软件分析与验证前沿

苏亭

软件科学与技术系

LLVM Primer

Slides are from Shunqi Liu, Aaditya Naik, and MayurNaik

Roadmap

Welcome! This primer has four parts:

Part I: Overview of LLVM

Part II: Structure of LLVM IR

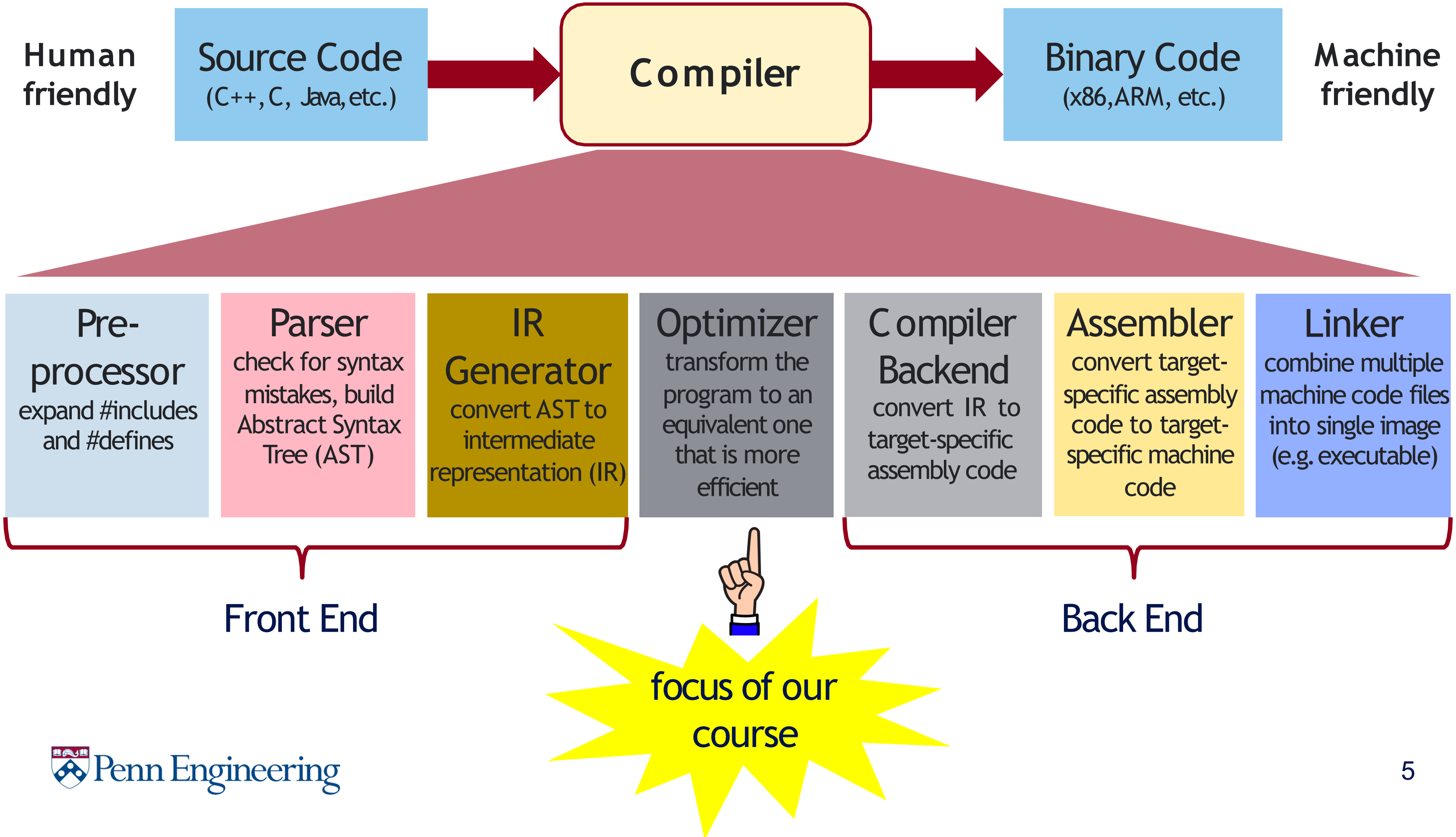
Part III:The LLVM API

Part IV: Navigating the Documentation

Part I: Overview of LLVM

What is LLVM?

A modular and reusable compiler framework supporting multiple front-ends and back-ends.



Human friendly

Source Code (C++, C, Java, etc.)

Compiler

Binary Code (x86, ARM, etc.)

Machine friendly

Pre-processor: expand #includes and #defines

Parser: check for syntax mistakes, build Abstract Syntax Tree (AST)

IR Generator: convert AST to intermediate representation (IR)

Optimizer: transform the program to an equivalent one that is more efficient

Compiler Backend: convert IR to target-specific assembly code

Assembler: convert target-specific assembly code to target-specific machine code

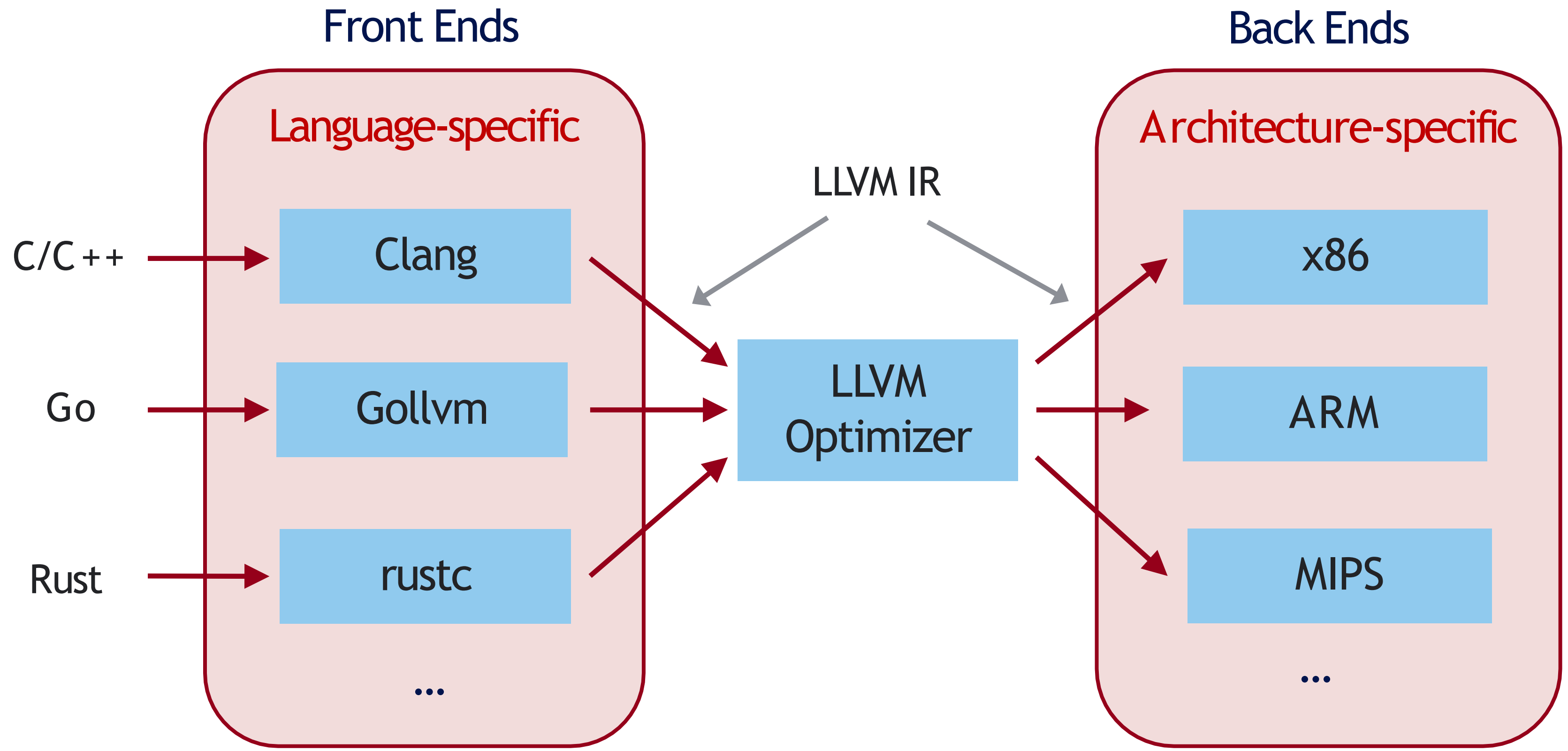
Linker: combine multiple machine code files into single image (e.g. executable)

Front End

Back End

focus of our course

Architecture of LLVM



Front Ends

Language-specific

C/C++

Go

Rust

Clang

Gollvm

rustc

...

LLVM IR

LLVM Optimizer

Back Ends

Architecture-specific

x86

ARM

MIPS

...

C Source Code:
if (b==0) a=0;

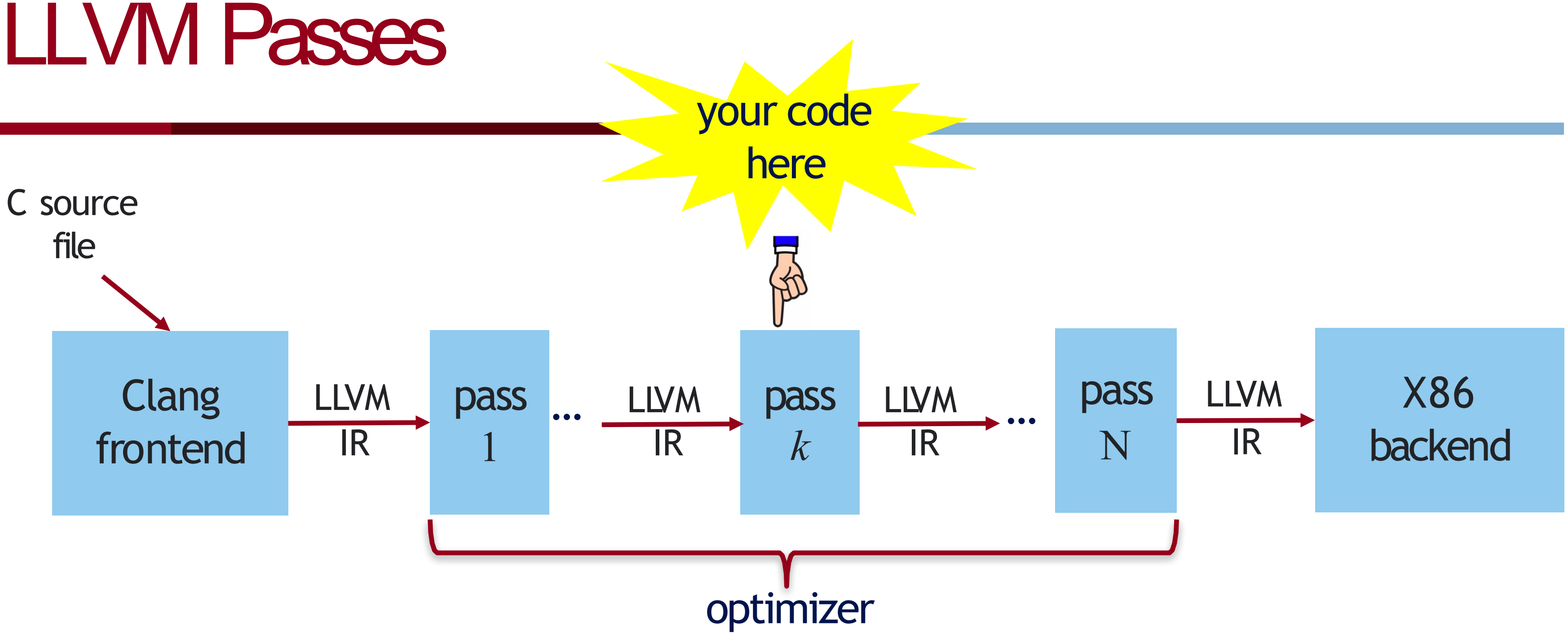
Front End →

```
LLVM IR:
%cmp = icmp eq i32 %0, 0
br i1 %cmp, label %if.then, label %if.end
```

Back End →

```
x86 assembly:
CMP ECX, 0
SETBZ EAX
```

LLVM Passes



The LLVM Optimizer (**opt**) is a series of “passes” that run one after another

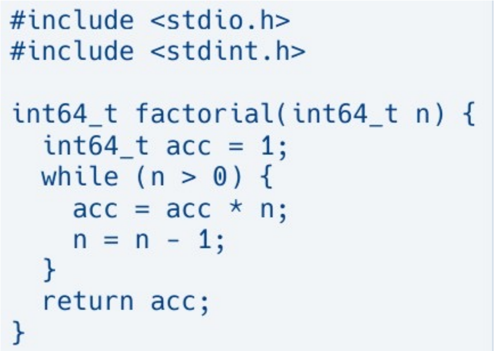
- Two kinds of passes: analysis and transformation
  - Analysis pass analyzes LLVM IR to *check* program properties
  - Transformation pass transforms LLVM IR to *monitor* or *optimize* the program

=> Analysis passes do not change code; transformation passes do

LLVM is typically extended by implementing new passes that look at and change the LLVM IR as it flows through the compilation process.

