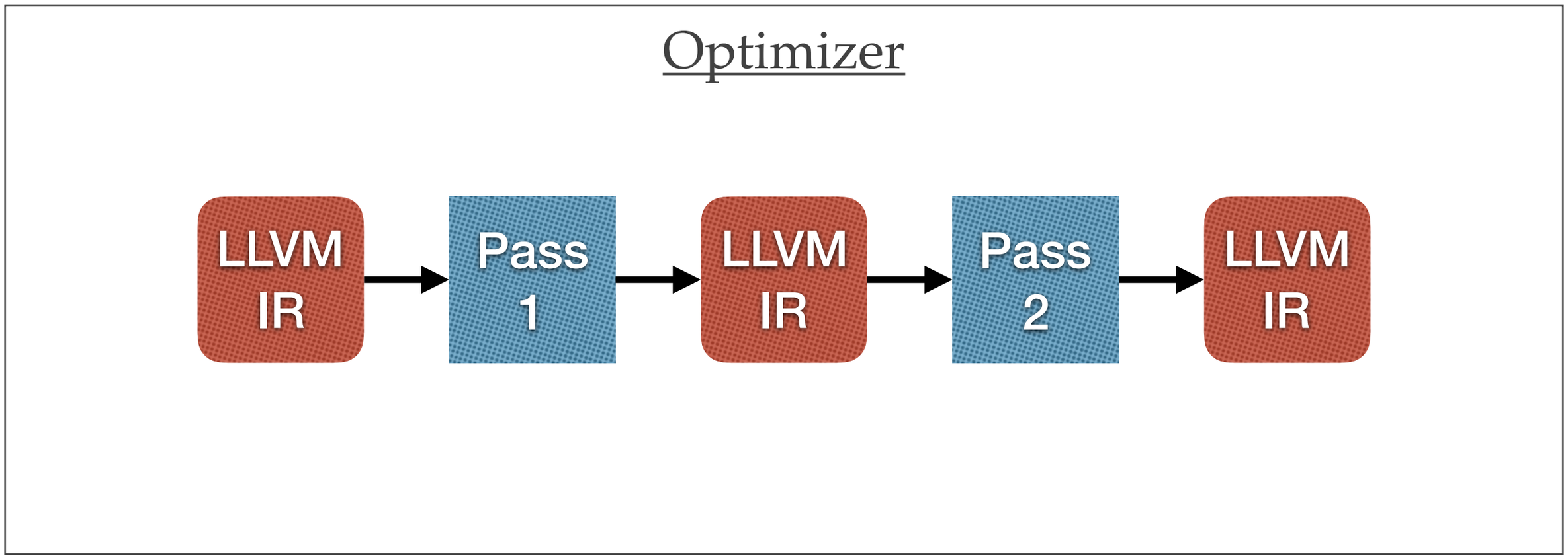


Writing an LLVM Pass

LLVM Passes: Separation of Concerns

- Break optimizer into passes
- Each pass performs one analysis or one transformation

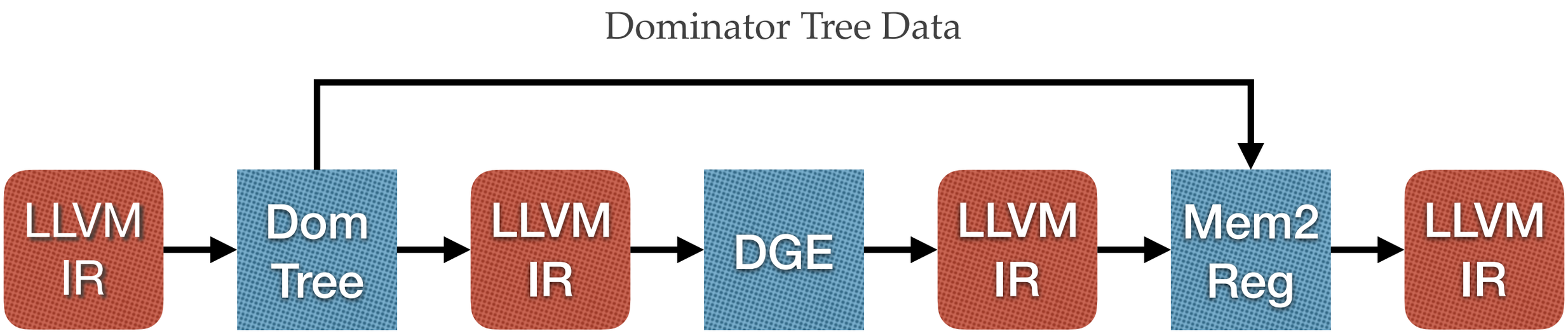
LLVM Passes



Two Types of Passes

- Passes that analyze code
  - Does not modify the program
  - Provides information "out of band" to other passes
- Passes that transform code
  - Make modifications to the code

