CSE 703 Seminar:
Advanced Software Security - Techniques and Tools

Instructor: Dr. Ziming Zhao

Location: Online
Time: Monday, 12:50 PM-2:55 PM

Announcements

- Presentation schedule
- Course projects
  - MITRE eCTF: Xi Tan, Gursimran Singh, Md Armanuzzaman, Malav Vyas, Ariel Shevah, Anjie Sun
  - Choose your own project-1: Jacquelyn Dufresne
  - Choose your own project-2: Charles Wiechec

Disclaimer

Many slides of this class were stolen from http://www.mshah.io/

http://www.mshah.io/LLVM/NortheasternMITIntroduction%20to%20LLVM.pdf

http://www.mshah.io/LLVM/NortheasternMITIntroductiontoClang.pdf

Agenda

- LLVM History
- LLVM IR
- Write LLVM Passes

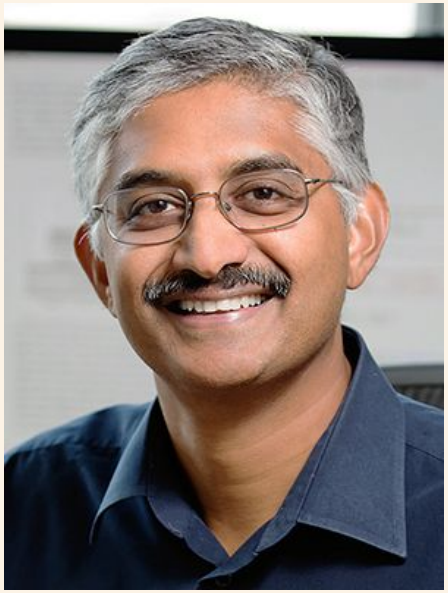
What is LLVM?

An open source framework for building tools
• Tools are created by linking together various libraries provided by the LLVM project and your own

An extensible, strongly typed intermediate representation, i.e. LLVM IR
• https://llvm.org/docs/LangRef.html

An industrial strength C/C++ optimizing compiler
• Which you might know as clang/clang++ but these are really just drivers that invoke different parts (libraries) of LLVM

History of LLVM



Vikram Adve

Chris Lattner

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, lifelong 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

mizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at run-time, 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

2004 International Symposium on Code Generation and Optimization

LLVM

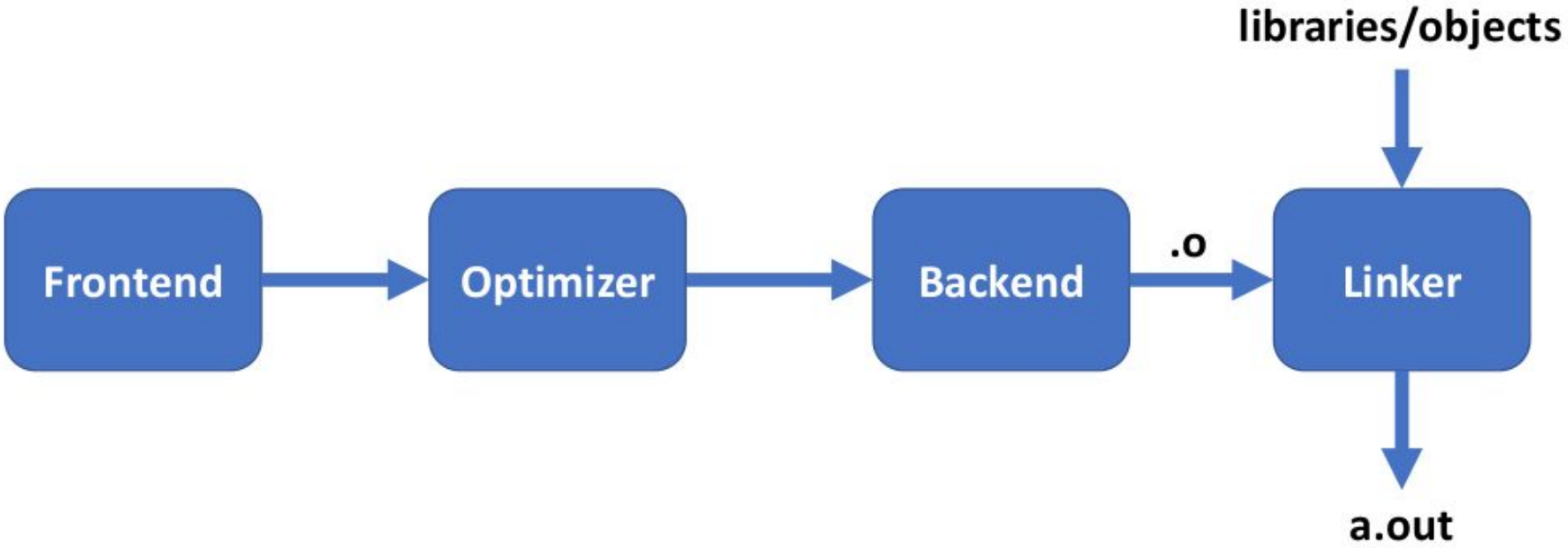
LLVM is written in C++; uses STL; vector, set and map