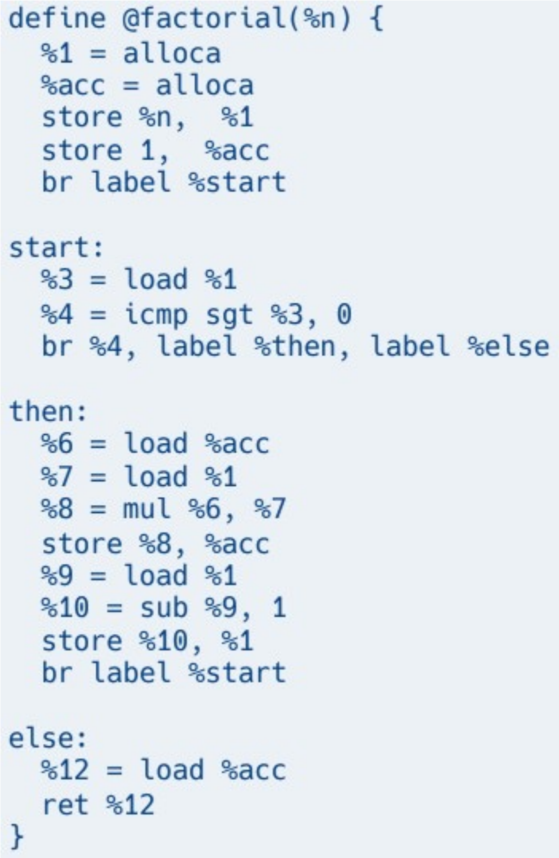
Example: Factorial Program

Factorial.c



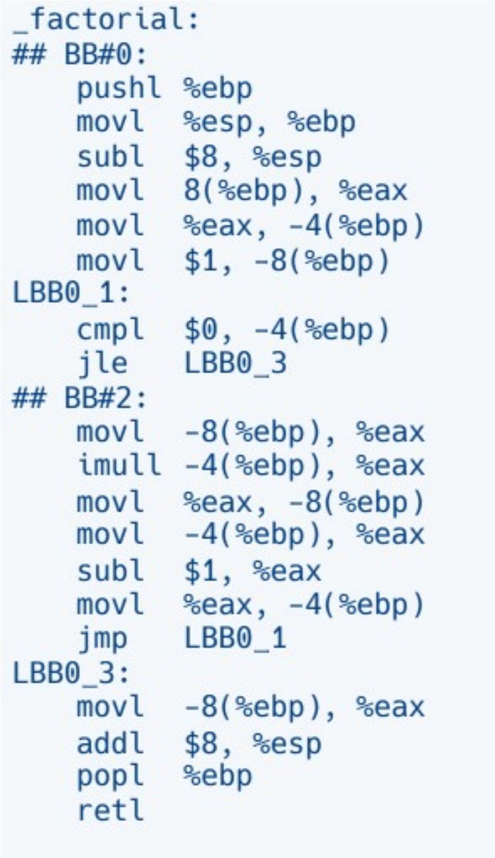
```c
#include <stdio.h>
#include <stdint.h>

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

Factorial.ll



```
define @factorial(%n) {
  %1 = alloca
  %acc = alloca
  store %n,  %1
  store 1,  %acc
  br label %start

start:
  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

else:
  %12 = load %acc
  ret %12
}
```

Factorial.s



```
_factorial:
## BB#0:
    pushl %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  8(%ebp), %eax
    movl  %eax, -4(%ebp)
    movl  $1, -8(%ebp)
LBB0_1:
    cmpl  $0, -4(%ebp)
    jle   LBB0_3
## BB#2:
    movl  -8(%ebp), %eax
    imull -4(%ebp), %eax
    movl  %eax, -8(%ebp)
    movl  -4(%ebp), %eax
    subl  $1, %eax
    movl  %eax, -4(%ebp)
    jmp   LBB0_1
LBB0_3:
    movl  -8(%ebp), %eax
    addl  $8, %esp
    popl  %ebp
    retl
```

Why LLVM IR?

- Easy to translate from the level above
- Easy to translate to the level below
- Narrow interface (simpler phases/optimizations)
- The IR language is independent of the source and target languages in order to maximize the compiler's ability to support multiple source and target languages.

Example: Source language might have "while", "for", and "foreach" loops

- IR language might have only "while" loops and sequence
- Translation eliminates "for" and "foreach"

LLVM IR Normal Form

Instead of handling AST of “((1 + X4) + (3 + (X1 * 5)))”

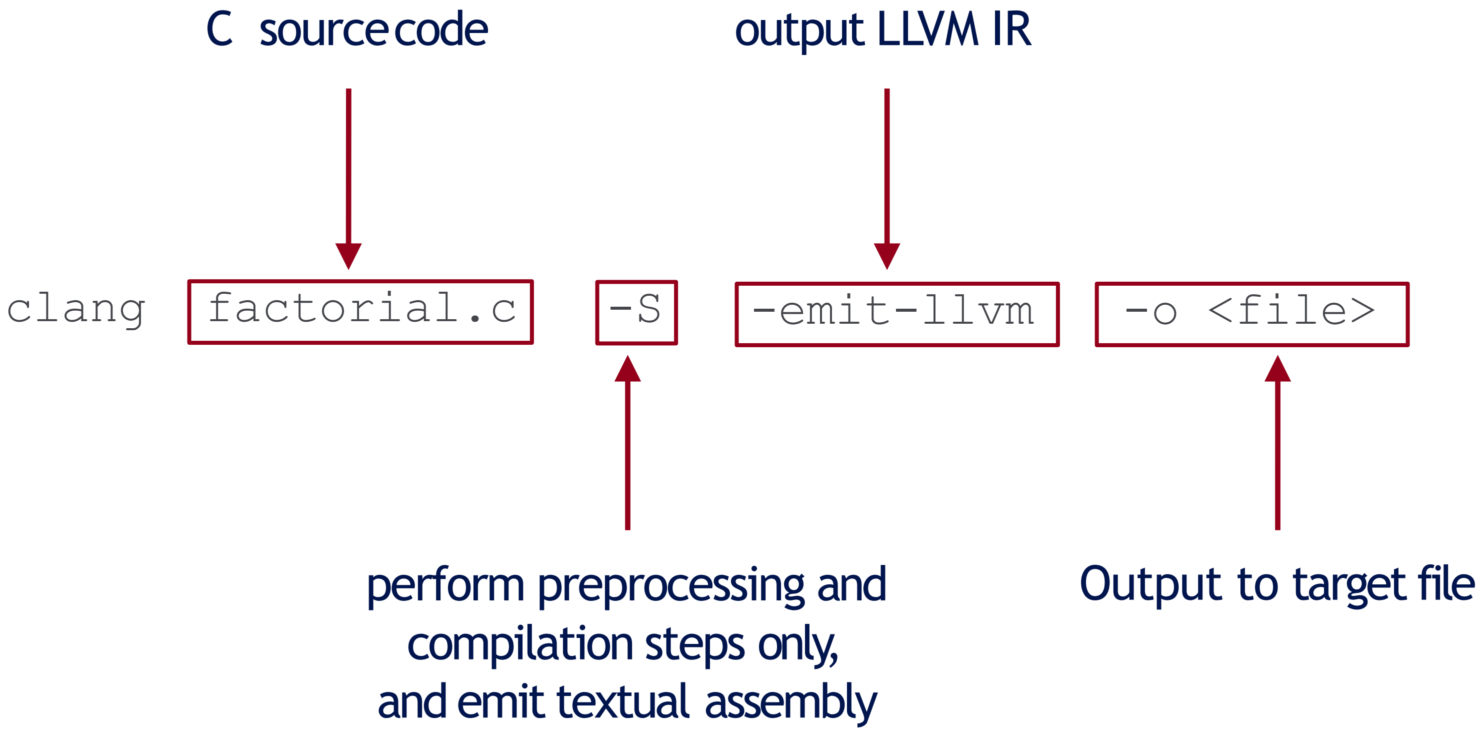
```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

we have to handle:

```
tmp0 = 1 + X4
tmp1 = X1 * 5
tmp2 = 3 + tmp1
tmp3 = tmp0 + tmp2
```

- Translation makes the order of evaluation explicit.
- Names intermediate values.
- Introduced temporaries are never modified.

Generate LLVM IR Yourself!



```
clang factorial.c -S -emit-llvm -o <file>
```

C source code

output LLVM IR

perform preprocessing and
compilation steps only,
and emit textual assembly

Output to target file

History of LLVM

- The LLVM project was initially developed by Vikram Adve and Chris Lattner at the University of Illinois at Urbana-Champaign in 2000. Their original purpose was to develop dynamic compilation techniques for static and dynamic programming languages.
- In 2005, Lattner entered Apple and continued to develop LLVM.
- In 2013, LLVM initially represented Low-Level Virtual Machines, but as the LLVM family grew larger, the original meaning was no longer applicable.
- Today, LLVM + Clang comprise a total LOC of 2.5 million lines of C++ code.

Where is LLVM Used?

