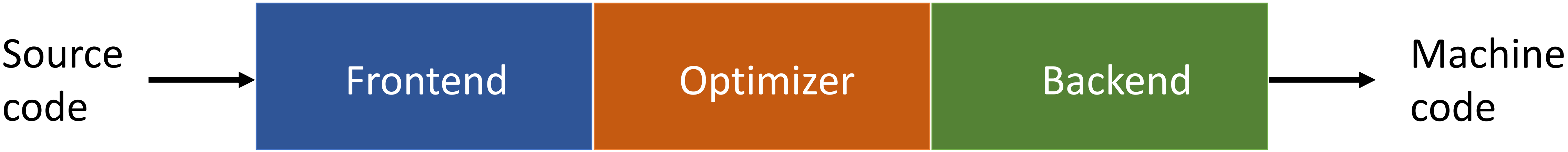
Swarnendu Biswas

Systems Bootcamp 2018

CSE, IIT Kanpur

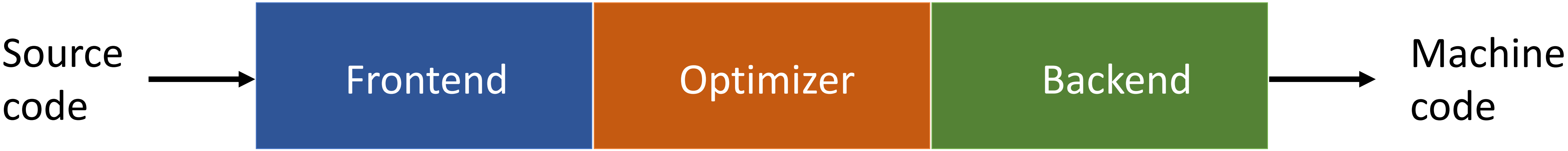
A VERY QUICK Introduction to LLVM

Classical Compiler Design



http://www.aosabook.org/en/llvm.html

Classical Compiler Design



Frontend

- Parses source code, checks for errors, builds an AST

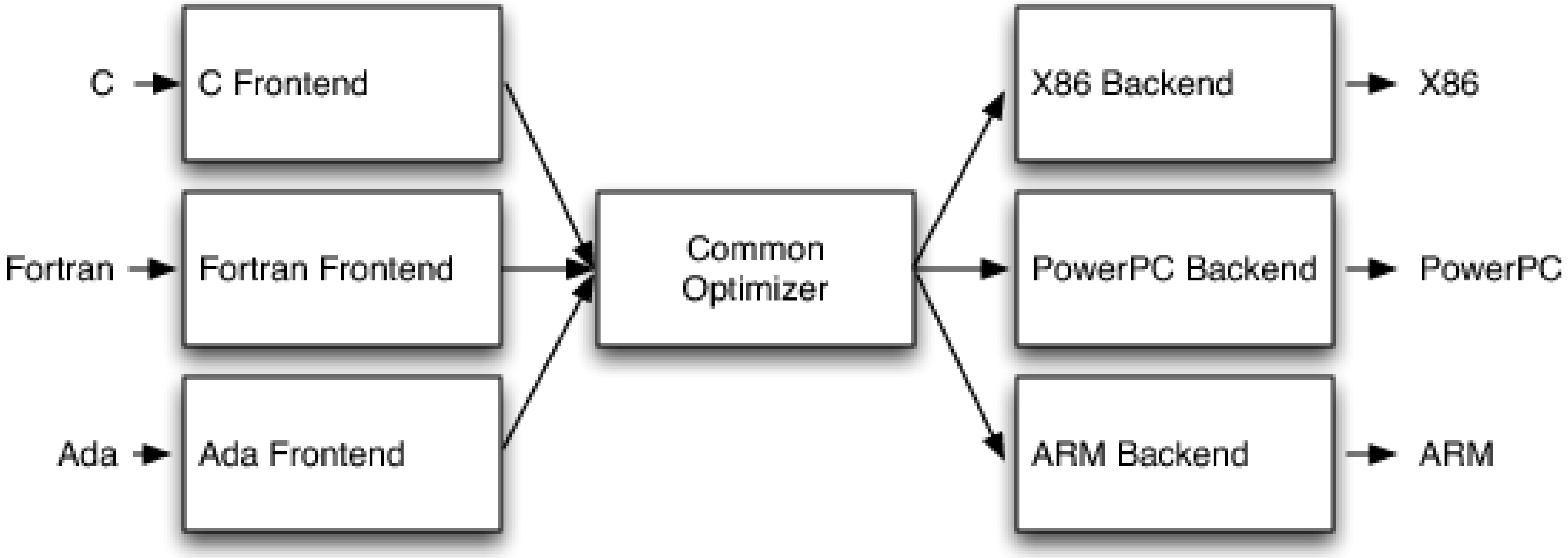
Optimizer

- Performs analyses and optimizations independent of the language and target architecture

Backend (code generator)

- Generates target code (performs register allocation, instruction selection, etc.)

