Introduction to LLVM
Projects, Components, Integration, Internals

Renato Golin

Overview

LLVM is not a toolchain, but a number of sub-projects that can behave like one.

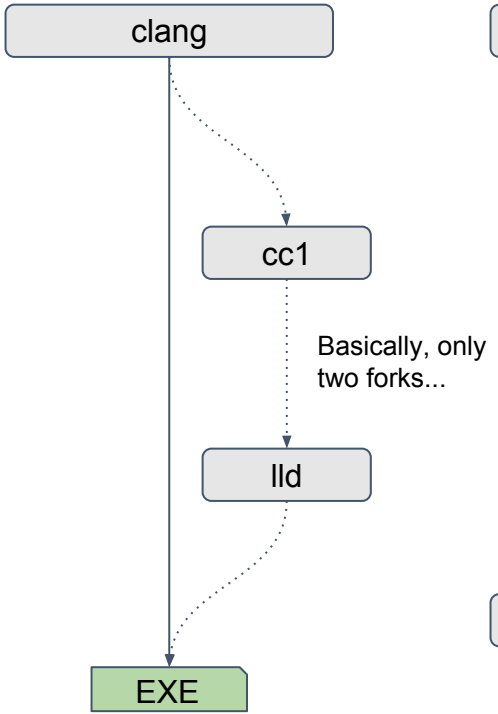
- Front-ends:
  - Clang (C/C++/ObjC/OpenCL/OpenMP), flang (Fortran), LDC (D), PGI's Fortran, etc
- Front-end plugins:
  - Static analyser, clang-tidy, clang-format, clang-complete, etc
- Middle-end:
  - Optimization and Analysis passes, integration with Polly, etc.
- Back-end:
  - JIT (MC and ORC), targets: ARM, AArch64, MIPS, PPC, x86, GPUs, BPF, WebAsm, etc.
- Libraries:
  - Compiler-RT, libc++/abi, libunwind, OpenMP, libCL, etc.
- Tools:
  - LLD, LLDB, LNT, readobj, llc, lli, bugpoint, objdump, lto, etc.

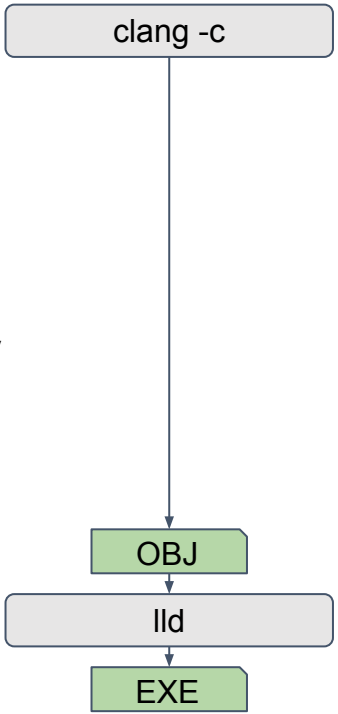
LLVM / GNU comparison

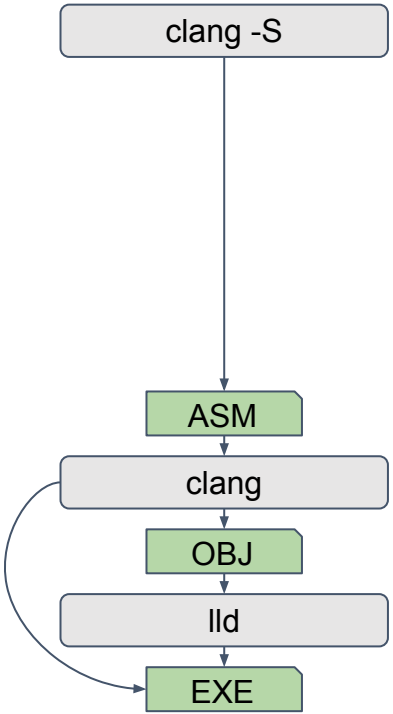
LLVM component / tools	GNU component / tool
Front-end: Clang	**Front-end**: CC1 / CPP
Middle-end: LLVM	**Middle-end**: GCC
Back-end: LLVM	**Back-end**: GCC
Assembler: LLVM (MC)	**Assembler**: GAS
Linker: LLD	**Linker**: GNU-LD
Libraries: Compiler-RT, libc++ (no libc)	**Libraries**: libgcc, stdlibc++, glibc
Debugger: LLDB / LLDBserver	**Debugger**: GDB / GDBserver

