Modern Compilers

NI(E)-GEN, Spring 2021

https://courses.fit.cvut.cz/NI-GEN

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FIT

Trying to outsmart a compiler defeats much of the purpose of using one.

Brian Kernighan

Modern Compilers

- extremely large systems (millions of LOC), extremely hard to build (hundreds man-years)

- to amortize, most support multiple languages and targets
- frontends and runtimes often dominate the size

- even parsers can get very complex

Most Vexing Parse

```
void f(double my_dbl) {
  int i(int(my_dbl));
}
```

Modern Compilers

- extremely large systems (millions of LOC), extremely hard to build (hundreds man-years)

- to amortize, most support multiple languages and targets
- frontends and runtimes often dominate the size

- even parsers can get very complex, often shared by different compilers


ONE DOES NOT SIMPLY
Write a compiler!
quickmeme.com

There Really are Only Two

- GCC
- LLVM

There Really are Only Two

- depends how you count

- GCC
- LLVM
- MSVC

There Really are Only Two

- really depends how you count

- GCC
- LLVM
- MSVC
- Java, .NET, Rust, Swift, D, Intel C++, Borland C++, Borland Delphi, Fortran,…

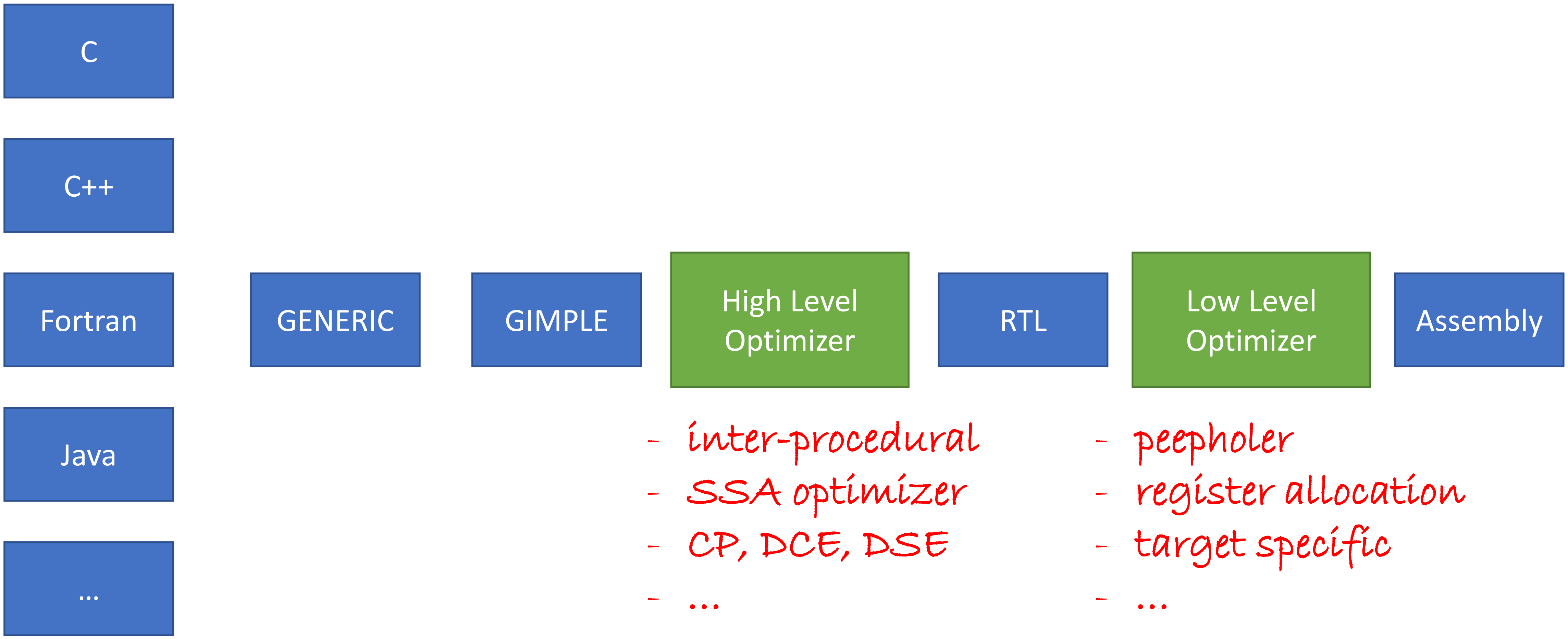
GCC

GNU C Compiler

GCC

- 1987 the first release, only C supported
- 1992 first support for C++
- 2001 uses SSA (LLVM started around this time)
- 2020: the largest set of languages and targets supported (often obscure ones)

GCC Architecture



GENERIC

- very high-level representation, loops, exceptions, complex types, variables
- pretty much like C
- middle-end input

GENERIC

```
if (foo (a + b, c))
    c = b++ / a
endif
return c
```

GIMPLE

- strict subset of GENERIC
- GENERIC code is lowered to GIMPLE by gimplifier
- 3AC

GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0)
    t3 = b
    b = b + 1
    c = t3 / a
endif
return c
```

Low GIMPLE

- removes complex control flow structures and replaces them with explicit jumps

Low GIMPLE

```
    t1 = a +b
    t2 = foo (t1, c)
    if (t2 != 0) <L1,L2>
L1:
    t3 = b
    b = b + 1
    c = t3 / a
    goto L3
L2:
L3:
    return c
```

Low GIMPLE

```
    t1 = a +b
    t2 = foo (t1, c)
    if (t2 != 0) <L1,L2>
L1:
    t3 = b
    b = b + 1
    c = t3 / a
    goto L3
L2:
L3:
    return c
```

Test & Conditional Branch

Empty basic blocks are allowed

RTL

- Register Transfer Language
- assembly for an unlimited register machine
- used for very low level and target specific optimizations
  - register classes, branch instructions, addressing modes, calling conventions
  - no types, but type modes -kind of what register you need
- uses lisp-like notion

RTL