- Traditional C/C++ toolchain: Qualcomm Snapdragon LLVM compiler for `Android`
- Programming languages: Pyston – performance oriented `Python` implementation by LLVM
- Language runtime systems: LLILC – LLVM based `.NET` MSIL compiler
- GPU: Majority of `OpenCL` implementations based on Clang/LLVM
- Linux/FreeBSD: `Debian` experimenting with Clang/LLVM as an additional compiler

Google
SAMSUNG
Microsoft
ARM
intel
QUALCOMM
SONY
COMPUTER ENTERTAINMENT
NVIDIA
Imagination
IBM
CRAY
AMD
Linaro
AZUL SYSTEMS

Contributing companies

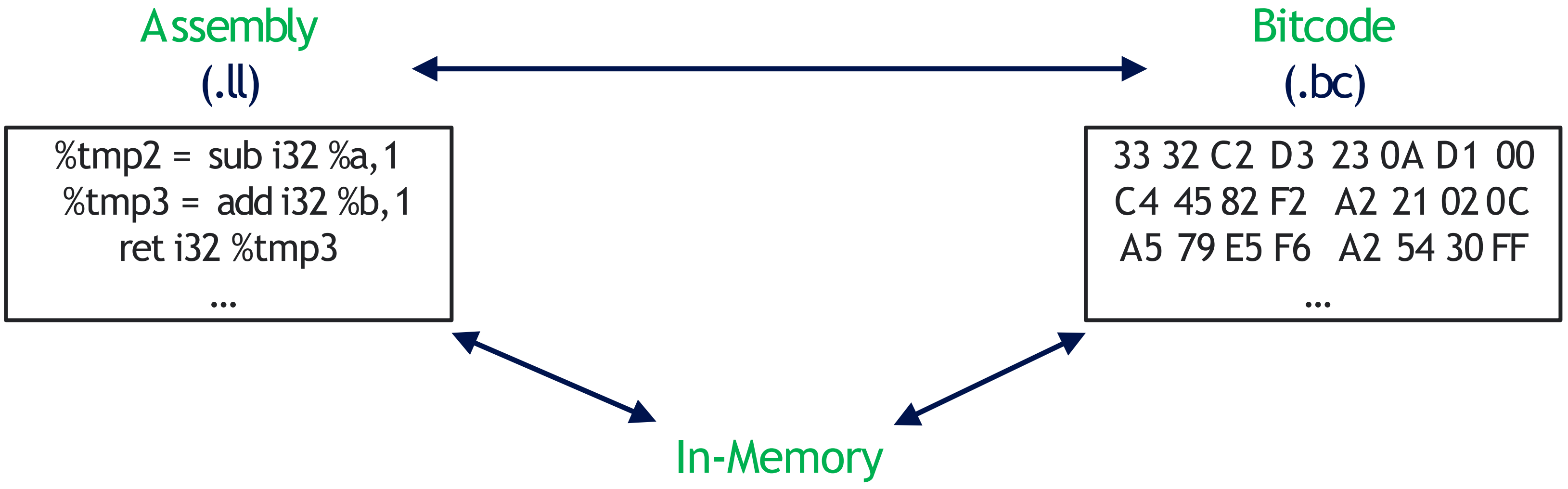
Source: “ Where is LLVM being used today?”, https://llvm.org/devmtg/2016-01/slides/fosdem-2016-llvm.pdf

Part II: Structure of LLVMIR

LLVM IR

Three formats:

- In-memory: binary in-memory format, used during compilation process
- Bitcode: binary on-disk format, suitable for fast loading
  (Obtained by “clang -emit-llvm -c factorial.c -o xxx.bc”)
- Assembly: human-readable format
  (Obtained by “clang -emit-llvm -S-c factorial.c -o xxx.ll”)



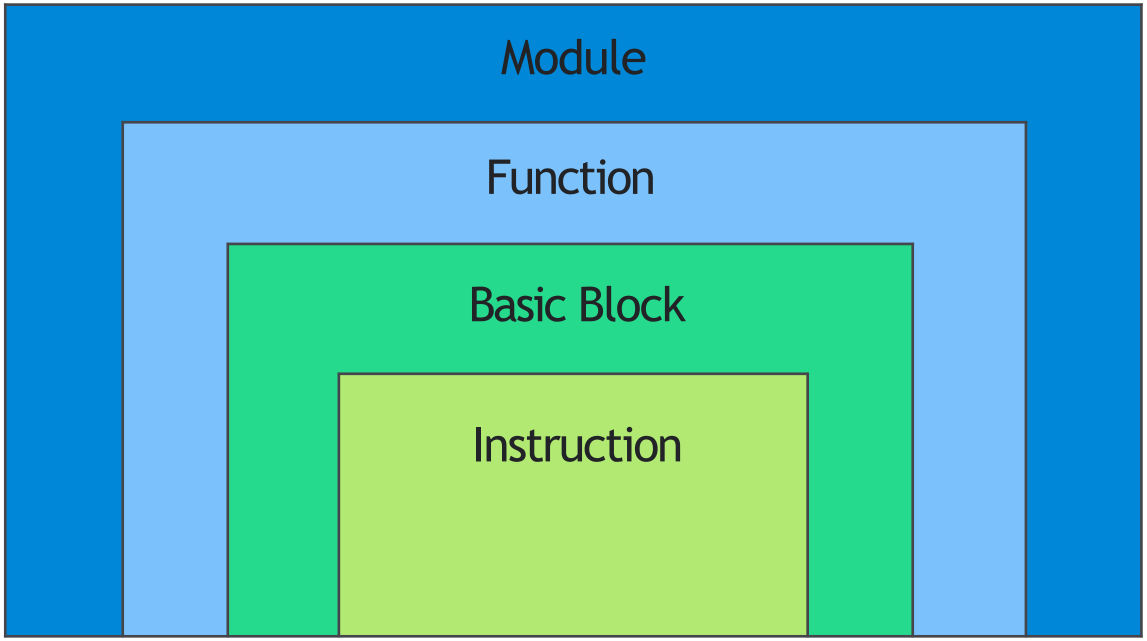
```
%tmp2 = sub i32 %a, 1
%tmp3 = add i32 %b, 1
ret i32 %tmp3
...
```

```
33 32 C2 D3 23 0A D1 00
C4 45 82 F2 A2 21 02 0C
A5 79 E5 F6 A2 54 30 FF
...
```

Compare to Java: instead of .class (bytecode), you get .bc

Program Structure in LLVM IR

Instruction ⊂Basic Block ⊂Function ⊂Module



Program Structure in LLVM IR

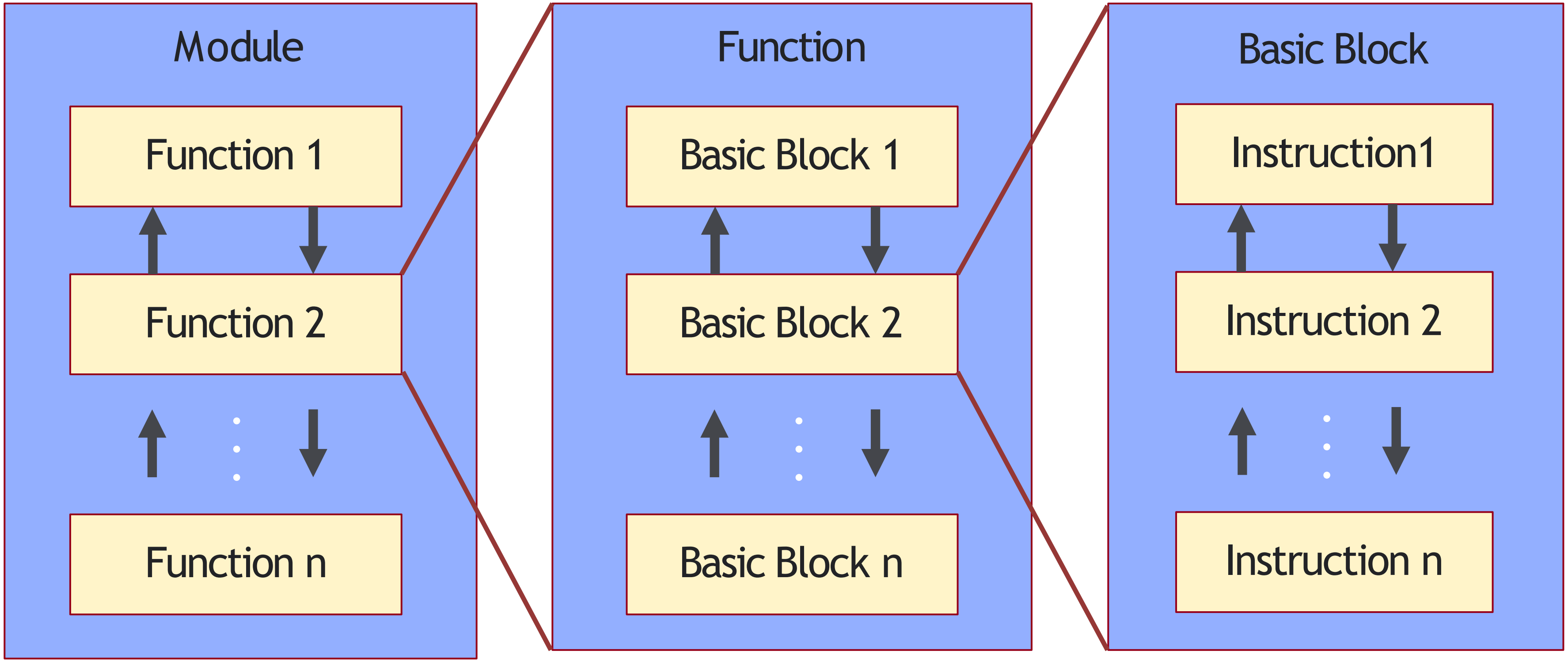
Module is a top-level container of LLVM IR, corresponding to each translation unit of the front-end compiler.

Function is a function in a programming language, including a function signature and several basic blocks. The first basic block in a function is called an entry basic block.

Basic Block is a set of instructions that are executed sequentially, with only one entry and one exit, and non-head and tail instructions will not jump to other instructions in the order they are executed.

Instruction is the smallest executable unit in LLVM IR; each instruction occupies asingle line.

LLVM IR Iterators



LLVM IR Iterators

Iterator types:

- Module::iterator
- Function::iterator
- BasicBlock::iterator
- Value::use_iterator
- User::op_iterator

Example uses:

Approach 1 (using STL iterator):

```
    for (Function::iterator FI = F->begin(); FI != F->end(); FI++) {
        for (BasicBlock::iterator BI = FI->begin(); BI != FI->end(); BI++) {
            // some operations
        }
    }
```

Approach 2 (using auto keyword):

```
    for (auto FI = F->begin(); FI != F->end(); FI++) {
        for (auto BI = FI->begin(); BI != FI->end(); BI++) {
            // some operations
        }
    }
```

Approach 3 (using InstIterator):

```
#include llvm/IR/InstIterator.h
for (inst_iterator It = inst_begin(F), E= inst_end(F); It != E;++It){
    // some operations
}
```

Variables and Types

Two kinds of variables: local and global

"%" indicates local variables:	%1 = add nsw i32 %a, %tmp
"@" indicates global variables:	@g = global i32 20, align 4

Two kinds of types: primitive (e.g. integer, floating-point) and derived (e.g. pointer, struct)

Integer type is used to specify an integer of desired bit width:

`i1`	A single-bit integer
`i32`	A 32-bit integer

Pointer type is used to specify memory locations:

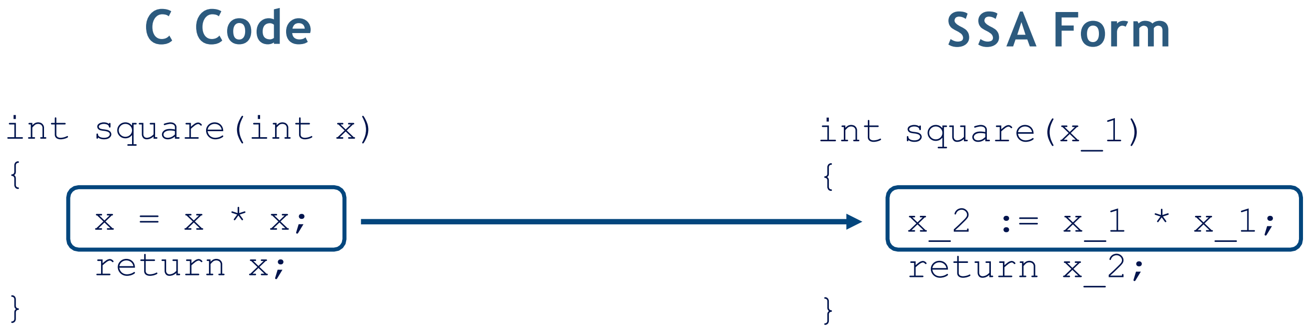
`i32**`	A pointer to a pointer to an integer.
`i32 (i32*) *`	A pointer to a function that takes as argument a pointer to an integer, and returns an integer as result.