Source Paths

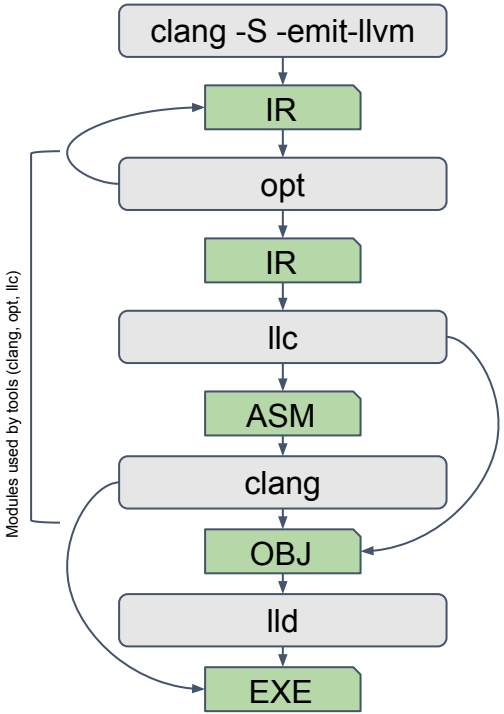
Direct route ...







... Individual steps



Perils of multiple paths

- Not all paths are created equal…
  - Core LLVM classes have options that significantly change code-gen
  - Target interpretation (triple, options) are somewhat independent
  - Default pass structure can be different
- Not all tools can pass all arguments…
  - Clang's driver can't handle some `-Wl`, and `-Wa`, options
  - Include paths, library paths, tools paths can be different depending on distro
  - GCC has build-time options (`--with-*`), LLVM doesn't (new flags are needed)
- Different order produces different results…
  - "`opt -O0`" + "`opt -O2`" != "`opt -O2`"
  - Assembly notation, parsing and disassembling not entirely unique / bijective
  - So, (clang -emit-llvm)+(llc -S)+(clang -c) != (clang -c)
  - Not guaranteed distributive, associative or commutative properties

C to IR

- IR is **not** target independent
  - Clang produces *reasonably* independent IR
  - Though, data sizes, casts, C++ structure layout, ABI, PCS are all taken into account
- `clang -target <triple> -O2 -S -emit-llvm file.c`

C

```
#include <stdio.h>

int foo(int j, int d) {
  return j / d ;
}

int main( void ) {
  int r = foo(5, 0);
  printf(" r = %d\n", r);
}
```