LLVM Passes


Dominator Tree Data
LLVM IR
Dom Tree
LLVM IR
DGE
LLVM IR
Mem2 Reg
LLVM IR

LLVM IR Pass Types

- ModulePass
- FunctionPass
- BasicBlockPass
- I recommend ignoring “funny” passes
  - LoopPass
  - RegionPass

Rules for LLVM Passes

- Only modify values and instructions at scope of pass
  - ModulePass can modify anything
  - FunctionPass should not modify anything outside of the function
  - BasicBlockPass should not modify anything outside of the basic block

Important Pass Methods: getAnalysisUsage()

- Tells PassManager which analysis passes you need
  - PassManager will schedule analysis passes for you
  - Cannot schedule transform passes this way

- Tells PassManager which analysis results are valid after a transformation
  - Avoids re-running expensive analysis passes

runOnModule()

- Entry point for ModulePass
- Passes a reference to the Module
- Can locate functions, basic blocks, globals from Module
- Return true if the pass modifies the program
  - An analysis pass ***always*** returns false.
  - A transform pass can return either true or false.

runOnFunction()

- Called for each function in the Module
- Passed reference to Function
- Return false for no modifications; true for modifications

runOnBasicBlock()

- You get the idea…

MyPass.h Example

```
class MyPass : public ModulePass {
  private:
    unsigned int analyzeThis (Instruction *I);

  public:
    static char ID;
    MyPass() : ModulePass(ID) {}
    const char *getPassName() const { return “My LLVM Pass"; }
    virtual bool runOnModule (Module & M);
    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      // We require Dominator information
      AU.addRequired<DominatorTree>();
    }
   unsigned int getAnalysisResultFor (Instruction *I);
};
```

MyPass.cpp Example

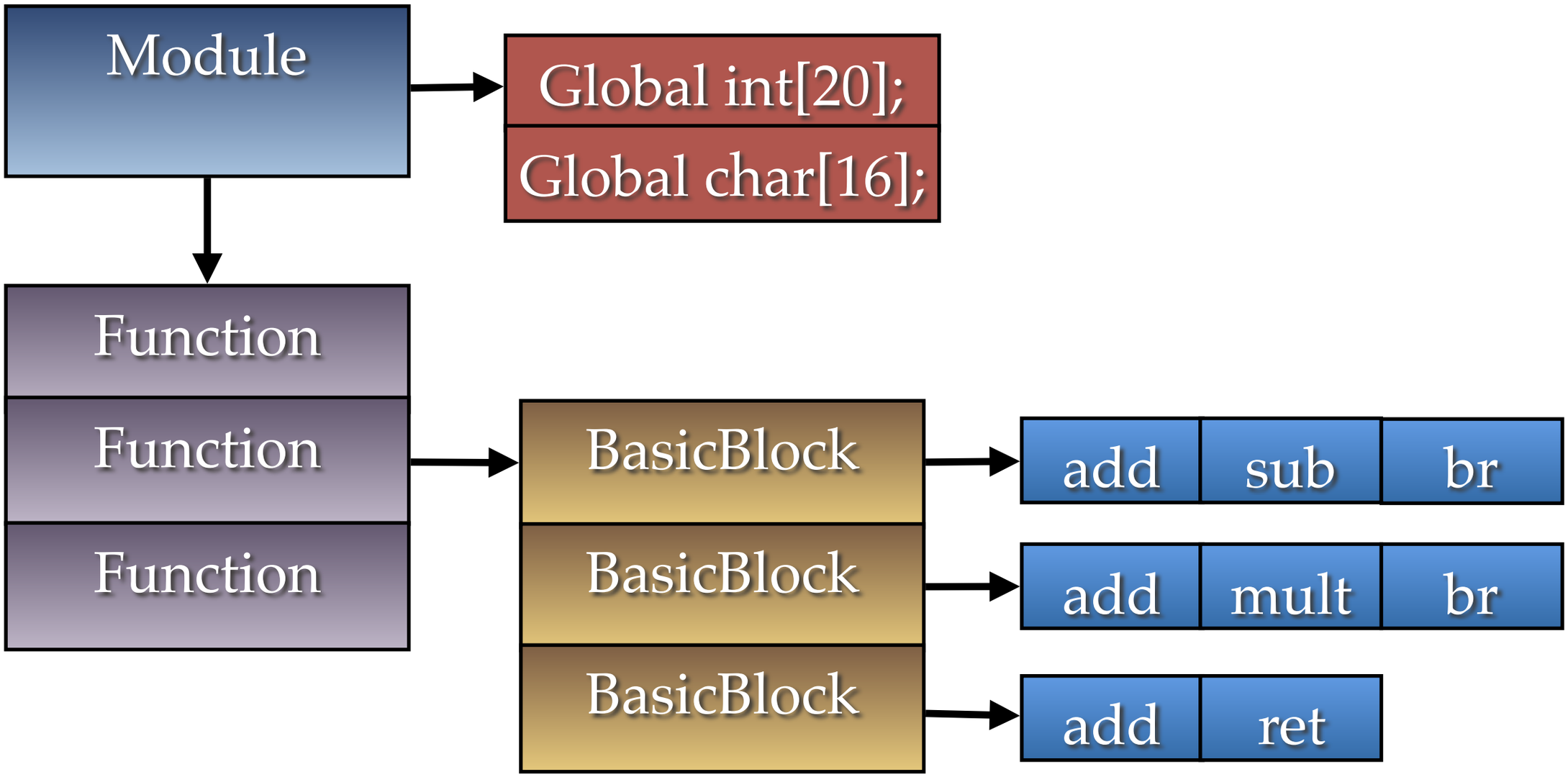
```
static RegisterPass<MyPass> P (“mypass”, “My First LLVM Analysis”);

bool
MyPass::runOnModule (Module & M) {
  //
  // Iterate over all instructions within a Module
  //
  for (Module::iterator fi = M.begin(); fi != M.end(); ++fi) {
    for (Function::iterator bi = fi->begin(); bi != fi->end(); ++bi) {
      for (BasicBlock::iterator it = bi->begin(); it != bi->end; ++it) {
        Instruction * I = *it;
      }
    }
  }
}
```