```
// b = a - 1
(set (reg/v:SI 59 [ b ])
    (plus:SI (
        reg/v:SI 60 [ a ]
        (const_int -1 [0xffffffff])
    ))
)
```

LLVM

Low Level Virtual Machine

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, life-long program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp/longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary

Low Level Virtual Machine

- Started around 2000 at University of Illinois at Urbana-Champaign by Chris Lattner, first public release around 2003
- Now maintained by Apple
  - Default compiler for OS X and IOS
- Users & contributors include: Sony, Adobe, Intel, NVIDIA, XMOS, and many others
- Used in various scenarios – including both AOT and JIT compilers, code analysis, etc.

Low Level Virtual Machine

- Permissive license (BSD-style)
- Modern compiler, written in C++

- Under active development

- Modular design
- Well documented
- Language agnostic
  - Does not care about the frontend
  - Many targets available (x86, ARM, PowerPC, etc.)

Low Level Virtual Machine

- Permissive license (BSD-style)
- Modern compiler, written in C++

  - Templates & advanced C++ constructs
  - Indecipherable error messages
  - On a plus side, does not use BOOST

- Under active development
- Modular design
- Well documented
- Language agnostic
  - Does not care about the frontend
  - Many targets available (x86, ARM, PowerPC, etc.)

Low Level Virtual Machine

- Permissive license (BSD-style)
- Modern compiler, written in C++
  - Templates & advanced C++ constructs
  - Indecipherable error messages
  - On a plus side, does not use BOOST
- Under active development
  - Backwards compatibility is frowned upon