LLVM sources are hosted on GitHub
https://github.com/llvm/llvm-project

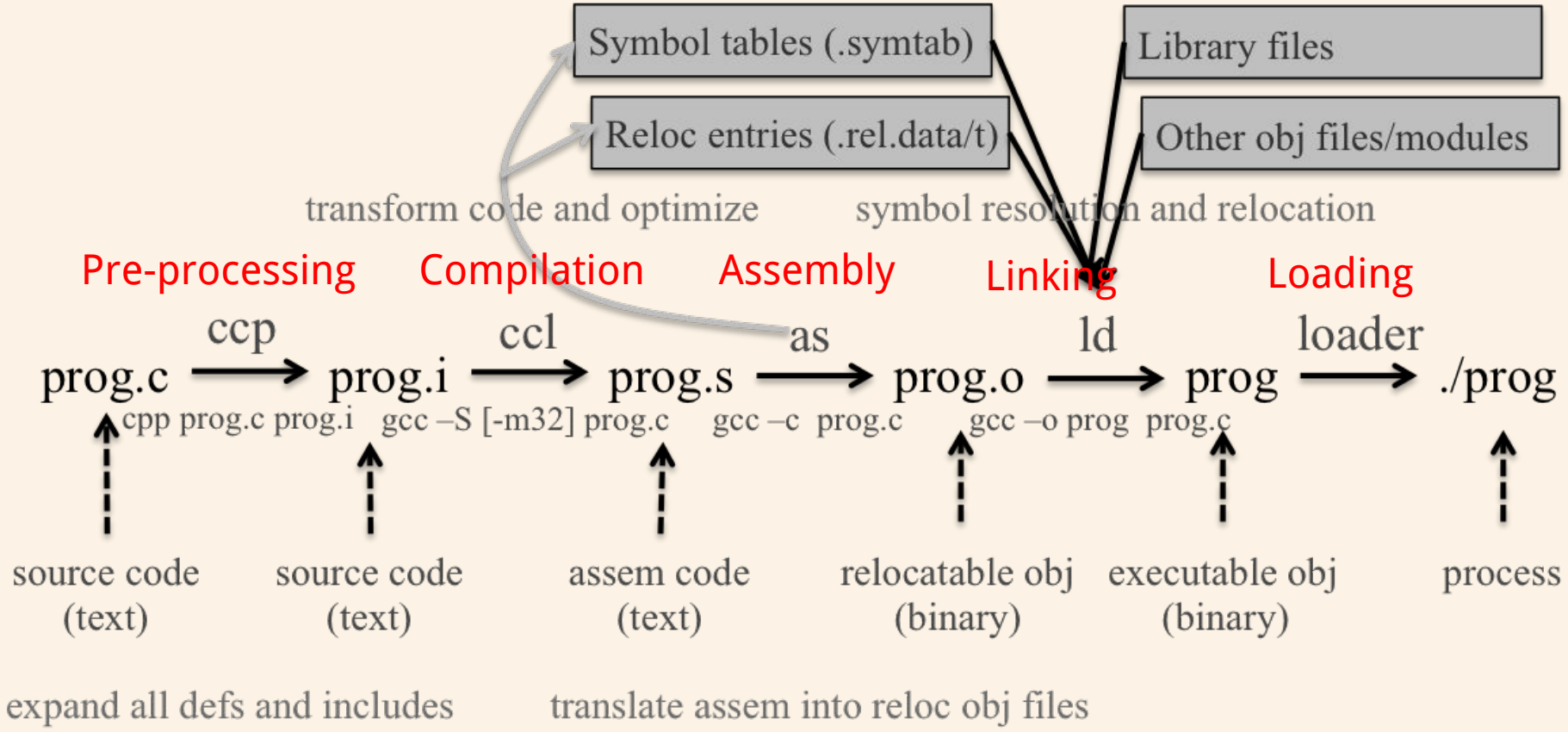
LLVM is split into multiple Git repositories
• For this class you will need the clang and llvm git repos