More details at https://llvm.org/docs/LangRef.html#type-system

The SSA Form

The Static Single Assignment (SSA) form requires that every variable be defined only **once**, but may be used multiple times.

SSA was proposed in 1988 and an efficient algorithm was developed in IBM, which is still in use in many compilers.



C Code

```
int square(int x)
{
    x = x * x;
    return x;
}
```

SSA Form

```
int square(x_1)
{
    x_2 := x_1 * x_1;
    return x_2;
}
```

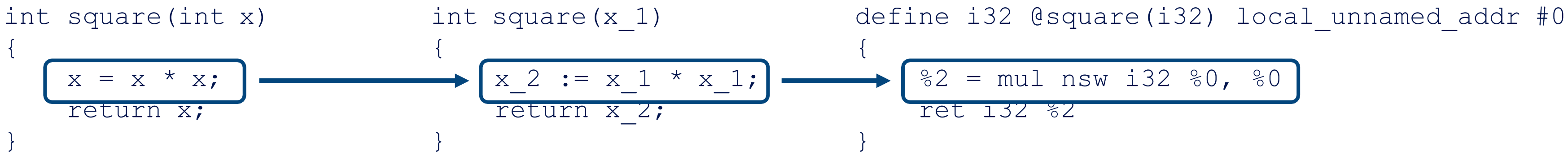
Notice how a new assignment to variable “x” is represented as an assignment to a new variable “x_2”

More about the SSA form: https://en.wikipedia.org/wiki/Static_single_assignment_form

The SSA Form

SSA is commonly used in compilers because it simplifies and improves a variety of compiler optimizations.

LLVM IR makes use of the SSA form.



C Code

```
int square(int x)
{
    x = x * x;
    return x;
}
```

SSA Form

```
int square(x_1)
{
    x_2 := x_1 * x_1;
    return x_2;
}
```

LLVM IR

```
define i32 @square(i32) local_unnamed_addr #0
{
    %2 = mul nsw i32 %0, %0
    ret i32 %2
}
```

Phi Nodes

A problem arises with SSAwhen the same variable is modified in multiple branches.

In the example, to return variable “x”, the SSAform has two choices “x_2” and“x_3” depending on the path taken.

A Phi node abstracts away this complexity by defining anew variable “x_4” which is assigned the value of “x_2” or“x_3”.

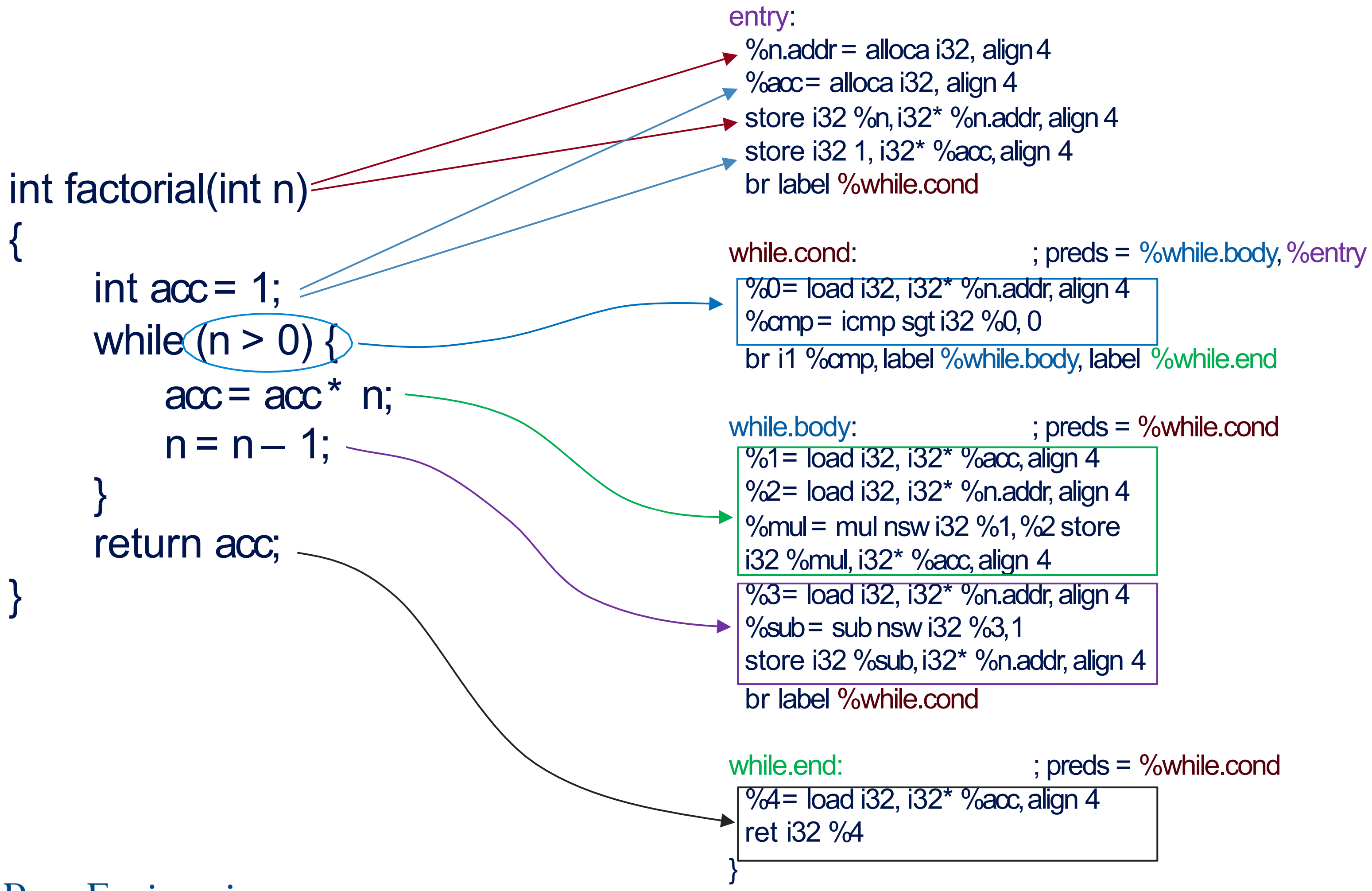
C Code

```
x = 0;
if (y < 1) {
    x++;
} else {
    x--;
}
return x;
```

SSA Code

```
x_1 := 0;
if (y_1 < 1) {
    x_2 := x_1 + 1;
} else {
    x_3 := x_1 - 1;
}
// do I return x_2 or x_3?

x_4 := phi(x_2, x_3);
// return x_4 instead

return x_4;
```

C Program and its LLVM IR Counterpart



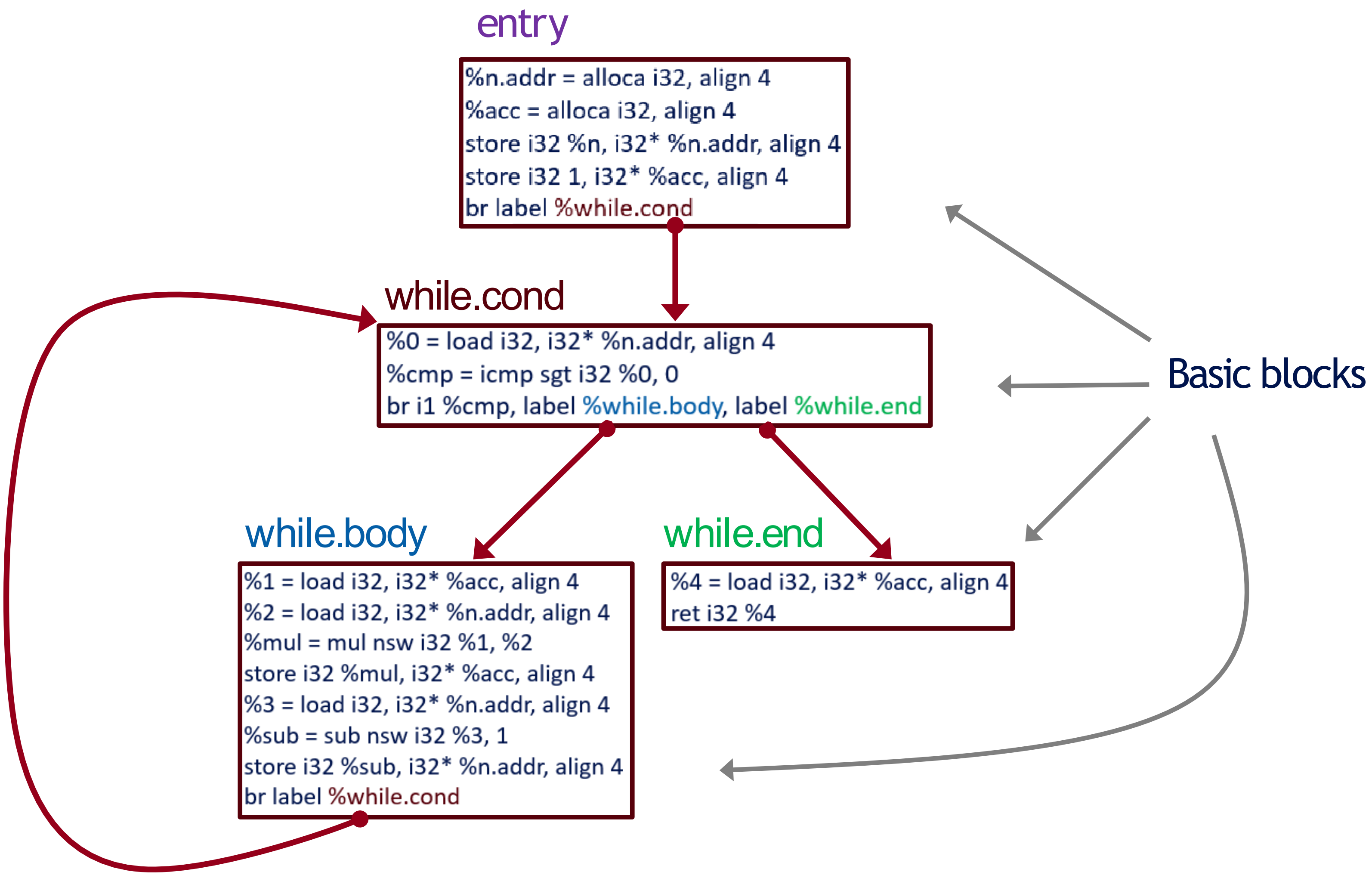
C code: Factorial.c

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

LLVM IR: Factorial.ll

```
entry:
  %n.addr = alloca i32, align 4
  %acc = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  store i32 1, i32* %acc, align 4
  br label %while.cond

while.cond:                        ; preds = %while.body, %entry
  %0 = load i32, i32* %n.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %while.body, label %while.end

while.body:                        ; preds = %while.cond
  %1 = load i32, i32* %acc, align 4
  %2 = load i32, i32* %n.addr, align 4
  %mul = mul nsw i32 %1, %2 store
  i32 %mul, i32* %acc, align 4
  %3 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %3, 1
  store i32 %sub, i32* %n.addr, align 4
  br label %while.cond

while.end:                         ; preds = %while.cond
  %4 = load i32, i32* %acc, align 4
  ret i32 %4
}
```