- Modular design
- Well documented
- Language agnostic
  - Does not care about the frontend
  - Many targets available (x86, ARM, PowerPC, etc.)

Low Level Virtual Machine

- Permissive license (BSD-style)
- Modern compiler, written in C++
  - Templates & advanced C++ constructs
  - Indecipherable error messages
  - On a plus side, does not use BOOST
- Under active development
  - Backwards compatibility is frowned upon
- Modular design - kind of
- Well documented
- Language agnostic
  - Does not care about the frontend
  - Many targets available (x86, ARM, PowerPC, etc.)

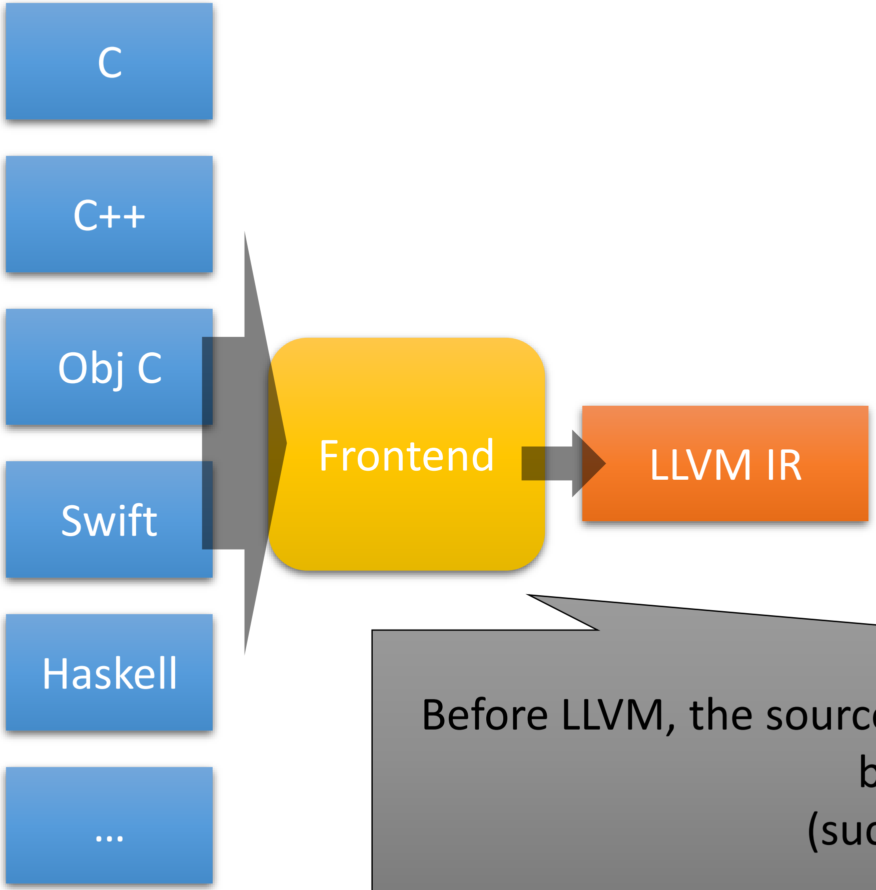
Low Level Virtual Machine

- Permissive license (BSD-style)
- Modern compiler, written in C++
  - Templates & advanced C++ constructs
  - Indecipherable error messages
  - On a plus side, does not use BOOST
- Under active development
  - Backwards compatibility is frowned upon
- Modular design - kind of
- ~~Well~~ documented – just better than others, but the bar is low
- Language agnostic
  - Does not care about the frontend
  - Many targets available (x86, ARM, PowerPC, etc.)

Low Level Virtual Machine

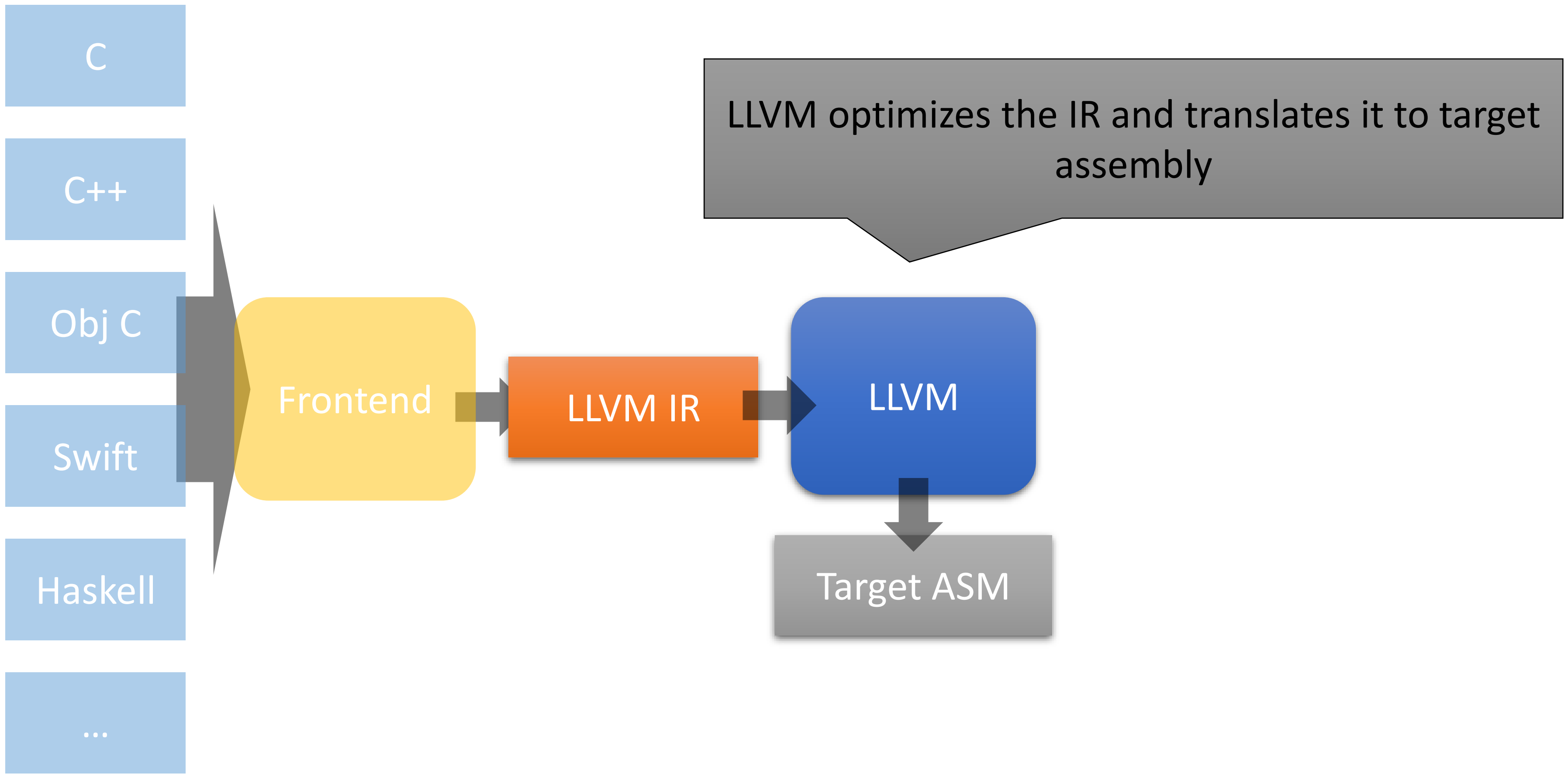
- C
- C++
- Obj C
- Swift
- Haskell
- …

Low Level Virtual Machine

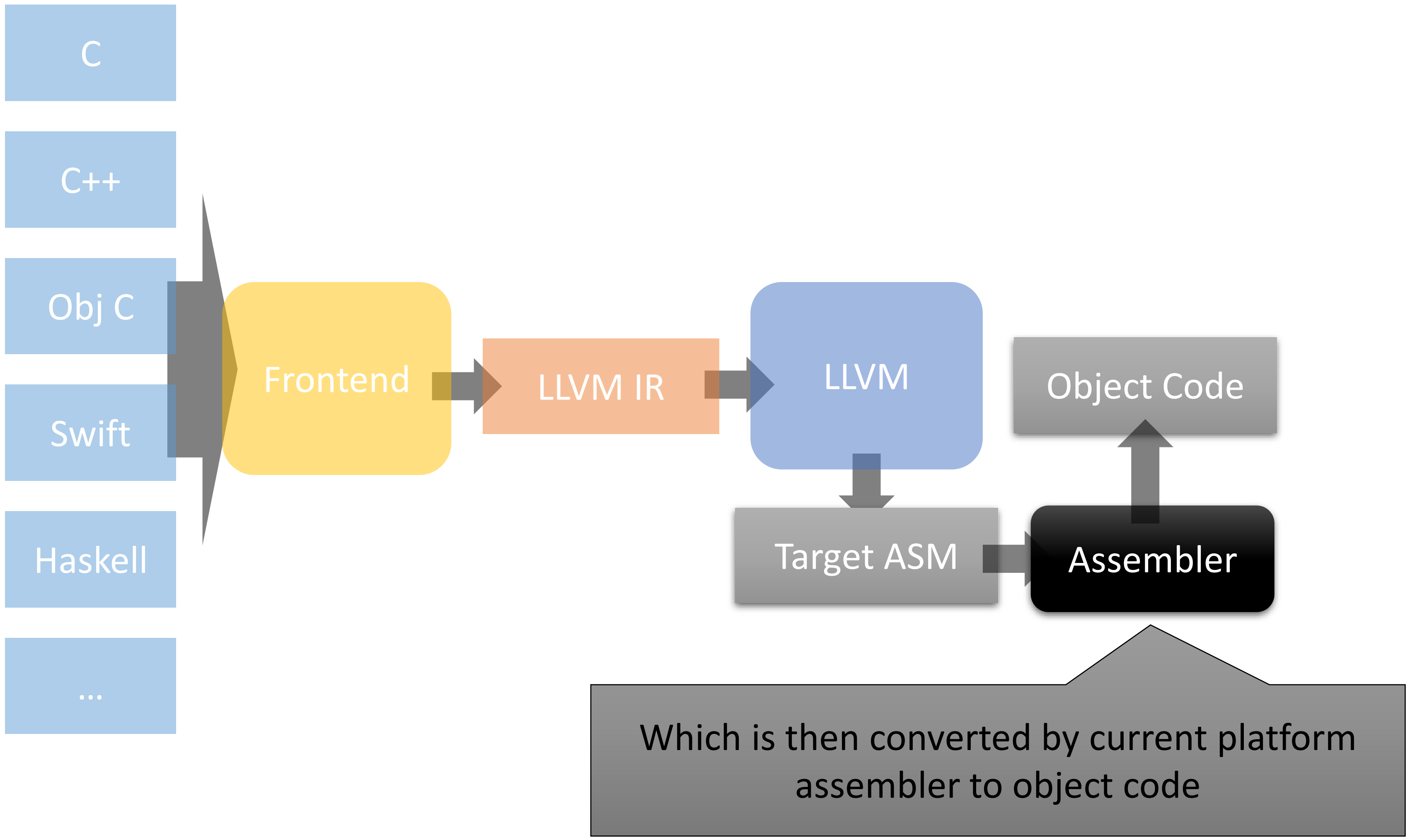


Before LLVM, the source language must be translated to LLVM IR by custom frontend (such as Clang for C/C++)

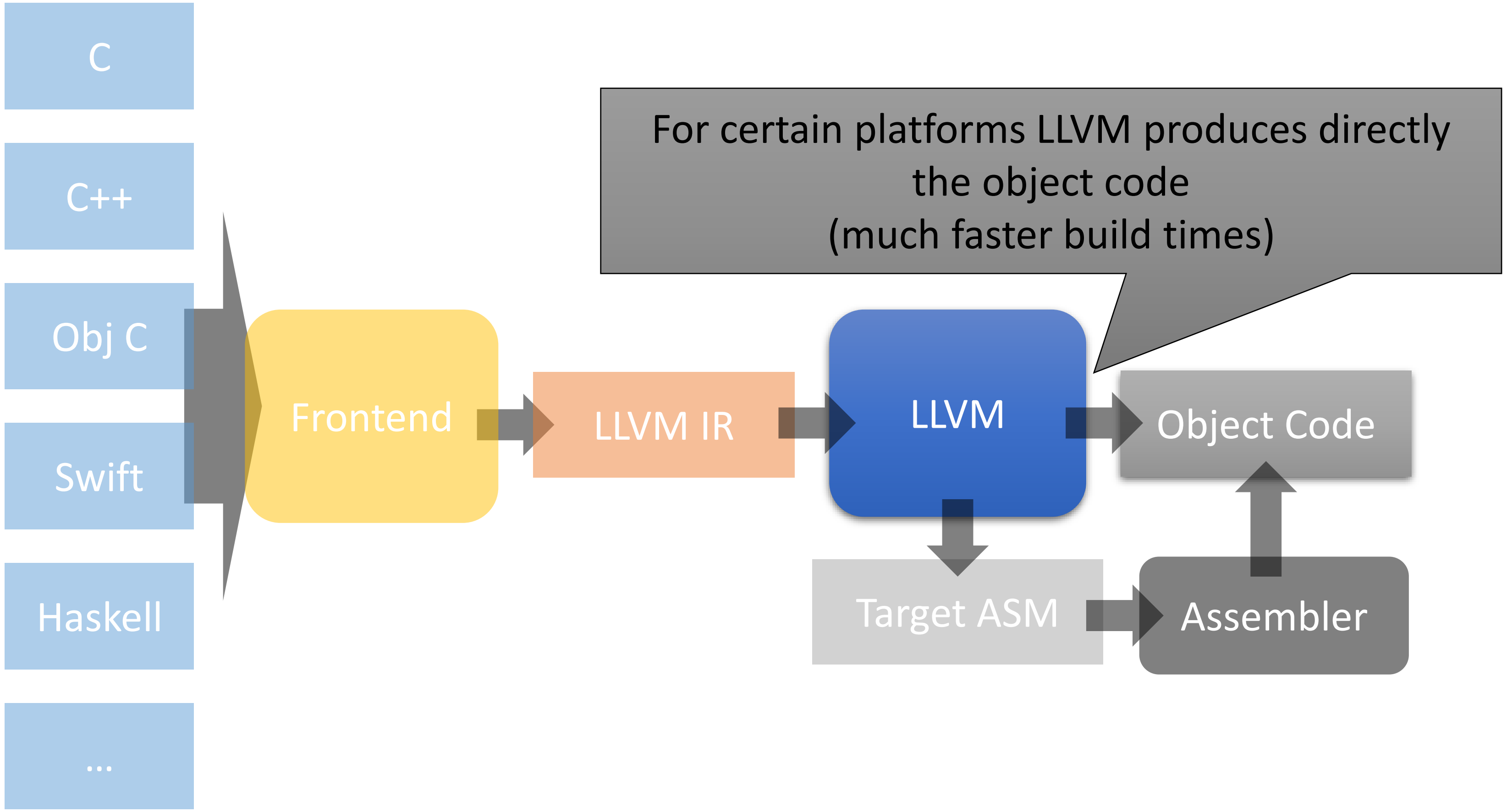
Low Level Virtual Machine



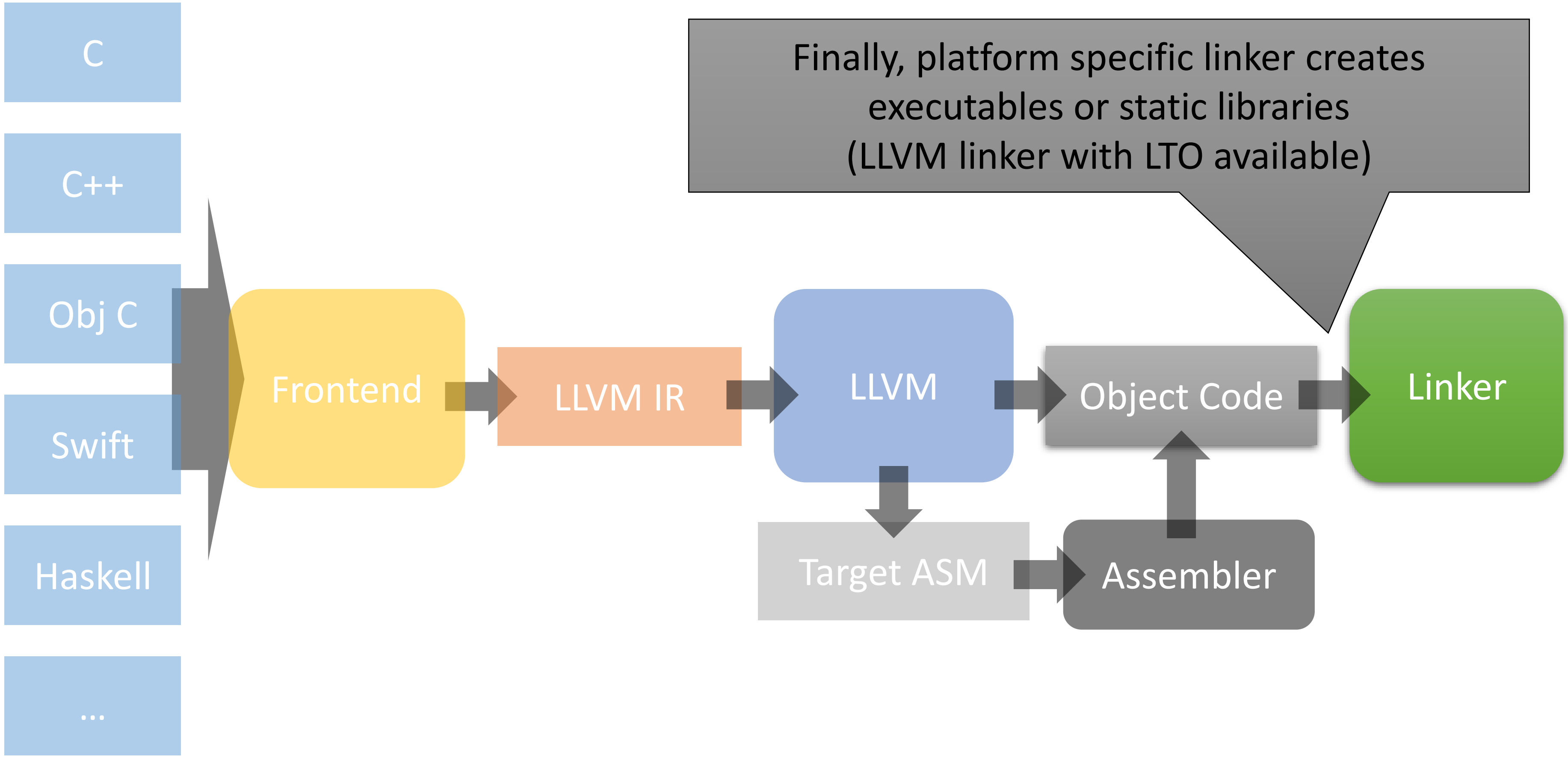
Low Level Virtual Machine



Low Level Virtual Machine



Low Level Virtual Machine

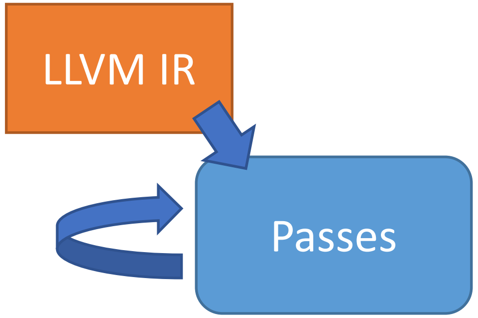


Low Level Virtual Machine