In-Memory LLVM IR

LLVM Classes

- There is a class for each type of IR object
  - Module class
  - Function class
  - BasicBlock class
  - Instruction class
- Classes provide iterators for objects within them

LLVM In-Memory IR



Class Iterators

- Each class provides iterators for items it contains
  - Module::iterator iterates over functions
  - Function::iterator iterates over basic blocks
  - BasicBlock::iterator iterates over instructions

Iterator Example

```
//
// Iterate over all instructions within a BasicBlock
//
BasicBlock * BB = …;
BasicBlock::iterator it;
BasicBlock::iterator ie;
for (it = BB->begin(), end = BB->end(); it != end; ++it) {
 Instruction * I = *it;
};
```

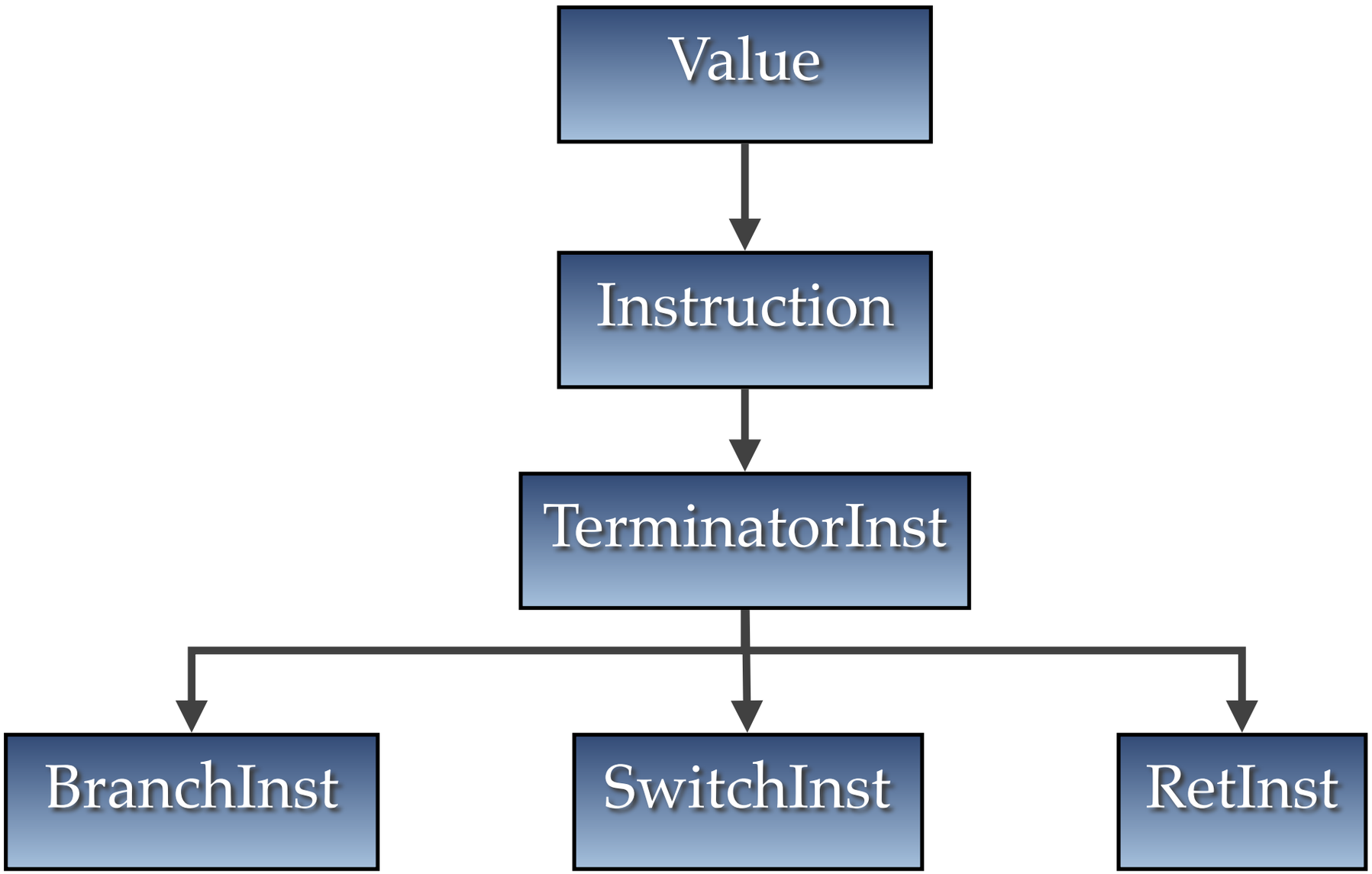
MyPass.cpp Example (Reprise)

```
static RegisterPass<MyPass> P (“mypass”, “My First LLVM Analysis”);

bool
MyPass::runOnModule (Module & M) {
  //
  // Iterate over all instructions within a Module
  //
  for (Module::iterator fi = M.begin(); fi != M.end(); ++fi) {
    for (Function::iterator bi = fi->begin(); bi != fi->end(); ++bi) {
      for (BasicBlock::iterator it = bi->begin(); it != bi->end; ++it) {
        Instruction * I = *it;
      }
    }
  }
}
```

LLVM Class Hierarchy

- Anything that is an SSA value is a subclass of Value
- All Instruction classes are a subclass of Instruction
- Similar instructions share a common superclass

Simplified LLVM Class Hierarchy



Locating Branch Instructions

```
//
// Iterate over all instructions within a BasicBlock
//
BasicBlock::iterator it;
BasicBlock::iterator ie;

for (it = BB->begin(), end = BB->end(); it != end; ++it) {