Basic Blocks & Control Flow Graph (CFG)



entry

```
%n.addr = alloca i32, align 4
%acc = alloca i32, align 4
store i32 %n, i32* %n.addr, align 4
store i32 1, i32* %acc, align 4
br label %while.cond
```

while.cond

```
%0 = load i32, i32* %n.addr, align 4
%cmp = icmp sgt i32 %0, 0
br i1 %cmp, label %while.body, label %while.end
```

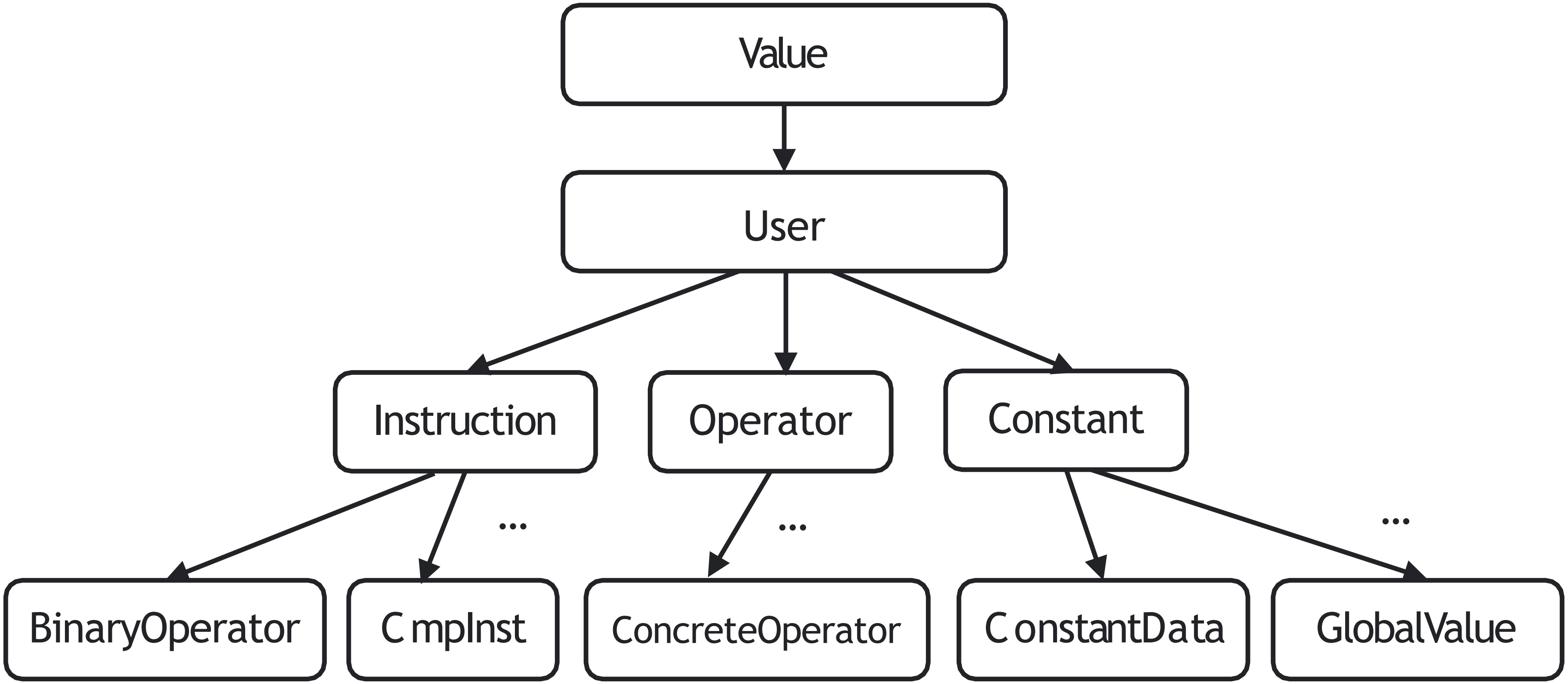
while.body

```
%1 = load i32, i32* %acc, align 4
%2 = load i32, i32* %n.addr, align 4
%mul = mul nsw i32 %1, %2
store i32 %mul, i32* %acc, align 4
%3 = load i32, i32* %n.addr, align 4
%sub = sub nsw i32 %3, 1
store i32 %sub, i32* %n.addr, align 4
br label %while.cond
```

while.end

```
%4 = load i32, i32* %acc, align 4
ret i32 %4
```

LLVM Class Hierarchy



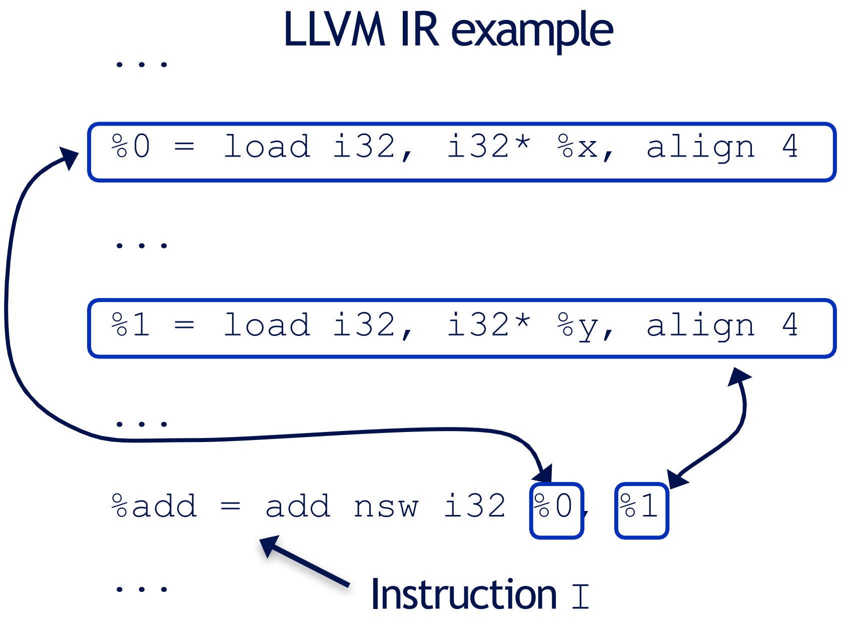
More classes at https://llvm.org/doxygen/classllvm_1_1Value.html

Instructions and Variables

Each Variable <=> the Instruction that assigns to it.

There is a unique instruction assigning to each variable since LLVM IR uses the SSA form.

Thus, each instruction can be viewed as the name of the assigned variable.



LLVM IR example

```
...
%0 = load i32, i32* %x, align 4
...
%1 = load i32, i32* %y, align 4
...
%add = add nsw i32 %0, %1
...
```

```
llvm::outs() << *I.getOperand(0);
```

will not output the operand variable “%0”; it will output the instruction that assigns to it:

“%0 = load i32, i32 %x, align 4”*

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

Printing Information

Use **`outs()`** and **`errs()`** to print instead of using std::cout , std::cerr, and printf. Also, there is no equivalent of std::endl in LLVM.

- Example 1 - printing a function name (Function* F) : ~~std::cout << F->getName().str() << std::endl;~~ outs() <<F->getName() << "\n";

- Example 2 - printing an instruction (Instruction *I): I->dump() or outs() << *I << "\n";

- Example 3 - printing a basic block (BasicBlock* BB): BB->dump() or outs() << *BB << "\n";

Instruction: AllocaInst

An instruction to allocate memory on the stack.

int z; %z= alloca i32, align 4

alloca:	Allocate memory in stack
i32:	Integer of size 32 bits
align:	Memory alignment (4 bytes)

int* z; %z= alloca i32*, align 8

alloca:	Allocate memory in stack
I32*:	Pointer to 32-bit integer
align:	Memory alignment (8 bytes)

More details at https://llvm.org/doxygen/classllvm_1_1AllocaInst.html

Instruction: StoreInst

An instruction for storing to memory.

E.g. store **T** v, **T*** %y
Store value v of type **T** into location pointed to by register %y.
The value may be a constant or a register.

T = i32

```
int x = 5;
```

```
%x = alloca i32, align 4
store i32 5, i32* %x, align 4
```

store: Set value of integer pointed to by register %x to 5

T = i32*

```
int* x = 0;
```

```
%x = alloca i32*, align 8
store i32* null, i32** %x, align 8
```

store: Set value of pointer pointed to by register %x to NULL

More details at https://llvm.org/doxygen/classllvm_1_1StoreInst.html

Instruction: LoadInst

An instruction for reading from memory.

E.g. %x= load **T**, **T*** %y
Load value of type **T** into register %x from location pointed to by register %y.

T = i32

```
int x = …;
… = 1 / x;
```

```
%x = alloca i32, align 4
%1 = load i32, i32* %x
%div = sdiv i32 1, %1
```

load: Load integer value into register %1 from location pointed to by register %x

T = i32*

```
int *x = …;
if (x) …
```

```
%x= alloca i32*, align 8
%1= load i32*, i32** %x
%tobool = icmp ne i32* %1, null
```

load: Load pointer value into register %1 from location pointed to by register %x

More details at https://llvm.org/doxygen/classllvm_1_1LoadInst.html

Instruction: BinaryOperator

An instruction for binary operations.

```
int x = 0;
int y = 2;
z = y + x;
```

Could be +, -, *, /

```
%1 = load i32, i32* %y, align 4
%2 = load i32, i32* %x, align 4
%z = add nsw i32 %1, %2
```

Could be add, sub, mul, udiv, sdiv

add: Store the sum of %1 and %2 in %z
(nsw: no signed wrap)

More details at https://llvm.org/doxygen/classllvm_1_1BinaryOperator.html

Instruction: BinaryOperator operations

- **‘add'** Instruction:The 'add' instruction returns the sum of its two operands.

  <result> = add <ty> <op1>,<op2>

- **'sub'** Instruction:The 'sub' instruction returns the difference of its two operands.

  <result> = sub <ty> <op1>,<op2>

- **'mul'** Instruction:The 'mul' instruction returns the product of its two operands.

  <result> = mul <ty> <op1>, <op2>

- **'udiv'** Instruction:The 'udiv' instruction returns the **unsigned** integer quotient of its two operands.

  <result> = udiv <ty> <op1>, <op2>

- **'sdiv'** Instruction:The 'sdiv' instruction returns the **signed integer** quotient of its two operands.

  <result> = sdiv <ty> <op1>, <op2>

Instruction: ReturnInst

Return avalue (possibly void), from afunction.

```
return void;          ret void

return 0;             ret i32 0
```

More details at https://llvm.org/doxygen/classllvm_1_1ReturnInst.html

Instruction: CmpInst

This instruction returns a bool value or a vector of bool values based on comparison of its two integer, integer vector, or pointer operands.