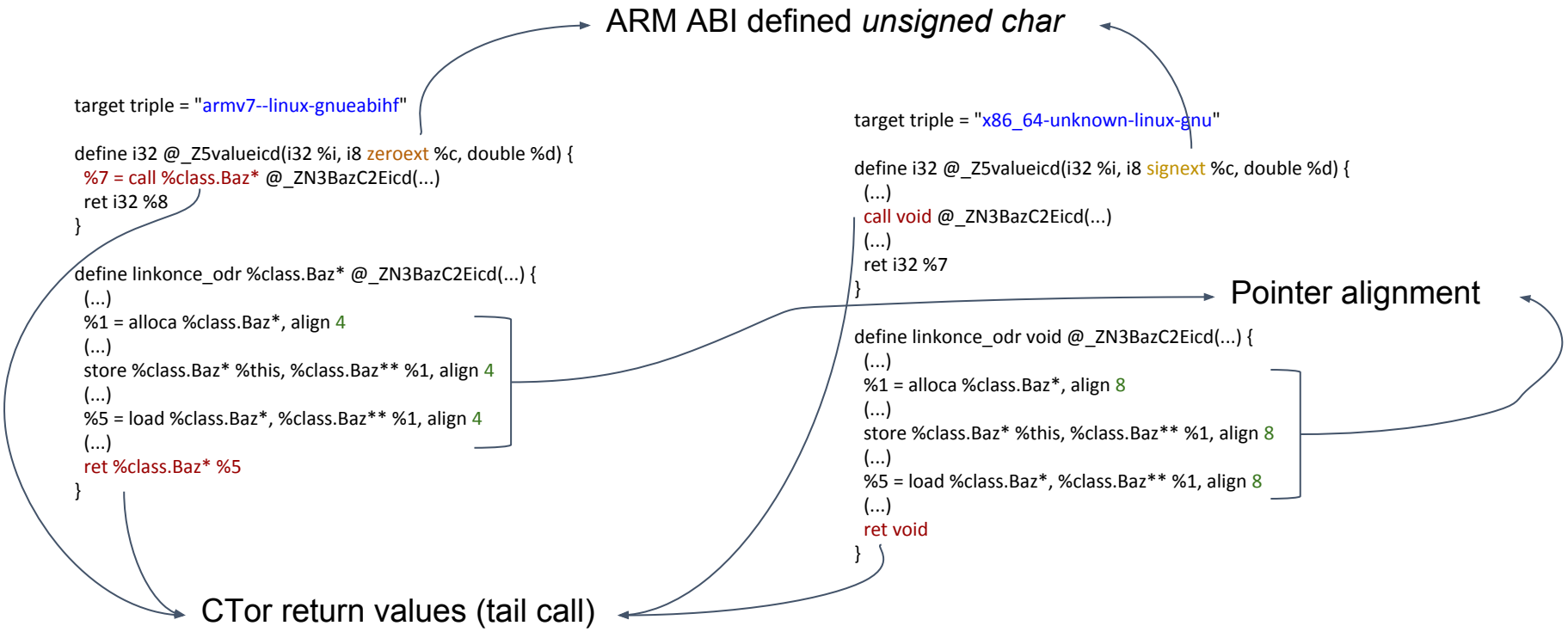
x86_64

```
@.str = private unnamed_addr constant [9 x i8] c" r =
%d\0A\00", align 1

; Function Attrs: norecurse nounwind readnone uwtable
define i32 @foo(i32 %j, i32 %d) #0 {
  %1 = sdiv i32 %j, %d
  ret i32 %1
}

; Function Attrs: nounwind uwtable
define i32 @main() #1 {
  %1 = tail call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([9 x i8], [9 x i8]* @.str, i64 0, i64 0), i32 undef)
  ret i32 0
}
```

ARM

```
@.str = private unnamed_addr constant [9 x i8] c" r =
%d\0A\00", align 1

; Function Attrs: norecurse nounwind readnone
define arm_aapcscc i32 @foo(i32 %j, i32 %d) #0 {
  %1 = sdiv i32 %j, %d
  ret i32 %1
}

; Function Attrs: nounwind
define arm_aapcscc i32 @main() #1 {
  %1 = tail call arm_aapcscc i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([9 x i8], [9 x i8]* @.str, i32 0, i32
0), i32 undef)
  ret i32 0
}
```

ABI differences in IR



Optimization Passes & Pass Manager

- Types of passes
  - Analysis: Gathers information about code, can annotate (metadata)
  - Transform: Can change instructions, entire blocks, usually rely on analysis passes
  - Scope: Module, Function, Loop, Region, BBlock, etc.
- Registration
  - Static, via `INITIALIZE_PASS_BEGIN / INITIALIZE_PASS_DEPENDENCY` macros
  - Implements `getAnalysisUsage()` by registering required / preserved passes
  - The PassManager is used by tools (clang, llc, opt) to add passes in specific order
- Execution
  - Registration order pass: Module, Function, …
  - Push dependencies to queue before next, unless it was preserved by previous passes
  - Create a new { module, function, basic block } → change → validate → replace all uses

IR transformations

- **opt** is a developer tool, to help test and debug passes
  - Clang, llc, lli use the same infrastructure (not necessarily in the same way)
  - `opt -S -sroa file.ll -o opt.ll`