Advantages of Three-Phase Partitioned Design



http://www.aosabook.org/en/llvm.html

GNU GCC Toolchain

- Probably most popular C/C++ toolchain
- Supports many frontends and backends
- Active and broad community

Problems with GNU GCC Toolchain

- Ancient code, monolithic structure
  - Global variables, poorly designed data structures, use of macros
  - Backend uses frontend ASTs
  - Difficult to reuse and modify for your own analyses
  - Little sharing across language implementations
  - Does not support JIT compilation
    - static compiler

What is LLVM?

- Umbrella project
  - Started by Chris Lattner and Vikram Adve
- a set of low-level toolchain components
  - assemblers, compilers, debuggers, etc.
- Old acronym: Low-Level Virtual Machine
  - **Not relevant any more**

Why LLVM?

- Beautiful architecture
- Ever-growing in popularity
  - Open-source project
    - ACM Software System Award 2012
  - Used by companies like Apple, Sony, Google
- Easier to play with compared to GCC

- Alas! Poor me!
  - Don’t make the same mistake!

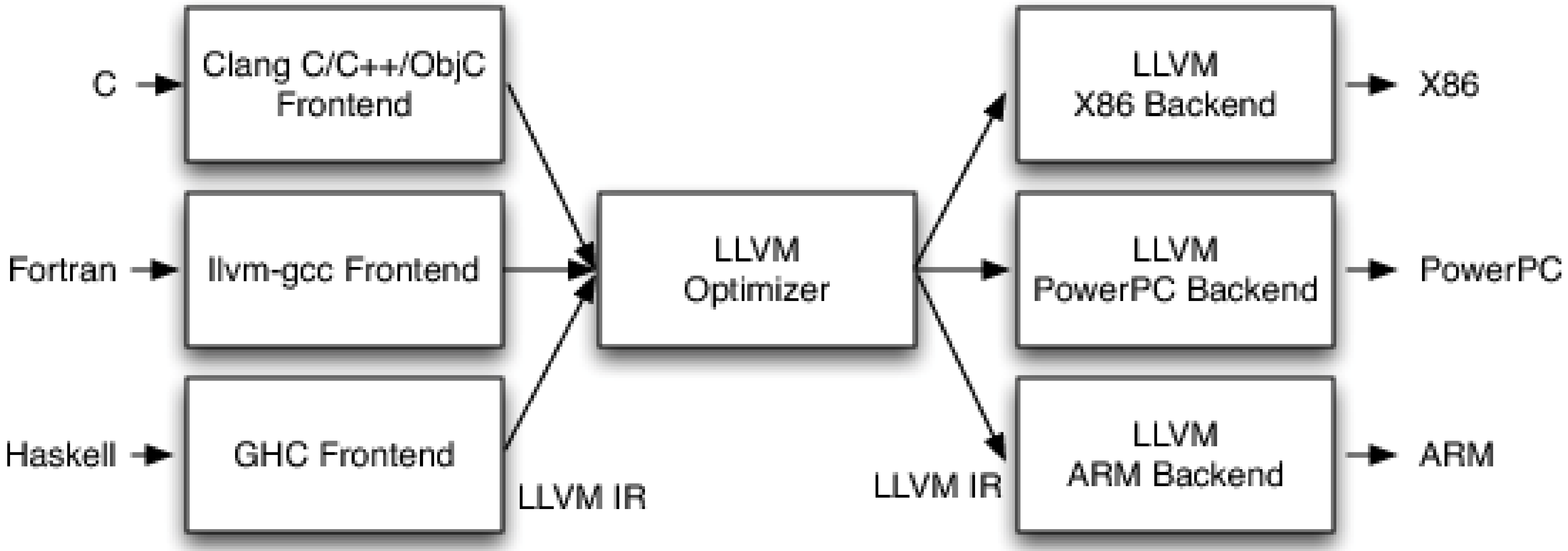
http://llvm.org/

Introduction to LLVM

- LLVM Infrastructure
  - Provides reusable components for building compilers
  - Reduces the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...
- LLVM Compiler Framework
  - End-to-end compilers using the infrastructure
  - Frontends for Ada, C, C++, D, Fortran, Haskell, Julia, Rust, Swift, …
  - Backends for C, X86, Sparc, PowerPC, Alpha, Arm, Thumb, IA-64…

David Koes. The LLVM Compiler Framework and Infrastructure.