Type: i1

```
int a= (x==y)
```

```
%cmp = icmp eq i32 %x, %y
```

icmp eq: Compare %x and %y, and set %cmp to 1 if %x is equal to %y, and to 0 otherwise

More details at https://llvm.org/doxygen/classllvm_1_1CmpInst.html

Instruction: CmpInst <cond>

Possible conditions <cond>:

eq:	equal
ne:	not equal
ugt:	unsigned greater than
uge:	unsigned greater or equal
ult:	unsigned less than
ule:	unsigned less or equal
sgt:	signed greater than
sge:	signed greater or equal
slt:	signed less than
sle:	signed less or equal

Instruction: BranchInst

Conditional branch instruction.

```
If (a==0) {
         // br1
        return 0;
} else {
        // br2
        return 1;
}
```

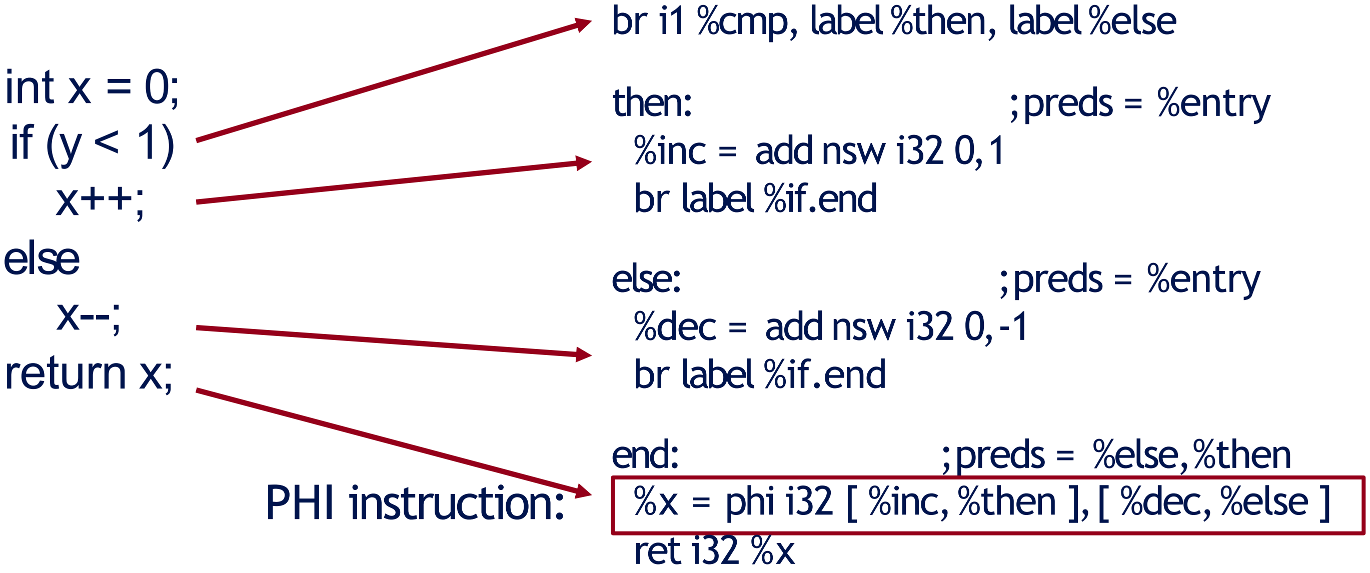
```
%cmp = icmp eq i32 %a, 0
br i1 %cmp, label %IfEqual, label %IfUnequal

IfEqual :
        ret i32 0
IfUnequal :
        ret i32 1
```

br: Determine which branch should be executed;
    jump to %IfEqual if %cmp is true, and to %IfUnequal otherwise

More details at https://llvm.org/doxygen/classllvm_1_1BranchInst.html

Instruction: PHINode

The 'phi' instruction is used to implement the ‘phi’ node in the SSA form.



```
int x = 0;
 if (y < 1)
    x++;
else
    x--;
return x;
```

```
br i1 %cmp, label %then, label %else

then:                          ;preds = %entry
  %inc = add nsw i32 0, 1
  br label %if.end

else:                          ;preds = %entry
  %dec = add nsw i32 0, -1
  br label %if.end

end:                      ;preds = %else, %then
  %x = phi i32 [ %inc, %then ], [ %dec, %else ]
  ret i32 %x
```

PHI instruction:

phi: Assign to %x the value of:

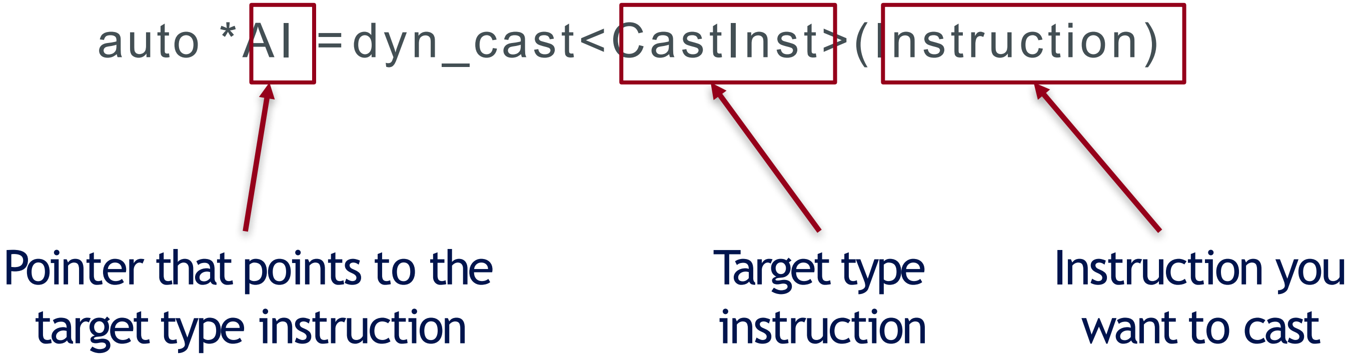
- %inc if predecessor basic block is %then, and
- %dec if predecessor basic block is %else

More details at https://llvm.org/doxygen/classllvm_1_1PHINode.html

Penn Engineering

Checking Instruction Type

A dynamic cast converts an instruction to a more specific type in its class hierarchy at runtime:



```
auto *AI =dyn_cast<CastInst>(Instruction)
```

Pointer that points to the target type instruction

Target type instruction

Instruction you want to cast

If target type is not in original instruction’s class hierarchy, AI will point to NULL. This property can be used to check if an instruction is of a particular type:

```
if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
        //if I can be converted to LoadInst, do something
}
```

Write your own LLVM Pass!

An LLVM pass is created by extending a subclass of the Pass class. We illustrate this for a function pass.

ID is the identifier of the pass class and must be explicitly defined outside the class definition.

runOnFunction will be called for each function in the module. It must return true if it modifies the LLVM IR, and false otherwise.

The RegisterPass class is used to register the pass.The template argument is the name of the pass class and the constructor takes 4 arguments: the name of the command line argument, the name of the pass, a bool if it modifies the CFG, and a bool if it is an analysis pass.

Upon compiling using cmake, a shared static library file “MyAnalysis.so” will be created.

To invoke this pass, run the following command:

```
opt -load MyAnalysis.so -MyAnalysis factorial.ll
```

```
#include "llvm/Pass.h”
#include "llvm/IR/Function.h”

using namespace llvm;

class MyAnalysis: public FunctionPass {
        static char ID;
        MyAnalysis() : FunctionPass(ID) { }
        bool runOnFunction(Function &F);
}

char MyAnalysis::ID = 1;

bool MyAnalysis::runOnFunction(Function &F) {
        // Your function analysis goes here
        return false;
}

static RegisterPass<MyAnalysis> X(
        ”MyAnalysis", ”MyAnalysis", false,false
);
```

More details at https://llvm.org/docs/WritingAnLLVMPass.html

Part III: The LLVM API

The Name of a Module

Class llvm::Module

const StringRef getName( ) const

- Get a short "name" for the module, useful for debugging or logging.

Example:

```
Module M = …
outs() << "Module name is" << M.getName() << "\n";
```

Iterating over Functions in a Module

Class llvm::Module

constiterator_range<iterator> functions( )

- Get an iterator over functions in module.

Example:

```
Module M = …
for (auto &f : M.functions()) {
        // some operations here
}
```

Counting Instructions in a Function

Class llvm::Function

unsigned getInstructionCount( ) const

- Return the number of non-debug IR instructions in this function.
- This is equivalent to the sum of the sizes of all the basic blocks contained in the function.

Example:

```
Module M = …
for (auto &f : M.functions()) { // Get number of instructions in function f
        NumOfFunctions += 1;
        NumOfInstructions += f.getInstructionCount();
}
```

Checking an Instruction’s Kind

Class llvm::Instruction

unsigned getOpcode( ) const

- Return a member of one of the enums, e.g. Instruction::Add.

Example:

```
Instruction instr = …
switch (instr.getOpcode()) {
case Instruction::Br:
        NumOfBranchInstrs += 1;
        break;
}
```

Checking an Instruction’s Kind

Class llvm::Instruction

constbool isBinaryOp( ) const

- Check if the instruction is abinary instruction.

Example:

```
Instruction instr = …
if (instr.isBinaryOp()) {
      NumOfBinaryInstrs += 1;
}
```

Getting an Instruction’s Operands

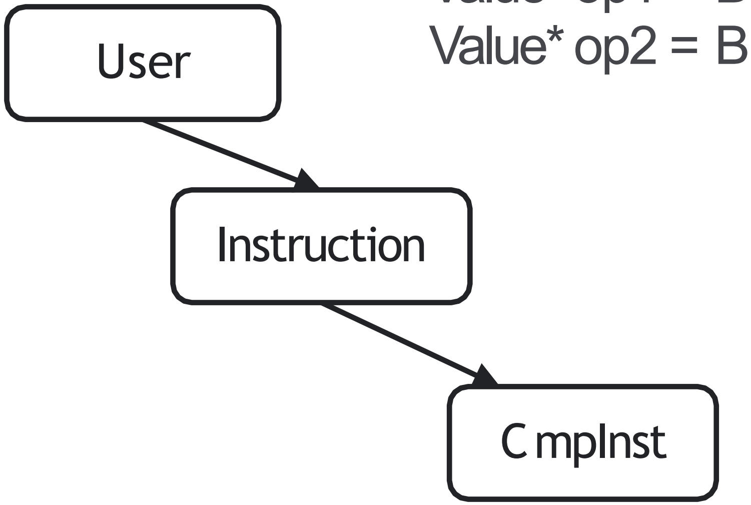
Class llvm::User

Value* llvm::User::getOperand(unsigned i) const

- Return the operand of this instruction, 0 for first operand, 1 for second operand.

Example:

```
BinaryOperator *BO = …
Value* op1 = BO -> getOperand(0);
Value* op2 = BO -> getOperand(1);
```



Also, any public function defined in a super-class can be called by an object of a sub-class

https://llvm.org/doxygen/index.html

Getting an Instruction's Operands

Class llvm::Value

Type* getType( ) const

- All values are typed; get the type of this value.

Example:

```
BinaryOperator *BO = …
Type* t = BO->getOperand(0)->getType();
```

Getting an Operand’s Type

Class llvm::Type

bool isIntegerTy( ) const

- True if this is an instance of IntegerType.

Example:

```
BinaryOperator *BO = …
if (!BO->getOperand(1)->getType()->isIntegerTy())
     return;
```

Evaluating a Conditional Expression

Class llvm::CmpInst

bool llvm::CmpInst::isTrueWhenEqual( ) const
bool llvm::CmpInst::isFalseWhenEqual( ) const

- Determine if this is true/false when both operands are the same (e.g. 0 == 0 TODO).

Example:

```
CastInst *CI = …
if (CI->isTrueWhenEqual()) {
        // some operations
}
if (CI->isFalseWhenEqual()) {
        // some operations
}
```

Store Instruction Operands

Class llvm::StoreInst

Value* getValueOperand( )

- Return 1st operand of store instruction.

Value* getPointerOperand( )

- Return 2nd operand of store instruction.

store i32 0, i32* %x align 4

Example:

```
StoreInst *SI = …
Value* S= SI-> getValueOperand();
// same as Value* S= SI-> getOperand(0);
 Value* S= SI-> getPointerOperand();
// same as Value* S= SI-> getOperand(1);
```

Penn Engineering

Load Instruction Operand

Class llvm::LoadInst

Value* getPointerOperand( )

- Return operand of load instruction.