 Instruction * I = *it;
 if (BranchInst * BI = dyn_cast<BranchInst>(I)) {
  // Do something with branch instruction BI
 }
}
```

Casting to Subclass in LLVM

Casting Function	Description	Example
isa<Class>()	Return true or false if value is of that class.	isa<BranchInst>(V)
dyn_cast<Class>()	Returns pointer to object of type Class or NULL	dyn_cast<BranchInst>(V)

Locating Branch Instructions

```
//
// Iterate over all instructions within a BasicBlock
//
BasicBlock::iterator it;
BasicBlock::iterator ie;

for (it = BB->begin(), end = BB->end(); it != end; ++it) {
 Instruction * I = *it;

 if (BranchInst * BI = dyn_cast<BranchInst>(I)) {

  // Do something with branch instruction BI
 }
}
```

LLVM Instruction Classes

- BinaryOperator - add, sub, mult, shift, and, or, etc.
- GetElementPointerInst
- LoadInst, Storeinst
- BranchInst, SwitchInst, RetInst
- CallInst
- CastInst

LLVM Class Methods

- Each class has methods to get information on value
  - BranchInst - Iterator over successor basic blocks
  - StoreInst - Get pointer operands of store instruction
  - GetElementPtrInst - Get indices used as operands
  - Instruction - Get containing basic block
- Method might belong to a superclass

Beyond the Tutorial

Data Flow Analysis

- The Dragon Book, Fourth Edition
- Papers on SSA-based algorithms
- Kam-Ullman paper on iterative data-flow analysis

Getting Involved with LLVM

- Research on program analysis (NSF REUs)
- Google Summer of Code projects
- Apple, Samsung, Google, Facebook build LLVM tools
- LLVM Developer's Meeting
  - One in California; one in Europe
  - Can present talks, posters, BoFs, etc.