- Does not contain frontends
- Or assemblers in many platforms
  - Using platform assembler instead

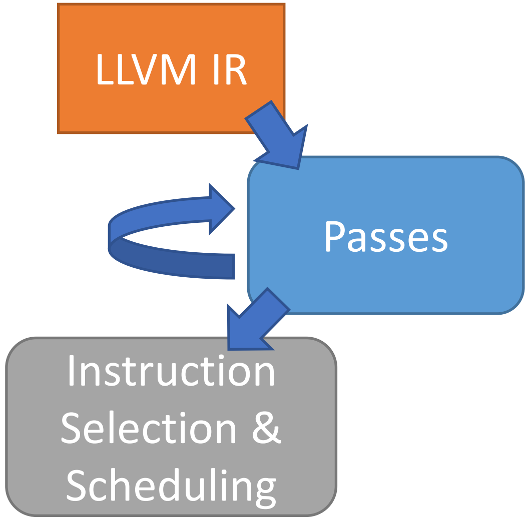
- LLVM’s main focus is on IR optimizations and backend up to assembly
  - SSA IR optimizations
  - Instruction scheduling
  - Register allocation
  - Low level peephole optimizations (to some extent)

LLVM workflow



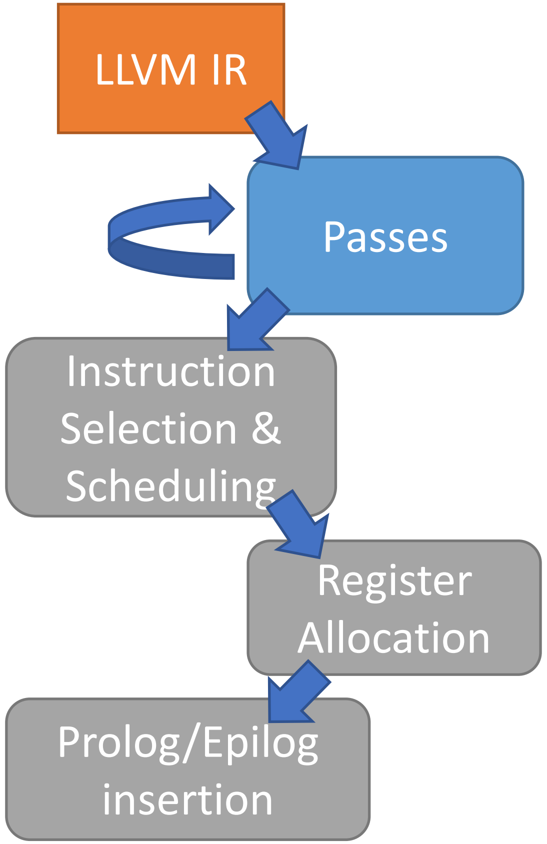
- Several analysis (readonly) and optimization passes are performed over the IR
- Passes can run on
  - Modules
  - Functions
  - Basic blocks
- Passes can
  - Depend on other passes
  - Preserve other passes
- LLVM provides a scheduler that calls the passes in a most effective way

LLVM workflow



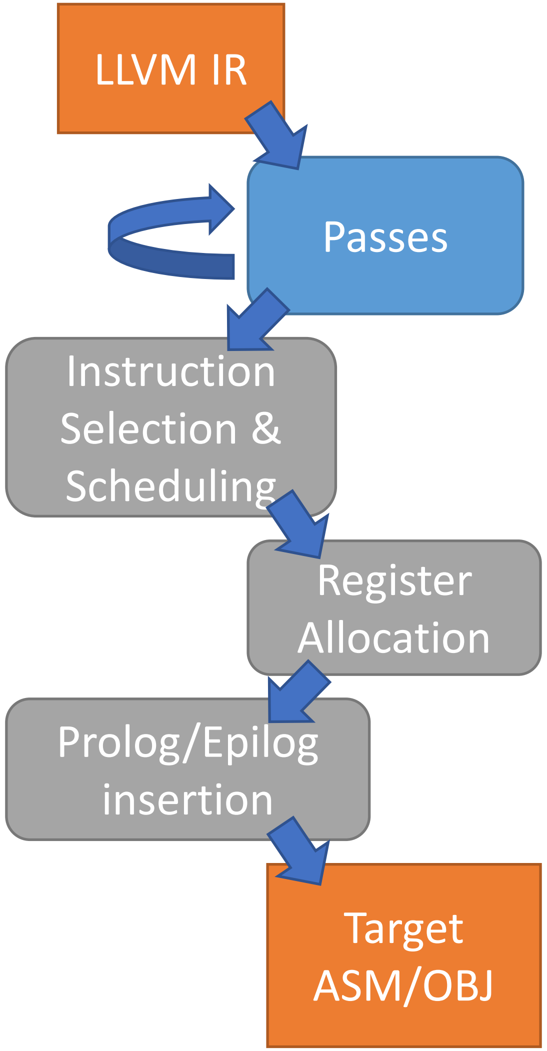
- When IR passes are finished, LLVM enters the target specific backend
- Target specific instructions are selected and attached to the IR
- Afterwards, target specific SSA passes may execute

LLVM workflow



- After SSA target passes, register allocator is executed
  - Linear scan
  - Greedy
  - Region-based
  - GRA
- Function Prologues and epilogues are inserted
- After these steps, late target optimizations are usually executed
  - peepholer

LLVM workflow



- Finally IR is dumped as target object, or assembly files

Bitcode organization

- Code in LLVM is organized into modules (akin to compilation units)

- Each module contains global variables and functions
  - Unique names
  - Linkage specification
  - Calling convention
  - Visibility,
  - Etc.

- Inside functions, code is organized into basic blocks

Types

- void (no value, no size)
- integer (variable size in bits)
  - i1, i32, i67837