https://llvm.org/

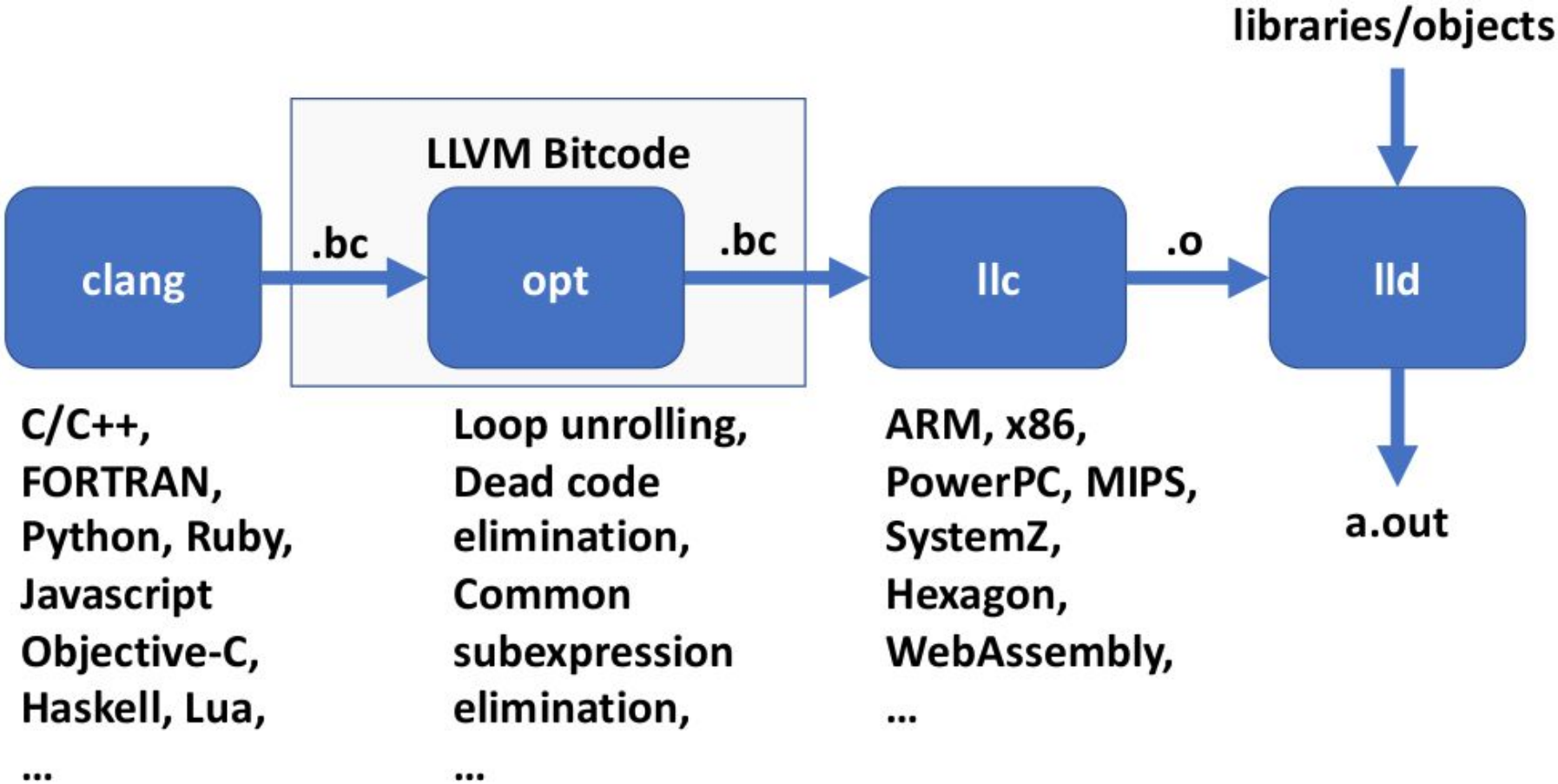
Typical Compiler Flow



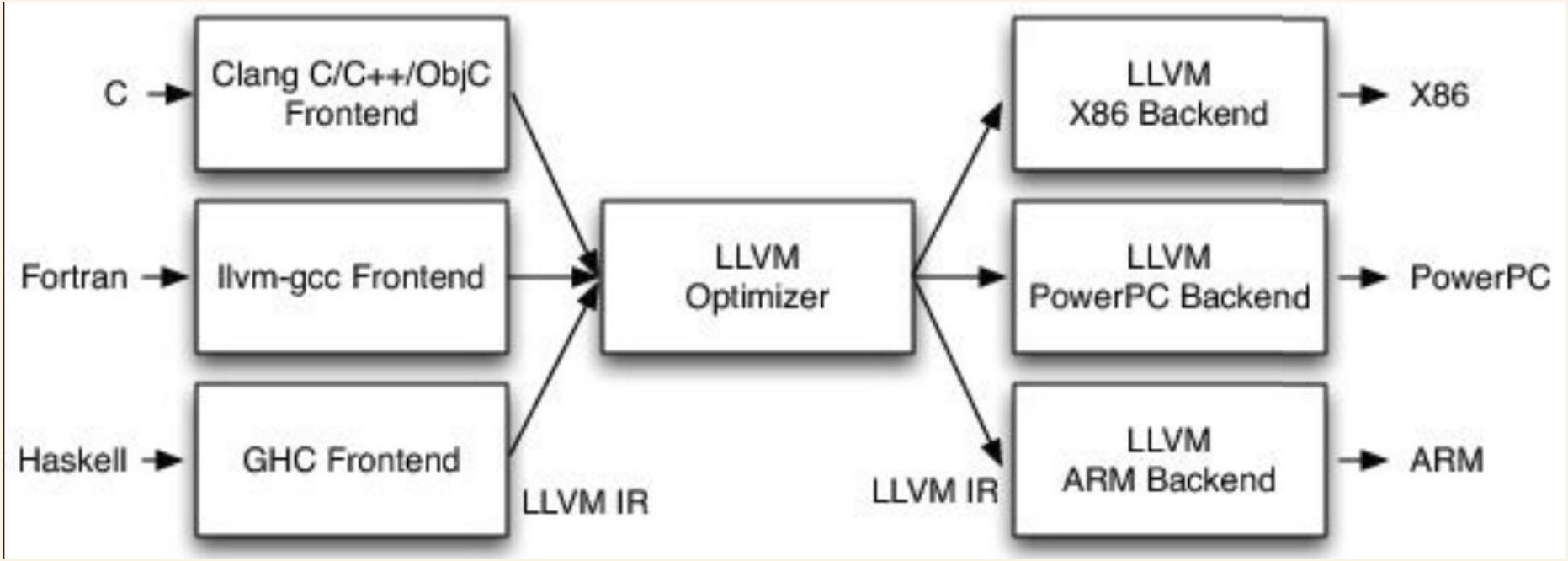
From a C program to a process



LLVM Flow


libraries/objects
LLVM Bitcode
clang
.bc
opt
.bc
llc
.o
lld
a.out
C/C++,
FORTRAN,
Python, Ruby,
Javascript
Objective-C,
Haskell, Lua,
…
Loop unrolling,
Dead code
elimination,
Common
subexpression
elimination,
…
ARM, x86,
PowerPC, MIPS,
SystemZ,
Hexagon,
WebAssembly,
…

LLVM Flow


C
Clang C/C++/ObjC Frontend
Fortran
llvm-gcc Frontend
Haskell
GHC Frontend
LLVM IR
LLVM Optimizer
LLVM IR
LLVM X86 Backend
X86
LLVM PowerPC Backend
PowerPC
LLVM ARM Backend
ARM

Clang/Clang++

Clang is a frontend for several C-family languages

- C and C++ being the most widely known
  - Supports C++11/14/17/20
- (Objective C/C++, OpenCL, CUDA< and RenderScript are the other C-style languages actively developed)

How to Generate LLVM IR from Source

To generate LLVM IR use clang with ‘-emit-llvm’ option
- ‘–S’ generates a text file and ‘–c’ generates a binary
- clang foo.c –emit-llvm –S
- clang foo.c –emit-llvm –c

To convert a binary file (.bc) to a text file (.ll) use the llvm disassembler
- llvm-dis foo.bc

To convert a text file (.ll) to a binary file (.bc) use the llvm assembler
- llvm-as foo.ll

What tools does LLVM provide?