%1= load i32, i32* %x

Example:

```
LoadInst *LI = …
Value* L = LI -> getPointerOperand();
// same as Value* L = LI -> getOperand(0);
```

Getting the Value of a Constant

Class llvm::Constant

Constant* get(Type* Ty, uint64_t V, bool isSigned = false)

- If Ty is a vector type, return a Constant with a splat of the given value.
- Otherwise return a ConstantInt for the given value.

Example:

```
Type *IntType = …
DebugLoc Debug = …
Value* Line = llvm::ConstantInt::get(IntType, Debug.getLine());
Value* Col   = llvm::ConstantInt::get(IntType, Debug.getCol());
```

Checking if Constant is Zero

Class llvm::Constant

bool isZeroValue( ) const

- Return true if the value is zero or NULL.

Example:

```
Value* V = …
if (ConstantData *CD = dyn_cast<ConstantData>(V))
      return CD->isZeroValue();
```

Getting the Constant Value of PHINode

Class llvm::PHINode

Value* hasConstantValue( ) const

- If the specified PHI node always merges the same value, return the value, otherwise return null.

Example:

```
PHINode *PHI = …
Value* cv = PHI->hasConstantValue();
```

Getting Incoming Values of PHINode

Class llvm::PHINode

unsigned getNumIncomingValues( ) const

- Return the number of incoming values into a PhiNode instruction.

Example:

```
PHINode *PHI = …
unsigned int n = PHI->getNumIncomingValues();
```

Getting an Instruction’s Debug Location

Class llvm::Instruction

const DebugLoc& getDebugLoc( ) const

- Return the debug location of an instruction as a DebugLoc object.

Example:

```
Instruction instr = …
const DebugLoc& Debug = instr.getDebugLoc();
```

Getting a Debug Location's Line

Class llvm::DebugLoc

unsigned getLine( ) const

- Get the line number information from a DebugLoc object.

Example:

```
DebugLoc Debug = …
unsigned DebugLine = Debug.getLine();
```

Getting a Debug Location's Column

Class llvm::DebugLoc

unsigned getCol( ) const

- Get the column number information from a DebugLoc object.

Example:

```
DebugLoc Debug = …
unsigned DebugLine = Debug.getCol();
```

Creating a Function Type

Class llvm::FunctionType

```
FunctionType* FunctionType::get(Type* Result,
                                ArrayRef< Type*> Params,
                                bool isVarArg)
```

- Create a FunctionType with given types of return result and parameters.

Example:

```
LLVMContext Ctx = …
Type *ArgsTypes[] = …
FunctionType* FType = FunctionType::get(
                         Type::getVoidTy(Ctx), ArgsTypes, false);
```

Inserting a Function in a Module

Class llvm::Module

```
FunctionCallee getOrInsertFunction(StringRef Name,
                                   FunctionType* T,
                                   AttributeList AttributeList)
```

- Look up or insert the specified function in the module symbol table.
- Four possibilities: If it does not exist, add a prototype for the function and return it. Otherwise, if the existing function has the correct prototype, return the existing function. Finally, the function exists but has the wrong prototype: return the function with a constantexpr cast to the right prototype. In all cases, the returned value is a FunctionCallee wrapper around the 'FunctionType T' passed in, as well as a 'Value' either of the Function or the bitcast to the function.

Example:

```
Module *M = …
Value* Sanitizer = M->getOrInsertFunction(
                        SanitizerFunctionName, FType);
```

Creating aCall Instruction

Class llvm::CallInst

```
static CallInst* Create(FunctionCallee Func,
                        ArrayRef< Value *> Args,
                        const Twine & NameStr,
                        Instruction * InsertBefore = nullptr)
```

- Create aCallInst object.

Example:

```
Function *Fun = …
std::vector<Value *> Args = …
CallInst *Call = CallInst::Create(Fun, Args, "", &I);
```

Getting Global Information

Class llvm::Value

LLVMContext& getContext( ) const

- Get global information about program including types and constants.

Example:

```
Module* M = …
LLVMContext& Ctx = M->getContext();
```

Getting the Int32 Type

Class llvm::Type

IntegerType* getInt32Ty(LLVMContext& C)

- Get an instance of Int32 type.

Example:

```
LLVMContext Ctx = …
Type* IntType = Type::getInt32Ty(Ctx);
```

Getting the VoidType

Class llvm::Type

Type* getVoidTy(LLVMContext& C)

- Get an instance of void type.

Example:

```
LLVMContext Ctx = …
Type* voidType = Type::getVoidTy(Ctx);
```

Part IV: Navigating the Documentation

Know Your LLVM Version ⚠

The links in this section may yield inaccurate information for uncommon APIs, since they point to the latest LLVM version whereas we use **LLVM 8**.

The LLVM version changes often due to frequent releases; so a naive web search could also produce inaccurate information.

E.g. the return type of llvm::Module::getOrInsertFunction() in different LLVM versions:

LLVM-8.0.0 vs. **LLVM-9.0.0**

```
Constant*
    getOrInsertFunction(
        StringRef Name, Type *RetTy, ArgsTy... Args)
```

```
FunctionCallee
    getOrInsertFunction(
        StringRef Name, Type *RetTy, ArgsTy... Args)
```

LLVM Programmer’s Manual

https://releases.llvm.org/8.0.0/docs/ProgrammersManual.html

A simple and basic way to find what functions you want. Highlights some of the important classes and interfaces available in the LLVM source-base.

Useful content for the labs:

- The isa<>, cast<> and dyn_cast<> templates: A way to convert one class to the desired class.
- The Core LLVM Class Hierarchy Reference: Overview of important functions in each class.
- Helpful Hints for Common Operations: Simple transformations of LLVM code (traversing, creating, etc.).

LLVM Doxygen Sources

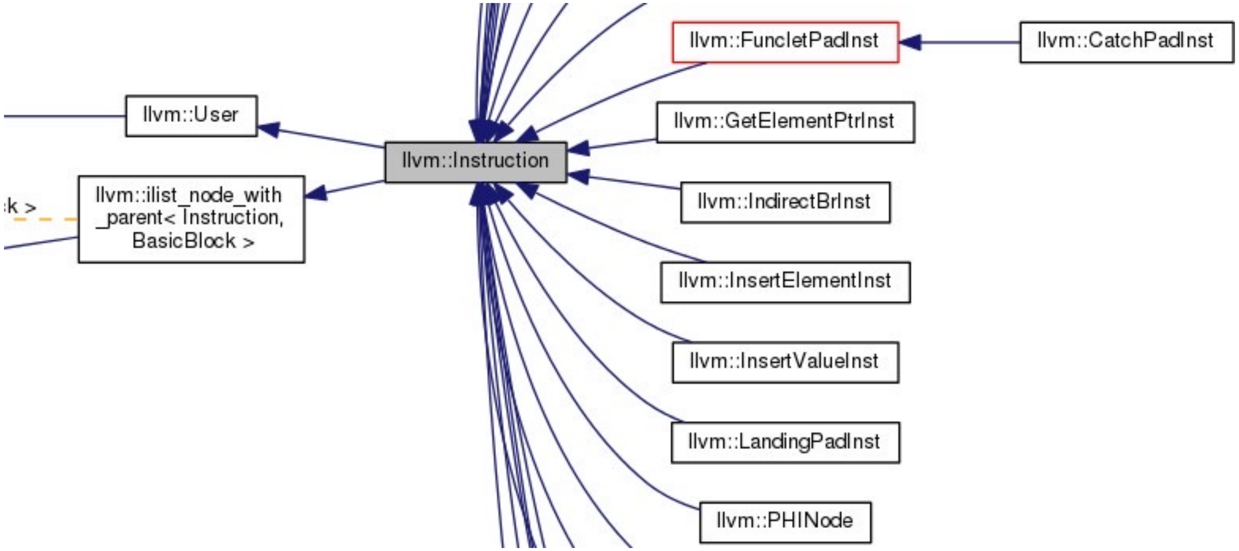
https://llvm.org/doxygen/

Very detailed and complete list guide of LLVM classes and functions.

- Inheritance graph: Relationships between different classes.
- APIs: List of functions for this class; Details and description about those members (arguments, syntax, etc.).
- Source code: Source code (C code) is provided.
- “References” / “Referenced by” sections: Relationship between functions.

LLVM Doxygen Sources

Inheritance graph (example of Instruction Class)



- Go left to find super-classes, go right to find sub-classes

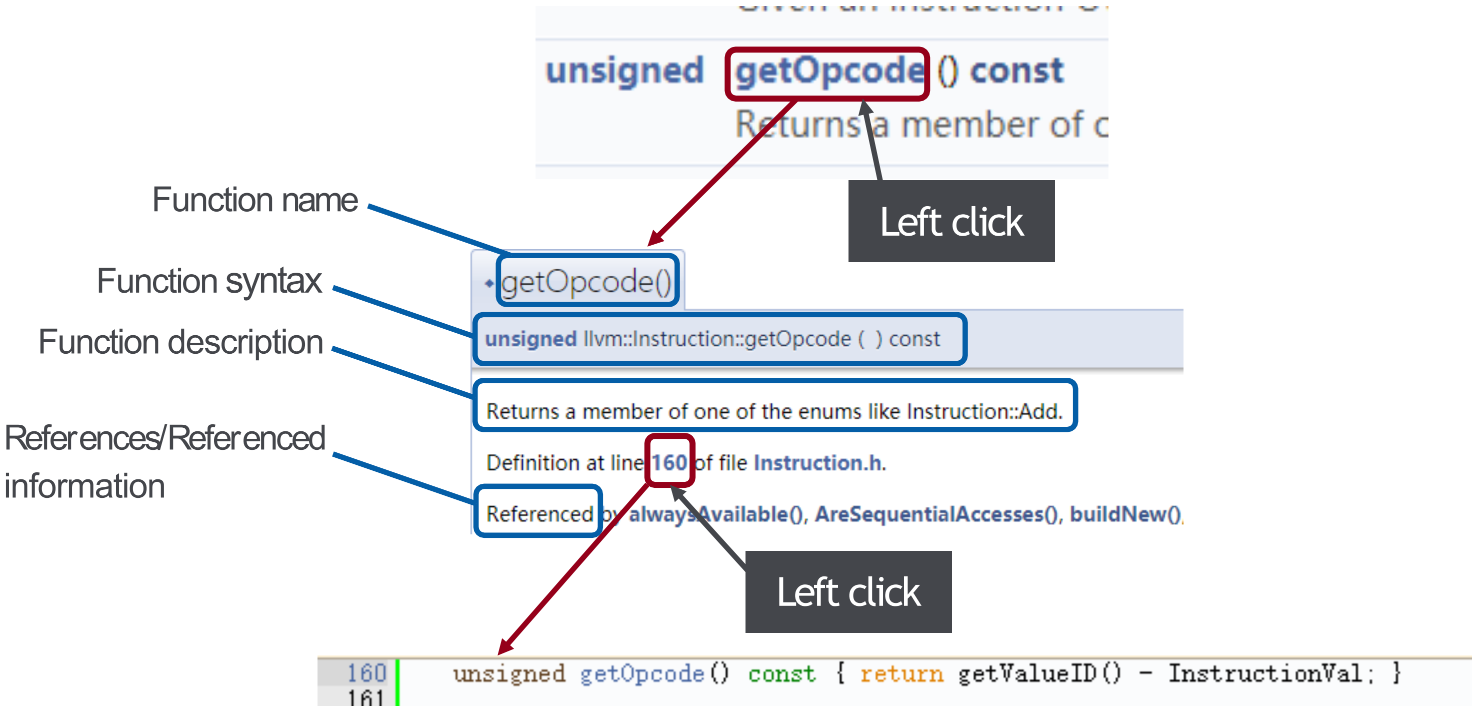
  E.g. User is the super-class of instruction; PHINode is the sub-class of instruction.