`O0` `+SROA` `-print-before|after-all`

```
define i32 @foo(i32 %j, i32 %d) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %j, i32* %1, align 4
  store i32 %d, i32* %2, align 4
  %3 = load i32, i32* %1, align 4
  %4 = load i32, i32* %2, align 4
  %5 = sdiv i32 %3, %4
  ret i32 %5
}
```

```
define i32 @foo(i32 %j, i32 %d) #0 {
  %1 = sdiv i32 %j, %d
  ret i32 %1
}
```

```
*** IR Dump Before SROA ***
; Function Attrs: nounwind uwtable
define i32 @foo(i32 %j, i32 %d) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %j, i32* %1, align 4
  store i32 %d, i32* %2, align 4
  %3 = load i32, i32* %1, align 4
  %4 = load i32, i32* %2, align 4
  %5 = sdiv i32 %3, %4
  ret i32 %5
}
```

```
*** IR Dump After SROA ***
; Function Attrs: nounwind uwtable
define i32 @foo(i32 %j, i32 %d) #0 {
  %1 = sdiv i32 %j, %d
  ret i32 %1
}
```

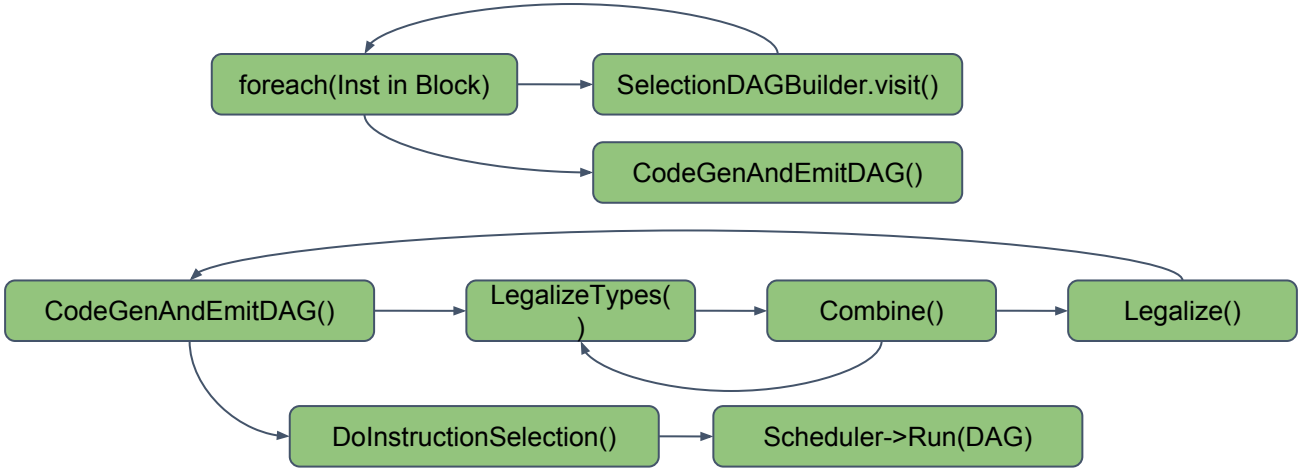
```
bool SROA::runOnFunction(Function &F) {
  if (skipFunction(F))
    return false;

  bool Changed = performPromotion(F);
  (...)
```

```
bool SROA::performPromotion(Function &F) {
  (...)
  if (HasDomTree)
    PromoteMemToReg(Allocas, *DT, nullptr, &AC);
```