- “Primitive” tools: do a single job
  - llvm-as: Convert from .ll (text) to .bc (binary)
  - llvm-dis: Convert from .bc (binary) to .ll (text)
  - llvm-link: Link multiple .bc files together
  - llvm-prof: Print profile output to human readers
  - llvmc: Configurable compiler driver
- Aggregate tools: pull in multiple features
  - gccas/gccld: Compile/link-time optimizers for C/C++ FE
  - bugpoint: automatic compiler debugger
  - llvm-gcc/llvm-g++: C/C++ compilers

See also: docs/CommandGuide/

What Optimizations does LLVM provide?

opt --help

opt tool: LLVM modular optimizer

- Invoke arbitrary sequence of passes:
  - Completely control PassManager from command line
  - Supports loading passes as plugins from .so files

  opt -load foo.so -pass1 -pass2 -pass3 x.bc -o y.bc
- Passes “register” themselves:

  61: RegisterOpt<SimpleArgPromotion> X("simpleargpromotion",
  "Promote 'by reference' arguments to 'by value'");
- From this, they are exposed through opt:

```
> opt -load libsimpleargpromote.so –help
 ...
-sccp            - Sparse Conditional Constant Propagation
-simpleargpromotion - Promote 'by reference' arguments to 'by
-simplifycfg       - Simplify the CFG
 ...
```

LLC Tool: Static code generator

- Compiles LLVM native assembly language
  - Currently for X86, Sparc, PowerPC (others in alpha)
  - llc file.bc -o file.s -march=x86
  - as file.s –o file.o
- Compiles LLVM portable C code
  - llc file.bc -o file.c -march=c
  - gcc –c file.c –o file.o
- Targets are modular & dynamically loadable:
  - llc –load libarm.so file.bc -march=arm

LLI Tool: LLVM Execution Engine

- LLI allows direct execution of .bc files
  - E.g.: lli grep.bc -i foo *.c
- LLI uses a Just-In-Time compiler if available:
  - Uses same code generator as LLC
    - Optionally uses faster components than LLC
  - Emits machine code to memory instead of “.s” file
  - JIT is a library that can be embedded in other tools
- Otherwise, it uses the LLVM interpreter:
  - Interpreter is extremely simple and very slow
  - Interpreter is portable though!

LLVM IR / LLVM Instruction Set

The LLVM Intermediate Representation

Some characteristics of LLVM IR

- RISC-like instruction set (3 addresses; human readable, assembly like)
- Strongly typed
- Explicit control flow
- Uses a virtual register set with infinite temporaries (%)
- In Static Single Assignment form
- Abstracts machine details such as calling conventions and stack references

LLVM IR reference is online

• https://llvm.org/docs/LangRef.html

The LLVM Intermediate Representation

LLVM IR is actually defined in three isomorphic forms

- the textual format above
- an in-memory data structure inspected and modified by optimizations themselves
- an efficient and dense on-disk binary "bitcode" format (.bc)

The LLVM Project also provides tools to convert the on-disk format from text to binary

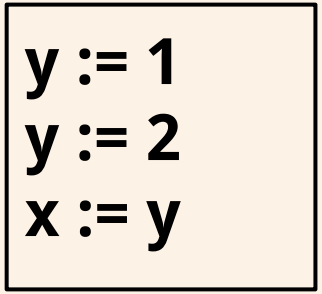
- llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop
- llvm-dis turns a .bc file into a .ll file.

Static Single Assignment (SSA) form

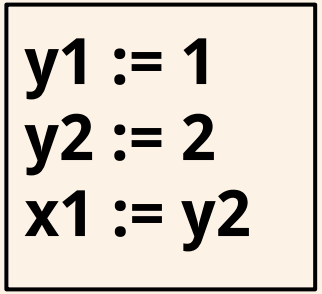
In compiler design, static single assignment form (often abbreviated as SSA form or simply SSA) is a property of an intermediate representation (IR), which requires that each variable be assigned exactly once, and every variable be defined before it is used.

SSA was proposed by Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck in POPL 1988

Static Single Assignment (SSA) form



Not SSA



SSA

An Example: add.c to add.ll

- Source filename Data layout
- Target Triple
- Functions, Structure Types
- Lots of % signs - These are registers (Remember the thing about SSA?)
- Other important things (not in this IR--phi nodes)
- Attributes
- type information! Cool--better than assembly!
- Metadata (At the end with the “!”)

LLVM tool: lli

lli - Directly executes programs bit-code using JIT

In fact the machine can read it, and the machine can directly execute the IR using it's Just-in-time (JIT compile for current architecture) execution engine.

LLVM tool: opt

opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files

It works by making several passes through a module of code looking for opportunities to ‘optimize’ the code.

There exists several ways to ‘pass’ through the code and gather information or make code changes.

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program. ■ i.e. A side effect occurs, which could invalidate other passes!

Hello Pass

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

LLVM Program Structure

- Module contains Functions/GlobalVariables
  - Module is unit of compilation/analysis/optimization
- Function contains BasicBlocks/Arguments
  - Functions roughly correspond to functions in C
- BasicBlock contains list of instructions
  - Each block ends in a control flow instruction
- Instruction is opcode + vector of operands
  - All operands have types
  - Instruction result is typed