  - integers do not care about their sign
- Floats
  - half, float, double, …
- Arrays (of same element types, fixed size)
  - [10 x i32], [10 x [10 x i8]]
- Structures
  - { i32, i32, float, i1 }
  - <{ i8, i8, i32 }> packed structure (padding=0, align=1byte)
- Functions
  - i32 (i32, i32)
- Pointers (*)

+ fp128
+ fp80 (x86)
+ ppc 128bit fp
+ mmx types
+ vectors (SIMD)
+ labels, tokens, metadata, …

Variables

- LLVM provides three types of variables:

- Stack allocated variables
  - `alloca` = creates variable on stack and returns a pointer to it
  - `load` & `store`

- Global variables (prefixed with @)
  - Contain pointers to the global variables (i.e. similar to stack allocated variables)

- Local variables (prefixed with %)
  - Result of each instruction goes into new local variable
  - In fact, the variable and the instruction are the same thing in LLVM
  - This makes the LLVM variables SSA values

Bitcode Instructions

- Terminator instructions
  - Terminate basic block (jumps, returns, exceptions)
- Binary Operations
  - Different opcodes for floats (fadd,…) and integers (add…)
- Shifts, rotations
  - arithmetic, logic
- Vector operations
  - Extract, insert, shuffle
- Memory access & addressing
  - allocation, load, store, fences, getelementptr
- Conversions
  - Zext, sext, …
- Calling
- Comparisons
  - Icmp, fcmp, …
- Exceptions
  - Catchpads, landingpads, etc.
- Virtuals (PHI node)
- Intrinsics (gc, padding, etc.)

Bitcode Instructions

- All instructions are strongly typed and types are almost always explicit in the IR
- Each instruction can be named, in which case the variable bearing the result of the instruction will carry the name