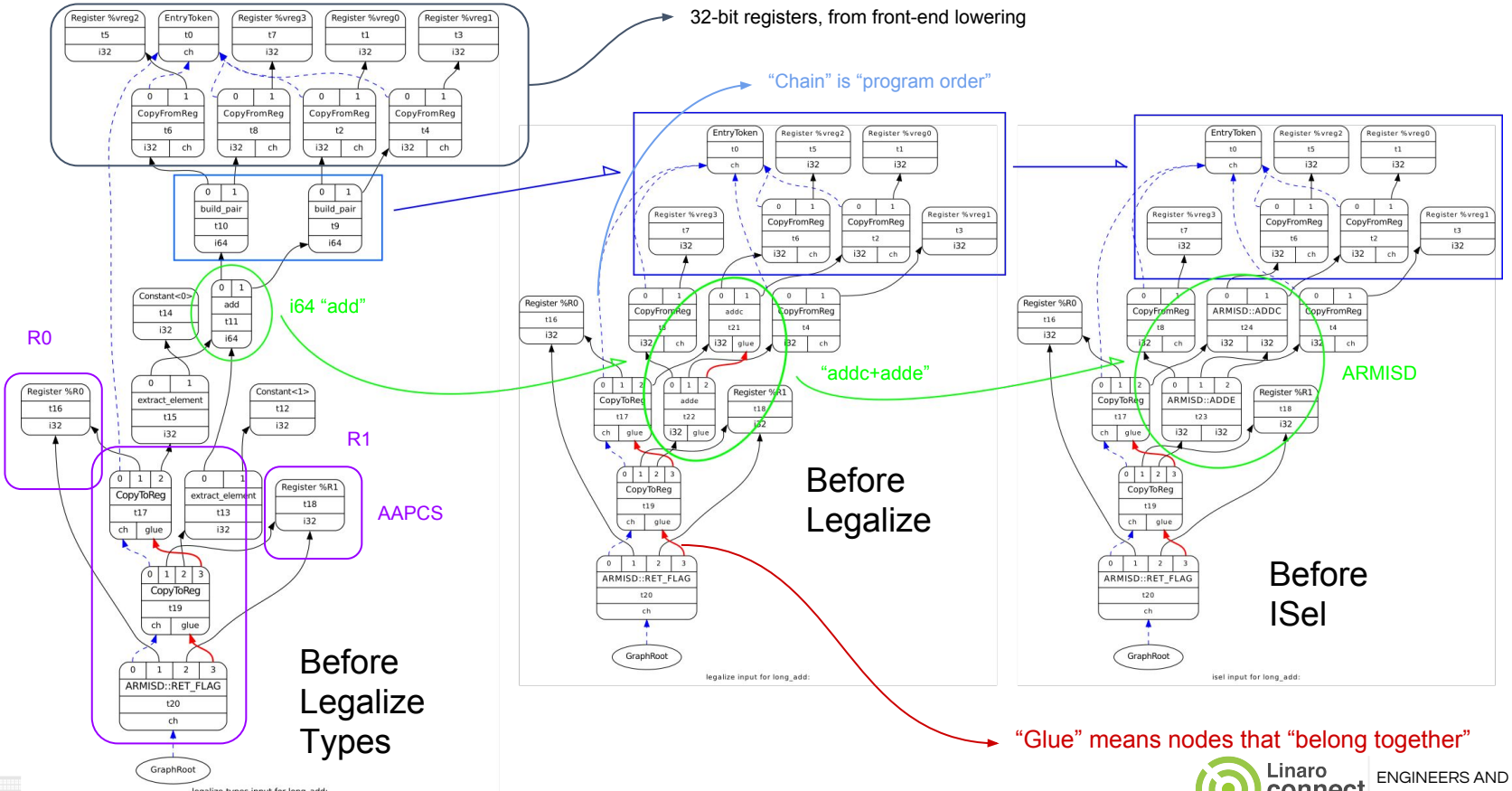
Nothing to do with SROA… :)

IR Lowering

- SelectionDAGISel
  - IR to DAG is target Independent (with some target-dependent hooks)
  - `runOnMachineFunction(MF)` → For each Block → `SelectBasicBlock()`
  - Multi-step legalization/combining because of type differences (patterns don't match)