- Public function from Left-hand side classes can be used in Right-hand side classes

E.g. Public functions from Instruction class can be used for PHINode objects.

LLVM Doxygen Sources

APIs (example of Instruction Class)



unsigned getOpcode () const
Returns a member of c

getOpcode()

unsigned llvm::Instruction::getOpcode ( ) const

Returns a member of one of the enums like Instruction::Add.

Definition at line 160 of file Instruction.h.

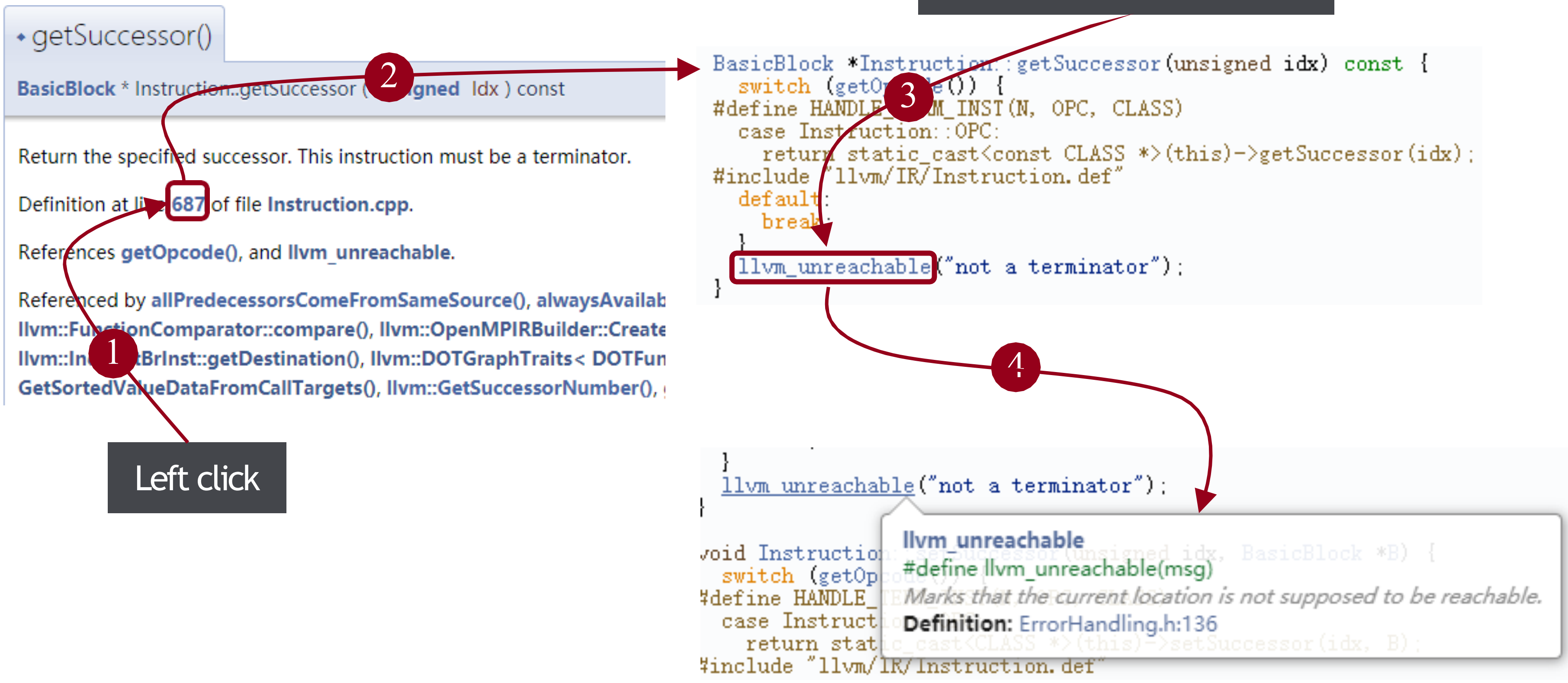
Referenced by alwaysAvailable(), AreSequentialAccesses(), buildNew(),

```
160    unsigned getOpcode() const { return getValueID() - InstructionVal; }
161
```

LLVM Doxygen Sources

Source code (example of Instruction Class)

Hover your cursor on / left click these “blue” links for more information



◆ getSuccessor()

BasicBlock * Instruction::getSuccessor (unsigned Idx) const

Return the specified successor. This instruction must be a terminator.

Definition at line 687 of file Instruction.cpp.

References getOpcode(), and llvm_unreachable.

Referenced by allPredecessorsComeFromSameSource(), alwaysAvailab
llvm::FunctionComparator::compare(), llvm::OpenMPIRBuilder::Create
llvm::IndirectBrInst::getDestination(), llvm::DOTGraphTraits< DOTFun
GetSortedValueDataFromCallTargets(), llvm::GetSuccessorNumber(),

Left click

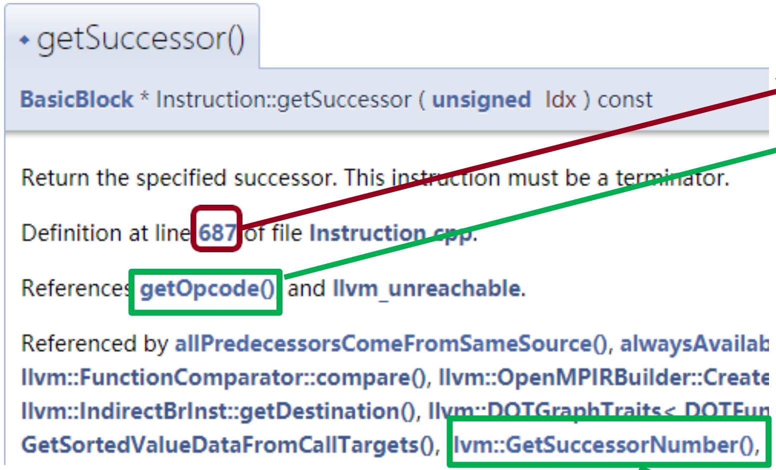
```
BasicBlock *Instruction::getSuccessor(unsigned idx) const {
  switch (getOpcode()) {
#define HANDLE_TERM_INST(N, OPC, CLASS)
  case Instruction::OPC:
    return static_cast<const CLASS *>(this)->getSuccessor(idx);
#include "llvm/IR/Instruction.def"
  default:
    break;
  }
  llvm_unreachable("not a terminator");
}
```

```
  }
  llvm_unreachable("not a terminator");
}

void Instruction::setSuccessor(unsigned idx, BasicBlock *B) {
  switch (getOpcode()) {
#define HANDLE_TERM_INST(N, OPC, CLASS)
  case Instruction::OPC:
    return static_cast<CLASS *>(this)->setSuccessor(idx, B);
#include "llvm/IR/Instruction.def"
```

llvm_unreachable
#define llvm_unreachable(msg)
Marks that the current location is not supposed to be reachable.
Definition: ErrorHandling.h:136

LLVM Doxygen Sources

References/Referenced by sections (example of Instruction Class)



◆ getSuccessor()

BasicBlock * Instruction::getSuccessor (**unsigned** Idx) const

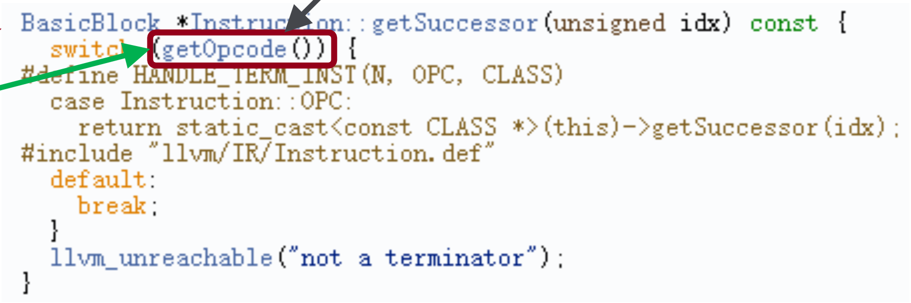
Return the specified successor. This instruction must be a terminator.

Definition at line **687** of file **Instruction.cpp**.

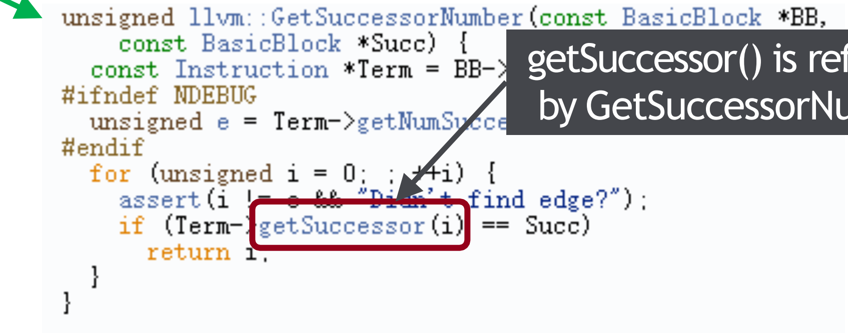
References **getOpcode()** and **llvm_unreachable**.

Referenced by **allPredecessorsComeFromSameSource()**, **alwaysAvailab**
llvm::FunctionComparator::compare(), **llvm::OpenMPIRBuilder::Create**
llvm::IndirectBrInst::getDestination(), **llvm::DOTGraphTraits< DOTFun**
GetSortedValueDataFromCallTargets(), **llvm::GetSuccessorNumber()**,

getSuccessor() references getOpcode() here



```
BasicBlock *Instruction::getSuccessor(unsigned idx) const {
  switch (getOpcode()) {
#define HANDLE_TERM_INST(N, OPC, CLASS)
  case Instruction::OPC:
    return static_cast<const CLASS *>(this)->getSuccessor(idx);
#include "llvm/IR/Instruction.def"
  default:
    break;
  }
  llvm_unreachable("not a terminator");
}
```



```
unsigned llvm::GetSuccessorNumber(const BasicBlock *BB,
    const BasicBlock *Succ) {
  const Instruction *Term = BB->
#ifndef NDEBUG
  unsigned e = Term->getNumSucce
#endif
  for (unsigned i = 0; ; ++i) {
    assert(i != e && "Didn't find edge?");
    if (Term->getSuccessor(i) == Succ)
      return i;
  }
}
```

getSuccessor() is referenced by GetSuccessorNumber()

Penn Engineering

Google / Stack Overflow

Google your question:

- APIs & Classes: Google “llvm+[class/APIs you want to search]” directly. (Normally it will lead you to doxygen documentation)

Use Stack Overflow:

- Search for or ask your question at https://stackoverflow.com/

Further Reading

- Language Frontend with LLVM Tutorial
  https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html
- LLVM Programmer's Manual
  http://llvm.org/docs/ProgrammersManual.html
- LLVM Language Reference Manual
  http://llvm.org/docs/LangRef.html
- Writing an LLVM Pass
  http://llvm.org/docs/WritingAnLLVMPass.html
- LLVM's Analysis and Transform Passes
  http://llvm.org/docs/Passes.html
- LLVM Internal Documentation
  http://llvm.org/docs/doxygen/html/
- LLVM Coding Standards
  http://llvm.org/docs/CodingStandards.html