Visualizing the LLVM Compiler System



Primary LLVM Components

- LLVM Virtual Instruction Set
  - Complete code representation
- Collection of libraries
  - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, …
- Tools built from the libraries
  - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer, …

David Koes. The LLVM Compiler Framework and Infrastructure.

Clang

- One Frontend to LLVM for the C language family
  - C, C++, Objective C/C++, OpenCL, CUDA, …
- Production-quality
  - Used in products like Google Chrome and Mozilla Firefox
- Supports all ISCO C++ standards
  - C++98/03/11/14/17
  - Specify the standard with –std=c++11

https://clang.llvm.org/

Clang Details

- Driver program
  - Preprocessing – “.i” (C), “.ii” (C++)
  - Parsing and semantic analysis – Parse tree -> AST
  - Code generation and optimization – AST -> LLVM IR -> “.s” assembly file
  - Assembler – Target “.o” object file
  - Linker – Link multiple object files into an “a.out” executable or “.so” dynamic library

Why Clang?

- Great for static analysis on C code
  - Been used for analyzing parts of the Linux kernel and drivers
  - LLVM will use its own IR (like assembly)

- Preprocessor – Expand macros
- Clang AST – Source representation

Compiling a C Program with Clang

1

Compile a C program

- clang prog2.c

2

See Clang AST

- clang –Xclang –ast-dump –fsyntax-only prog2.c

Read Clang AST

Important LLVM Tools

- clang – C/C++ compiler
- llvm-as – Assembles the textual .ll file to .bc file
- llvm-dis – Disassembles the .bc file to human-readable .ll file
- llvm-link – Bitcode linker that links several .bc files into a single LLVM .bc file
- lli – Interpreter and dynamic compiler
- llc – Compiles source inputs into assembly language for a specified architecture

Practice Using LLVM Toolchain

LLVM Internals

LLVM Instruction Set

- Low-level and target-independent semantics
  - First class language with well-defined semantics
  - RISC-like virtual instruction set
  - Three address code
  - Simple control flow constructs
  - Infinite virtual register set in SSA form
  - IR is strongly-typed

LLVM Instruction Set

- IR has text, binary, and in-memory isomorphic forms
  - Textual .ll format
  - On-disk binary bitcode (.bc) format
  - In-memory data structure

- Optimization passes only have to deal with IRs

Read LLVM IR (.ll?, .bc?)

Optimization Passes

- Compilation involves a series of passes
  - Each pass involves a code analysis or transformation
  - A pass can use information from other earlier passes