```
%2 = call i32 @min(i32 1,%1)
```

Bitcode Instructions

- All instructions are strongly typed and types are almost always explicit in the IR
- Each instruction can be named, in which case the variable bearing the result of the instruction will carry the name

```
%2 = call i32 @min(i32 1,%1)
```

Name of the variable holding result of the instruction (if no name is provided, llvm assigns unique number)

Bitcode Instructions

- All instructions are strongly typed and types are almost always explicit in the IR
- Each instruction can be named, in which case the variable bearing the result of the instruction will carry the name

Llvm instruction (function call)

```
%2 = call i32 @min(i32 1,%1)
```

Bitcode Instructions

- All instructions are strongly typed and types are almost always explicit in the IR
- Each instruction can be named, in which case the variable bearing the result of the instruction will carry the name

```
%2 = call i32 @min(i32 1,%1)
```

Type of the result of the instruction (explicit)

Bitcode Instructions

- All instructions are strongly typed and types are almost always explicit in the IR
- Each instruction can be named, in which case the variable bearing the result of the instruction will carry the name

Value of argument 1 (constant 1)

```
%2 = call i32 @min(i32 1,%1)
```

Value of argument 2 (variable 1)

Terminators

- `ret`
  - return from a function, may or may not return a value

    ```
    ret i32 %3
    ```

- `br`
  - Unconditional

    ```
    br label %4
    ```

  - Conditional (branches to two basic blocks based on condition)

    ```
    br i1 %2, label %3, label %4
    ```

- Also indirect branch, switch, exceptions throwing & catching

Binary Operators

- `add, sub, mul`
  - Do not care about signedness of operands

```
%1 = add i32 %a, %b
```

- `udiv, sdiv`
  - Unsigned and signed division of integers

- `fadd, fsub, fmul, fdiv`
  - Floating point arithmetics

```
%3 = fadd double %a, %b
```

Memory

- `alloca`
  - allocates space for given type on stack and returns a pointer to it

    ```
    %1 = alloca i32, align 4
    ```

- `load`
  - loads contents of given pointer to register

    ```
    %2 = load i32 i32* %1, align 4
    ```

- `store`
  - stores to a pointer

    ```
    store i32 %I, i32* %1, align 4
    ```

Memory

- `getelementptr`
  - address calculation for subelements of aggregate types (array, structure)
  - different semantics for arrays and structs
  - very important and often misunderstood instruction

```
%A = type { i32, double }
# type of %1 is %A*
# A*[3].double becomes
%3 = getelementptr %A, %A* %1, i32 3
%4 = getelementptr %A, %A* %3, i32 0, i32 1
```

Memory

- `getelementptr`
  - address calculation for subelements of aggregate types (array, structure)
  - different semantics for arrays and structs
  - very important and often misunderstood instruction

For arrays, index of element we want to access

Pointer to the aggregate type