- O0 – no passes
- O3 – ~70 passes

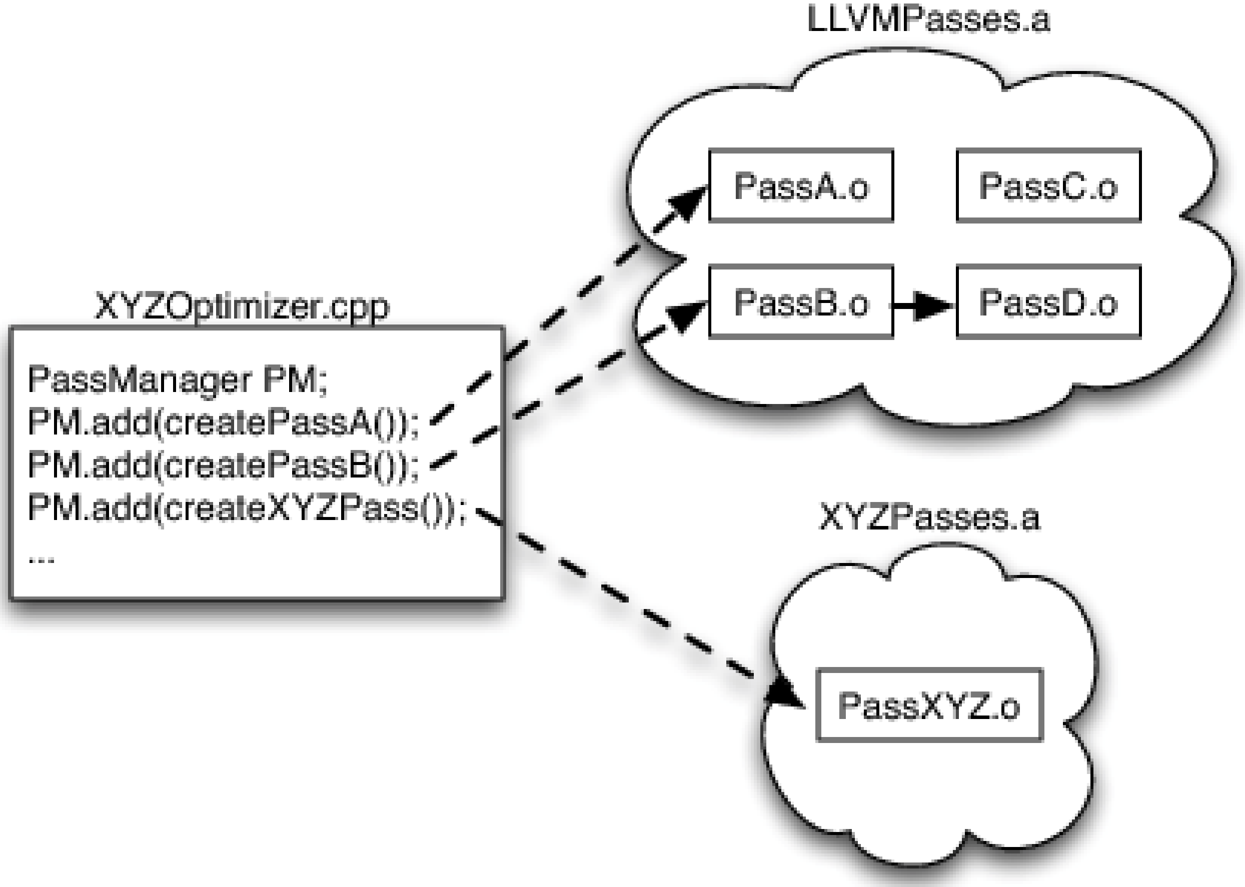
Some Important Optimization Passes

- Function inliner
- Loop invariant code motion
- Dominator Tree Construction
- Basic Alias Analysis
- Call Graph
- Module Verifier

Anatomy of Passes

- Optimization passes
  - **Processes a function at a time instead of a pass at a time**
  - Suppose a translation unit has three functions: F, G, and H, and there are two passes X and Y
  - The following are two possible orders of execution
    - `X(F)X(G)X(H) Y(F)Y(G)Y(H)`
    - `X(F)Y(F) X(G)Y(G) X(H)Y(H)`
    - Which do you think is advantageous?

Linking Passes



```
PassManager PM;
PM.add(createPassA());
PM.add(createPassB());
PM.add(createXYZPass());
...
```

http://www.aosabook.org/en/llvm.html

Implementing Passes in LLVM

- Different types
  - ModulePass
  - FuncionPass
    - Analyzes functions one by one, does not maintain state across functions
  - BasicBlockPass
  - CallGraphSCCPass
  - …

opt Tool: LLVM Modular Optimizer and Analyzer

- Invokes an arbitrary sequence of passes
  - Supports loading passes as plugins from .so files

Hacking LLVM

Understanding LLVM Code

- Written in modern C++, extensive use of STL
- LLVM IR is hierarchical
  - Module represents a translational unit (e.g., a .c/.cpp file)
    - List of GlobalVariables and Functions
  - Function consists of Arguments and BasicBlocks
  - BasicBlock contains list of Instructions
  - Instruction has Opcode + vector of Operands
    - Operands have types
    - Instruction result has a type

Traversing LLVM IR

- Linked lists are traversed with iterators
  - **Pre-increments on objects are more efficient than post-increment**

IterateFunctionIR Example

DirectCallSite Example

MutateOperator Example

InstrumentLocks Example

References

- https://llvm.org/docs/index.html
- Chris Lattner . "Introduction to the LLVM Compiler System", ACAT 2008.
- Chris Lattner. “The Architecture of Open Source Applications: LLVM”, http://www.aosabook.org/en/llvm.html
- https://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-i-introduction/#more-3

Swarnendu Biswas

Systems Bootcamp 2018

CSE, IIT Kanpur

A VERY QUICK Introduction to LLVM