```
%A = type { i32, double }
# t
# A*[3].double becomes
%3 = getelementptr %A, %A* %1, i32 3
%4 = getelementptr %A, %A* %3, i32 0, i32 1
```

Type we are operating on

Memory

- `getelementptr`
  - address calculation for subelements of aggregate types (array, structure)
  - different semantics for arrays and structs
  - very important and often misunderstood instruction

```
%A = type { i32, double }
# type of %1 is %A*
#
%3 = getelementptr %A, %A* %1, i32 3
%4 = getelementptr %A, %A* %3, i32 0, i32 1
```

For structs, index of element in declaration

Pointer to the aggregate type

Type we are operating on

Memory

- `getelementptr`
  - address calculation for subelements of aggregate types (array, structure)
  - different semantics for arrays and structs
  - very important and often misunderstood instruction

```
%A = type { i32, double }
# type of %1 is %A*
#
%3 = getelementptr %A, %A* %1, i32 3
%4 = getelementptr %A, %A* %3, i32 0, i32 1
```

Pointer to the aggregate type

???

For structs, index of element in declaration

Type we are operating on

Bitcode Representations

- human readable LLVM (this is what we have seen so far)

```
%1 = add i32 %a, %b
```

- binary bitcode (this is what LLVM toolchain members usually pass)

```
0x0 f2 cd 0a 0b 54 5a 35 12 7e 2f
```

- C++ API (this is what frontend developers use to construct the IR

```
auto b = llvm::BasicBlock::Create(llvm::GetGlobalContext(), “”, f);
auto li = new llvm::LoadInst(ptr, “”, false, b);
llvm::ReturnInst::Create(llvm::getGlobalContext(), li, b);
```

LLVM IR C++ API

Everything is a class, something is a pointer

- Modules, Functions and BasicBlocks
  - `llvm::Module`, `llvm::Function`, `llvm::BasicBlock`
- Types, values and instructions are represented by classes
  - `llvm::Type`, `llvm::Value` and `llvm::Instruction`
- Usually pointers to the classes are expected
- Many (but not all) of the classes should not be created using constructors, but provide static `Create` methods

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

```
auto tv=llvm::Type::getVoidTy(llvm::getGlobalContext())
auto ti=llvm::IntegerType::get(context, 32)
auto td=llvm::Type::getDoubleTy(context)
auto ts=llvm::StructType::create(context, “name”)
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

Often, context is required (in case of multiple llvm instances, we can always use global context for single instance)

```
auto tv=llvm::Type::getVoidTy(llvm::getGlobalContext())
auto ti=llvm::IntegerType::get(context, 32)
auto td=llvm::Type::getDoubleTy(context)
auto ts=llvm::StructType::create(context, “name”)
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

C++ void

```
auto tv=llvm::Type::getVoidTy(llvm::getGlobalContext())
auto ti=llvm::IntegerType::get(context, 32)
auto td=llvm::Type::getDoubleTy(context)
auto ts=llvm::StructType::create(context, “name”)
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

C++ int or unsigned

distinguished by instructions

```
auto tv=llvm::Type::getVoidTy(llvm::getGlobalContext())
auto ti=llvm::IntegerType::get(context, 32)
auto td=llvm::Type::getDoubleTy(context)
auto ts=llvm::StructType::create(context, “name”)
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

C++ double

```
auto t[illegible]getVoidTy(llvm::getGlobalContext())
auto ti=llvm::I[illegible]egerType::get(context, 32)
auto td=llvm::Type::getDoubleTy(context)
auto ts=llvm::StructType::create(context, “name”)
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

Types

- Each type must be specified and have corresponding class
- Types are common to all modules in llvm

```
auto tv=llvm::Type::getVoidTy(llvm::getGlobalContext())
:Integer context, 32)
auto td=ll m::Type::get oubleTy(context)
auto ts=llvm::StructTyp e::create(context, "name")
std::vector<llvm::Type *> fields = { ti, ti, td }
ts->setBody(fields, false);
```

struct { int a; int b; double c; }

Packed?

Creating constants

- constants inherit from llvm::Value and can be used as arguments to instructions
- as types, constants live outside modules

```
llvm::ConstantInt::get(context, llvm::APInt(32, 1, false));
```

```
llvm::ConstantFP::get(context, llvm::APFloat(3.14))
```

Creating constants

- constants inherit from llvm::Value and can be used as arguments to instructions
- as types, constants live outside modules

size

```
llvm::ConstantInt::get(context, llvm::APInt(32, 1,
false))
```

value

signed?

```
llvm::ConstantFP::get(context, llvm::APFloat(3.14))
```

Creating a function

```
int min(int i, int j) {
   return i < j ? i : j;
}
```

Creating a function

```
auto m = llvm::Module::Create("name", context);

auto ft = llvm::FunctionType::get(ti, { ti, ti }, false);

auto f = llvm::Function::Create(ft,
llvm::GlobalValue::ExternalLinkage, "min", m);

f->setCallingConvention(llvm::CallingConv::C);
```

Creating a function

First create a module with given name

```
auto m = llvm::Module::Create("name", context);
```

varargs?

```
auto ft = llvm::FunctionType::get(ti, { ti, ti }, false);
```

The function must have a type, in our case
int (*ptr)(int, int)

```
auto f = llvm::Function::Create(ft,
llvm::GlobalValue::ExternalLinkage, "min", m);

f->setCallingConvention(llvm::CallingConv::C);
```

Creating a function

```
auto m = llvm::Module::Create("name", context);

auto ft = llvm::FunctionType::get(ti, {ti, ti}, false);

auto f = llvm::Function::Create(ft,
llvm::GlobalValue::ExternalLinkage, "min", m);

f->setCallingConvention(llvm::CallingConv::C);
```

type

Create the function object

Module the function belongs to

symbol visibility

name of the function (unique)

sets C calling convention for the function

At this point we have created a proper function declaration in LLVM IR. The function can be called in the module, but it does not have any code in it.

```
auto m = llvm::Module::Create("name", context);

auto ft = llvm::FunctionType::get(ti, { ti, ti }, false);

auto f = llvm::Function::Create(ft,
llvm::GlobalValue::ExternalLinkage, "min", m);

f->setCallingConvention(llvm::CallingConv::C);
```

Creating code

```
auto args = f->arg_begin();
llvm::Value * first = &* args++;
llvm::Value * second = &* args;

auto b = llvm::BasicBlock::Create(context, “first”, f);
auto cmp = new llvm::ICmpInst(*b, llvm::ICmpInst::ICMP_SLT, first,
second);

auto lt = llvm::BasicBlock::Create(context, “lt”, f);
auto gte = llvm::BasicBlock::Create(context, “gte”, f);

llvm::BranchInst::Create(lt, gte, cmp, b);
llvm::ReturnInst::Create(context, lt, first);
llvm::ReturnInst::Create(context, gte, second);
```

Creating

Create shorthand values for function arguments

```
auto args = f->arg_begin();
llvm::Value * first = &* args++;
llvm::Value * second = &* args;
```

Create the first basic block

```
auto b = llvm::BasicBlock::Create(context, “first”, f);
auto cmp = new llvm::ICmpInst(*b, llvm::ICmpInst::ICMP_SLT, first,
second);
```

Insert signed less than comparison of the arguments at the end of the basic block

Create basic blocks for i < j and and i >= j cases

```
auto lt = llvm::BasicBlock::Create(context, “lt”, f);
auto gte = llvm::BasicBlock::Create(context, “gte”, f);
```

Conditional branch based on the result of the comparison to either lt, or gte basic blocks

```
llvm::BranchInst::Create(lt, gte, cmp, b);
llvm::ReturnInst::Create(context, lt, first);
llvm::ReturnInst::Create(context, gte, second);
```

Return the respective values

Creating code

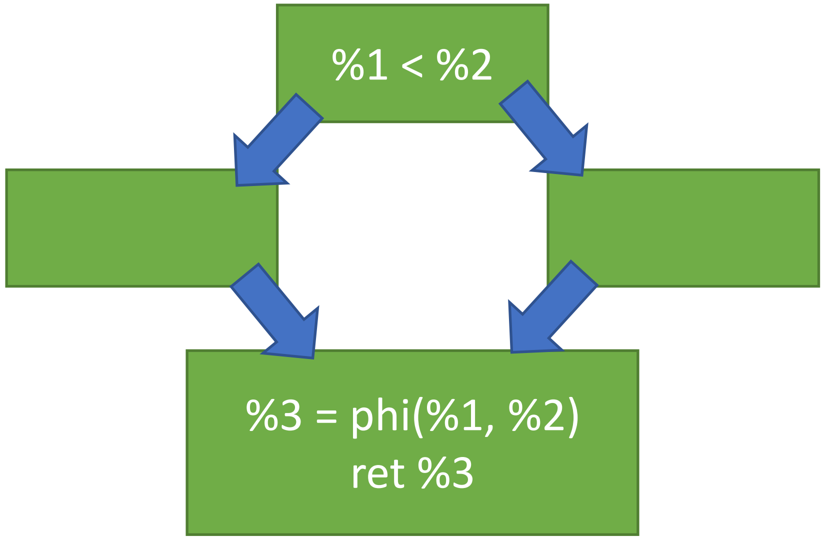
```
define i32 @min(i32 %1, i32 %2) {
    %3 = icmp slt i32 %1, %2
    br i1 %3, label %4, label %5

; <label>:4
    ret i32 %1

; <label>:5
    ret i32 %2
}
```

Working with PHI nodes

- In the min function, we can create only one exit point from the function
- But it would require us to use a PHI node as we would return either first, or second argument in the last basic block based on where we arrived from



Working with PHI nodes

```
define i32 @min(i32 %1, i32 %2) {
    %3 = icmp slt i32 %1, %2
    br i1 %3, label %4, label %5

; <label>:4
    br label %6
; <label>:5
    br label %6
; <label>:6
    %7 = phi i32 [ %1 %4 ], [ %2 %5 ]
    ret i32 %7
}
```

Working with PHI nodes

```
define i32 @min(i32 %1, i32 %2) {
    %3 = icmp slt i32 %1, %2
    br i1 %3, label %4, label %5
; <label>:4
    br label %6
; <label>:5
    br label %6
; <label>:6
    %7 = phi i32 [ %1 %4 ], [ %2 %5 ]
    ret i32 %7
}
```

If arriving from block %4, value will be %1

If arriving from block %5, value will be %2

Working with PHI nodes

```
auto args = f->arg_begin();
llvm::Value * first = args++;
llvm::Value * second = args;

auto b = llvm::BasicBlock::Create(context, “first”, f);
auto cmp = new llvm::ICmpInst(*b, llvm::ICmpInst::ICMP_SLT, first,
second);

auto lt = llvm::BasicBlock::Create(context, “lt”, f);
auto gte = llvm::BasicBlock::Create(context, “gte”, f);
auto end = llvm::BasicBlock::Create(context, “end”, f);

llvm::BranchInst::Create(lt, gte, cmp, b);
llvm::BranchInst::Create(end, lt);
llvm::BranchInst::Create(end, gte);

auto phi = llvm::PHINode::Create(ti, 2, “”, end);
phi->addIncomming(first, lt);
phi->addIncomming(second, gte);

llvm::ReturnInst::Create(context, end, phi);
```

Working with PHI nodes

```
auto args = f->arg_begin();
llvm::Value * first = args++;
llvm::Value * second = args;

auto b = llvm::BasicBlock::Create(context, “first”, f);
auto cmp = new llvm::ICmpInst(*b, llvm::ICmpInst::ICMP_SLT, first,
second);

auto lt = llv
auto gte = ll
auto end = llvm::Basi  ::Create(context, “end”, f);

llvm::BranchInst::Create(lt, gte, cmp, b);
llvm::BranchInst::Create(end, lt);
llvm::BranchInst::Create(end, gte);

auto phi = llvm::PHINode::Create(ti, 2, “”, end);
phi->addIncomming(first, lt);
phi->addIncomming(second, gte);

llvm::ReturnInst::Create(context, end, phi);
```

Create the ending basic block and jumps from lt and gte blocks to it (they will be empty)

Create phi node in the last basic block, reserve 2 incoming edges

Add incoming edges

Further Reading

https://llvm.org/docs/tutorial/index.html – LLVM tutorial

ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf – GCC's GENERIC and GIMPLE information