DAG Transformation

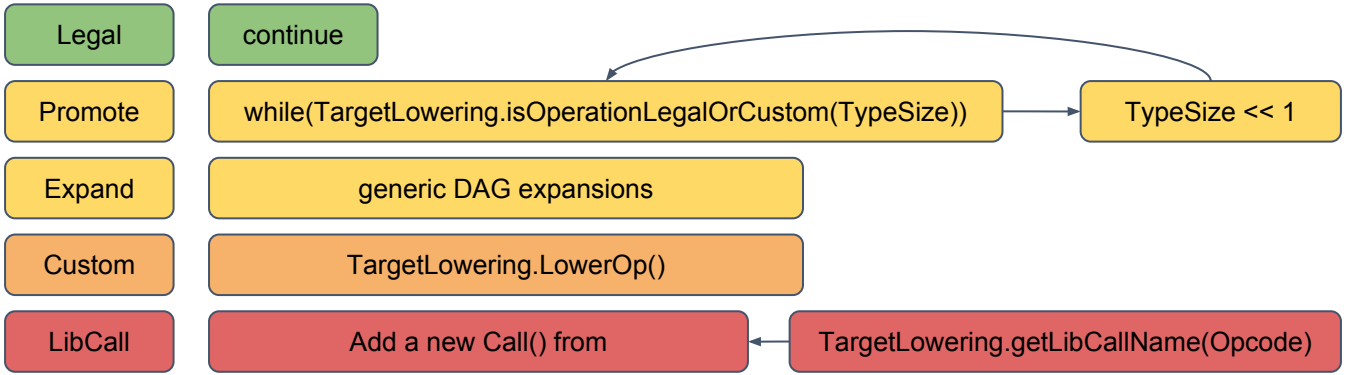


Legalize Types & DAG Combining

- LegalizeTypes
  - `for(Node in Block) { Target.getTypeAction(Node.Type);`
  - If type is not Legal, `TargetLowering::Type<action><type>`, ex:
    - `TypeExpandInteger`
    - `TypePromoteFloat`
    - `TypeScalarizeVector`
    - etc.
  - An ugly chain of GOTOs and switches with the same overall idea (switch(Type):TypeOpTy)
- DAGCombine
  - Clean up dead nodes
  - Uses TargetLowering to combine DAG nodes, bulk of it C++ methods `combine<Opcode>()`
  - Promotes types after combining, to help next cycle's type legalization

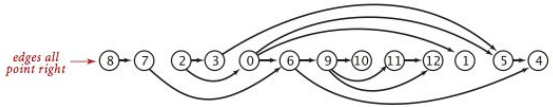
DAG Legalization

- LegalizeDAG
  - `for(Node in Block) { LegalizeOp(Node); }`
  - `Action = TargetLowering.getOperationAction(Opcode, Type)`



Instruction Selection & Scheduler

- Instruction Selection
  - <Target>ISelLowering: From SDNode (ISD::) to (ARMISD::)
  - <Target>ISelDAGToDAG: From SDNode (ARMISD::) to MachineSDNode (ARM::)
  - ABI/PCS registers, builtins, intrinsics
  - Still, some type legalization (for new nodes)
  - Inline assembly is still text (will be expanded in the MC layer)
- Scheduler
  - Sorts DAG in topological order
  - Inserts / removes edges, updates costs based on TargetInformation
  - *Glue* keeps paired / dependent instructions together
  - Target's schedule is in TableGen (most inherit from basic description + specific rules)
  - Produces `MachineBasicBlocks` and `MachineInstructions` (MI)
  - Still in SSA form (virtual registers)



Register Allocation & Serialization

- Register allocators
  - **Fast**: Linear scan, multi-pass (define ranges, allocate, collect dead, coalesce)
  - **Greedy**: default on optimised builds (live ranges, interference graph / colouring)
  - **PBQP**: Partitioned Boolean Quadratic Programming (constraint solver, useful for DSP)
- MachineFunction passes
  - Before/after register allocation
  - Frame lowering (prologue/epilogue), EH tables, constant pools, late opts.
- Machine Code (MC) Layer
  - Can emit both assembly (`<Target>InstPrinter`) and object (`<Target>ELFStreamer`)
  - Most MCInst objects can be constructed from TableGen, some need custom lowering
  - Parses inline assembly and inserts instructions in the MC stream, matches registers, etc
  - Inline Asm local registers are reserved in the register allocator and linked here
  - Also used by assembler (`<Target>AsmParser`) and disassembler (`<Target>Disassembler`)

Assembler / Disassembler

- AsmParser
  - Used for both asm files and inline asm
  - Uses mostly TableGen instruction definitions (Inst, InstAlias, PseudoInst)
  - Single pass assembler with a few hard-coded transformations (which makes it messy)
  - Connects into MC layer and can output text (ex. llvm-mc) or object code (ex. clang)
- MCDisassembler
  - Iteration of trial and fail (ARM, Thumb, VFP, NEON, etc)
  - Most of it relies on TableGen encodings, but there's **a lot** of hard-coded stuff
  - Doesn't know much about object formats (ELF/COFF/MachO)
  - Used by `llvm-objdump`, `llvm-mc`, connects back to MC layer

TableGen

- Parse hardware description and generates code and tables to describe them
  - Common parser (same language), multiple back-ends (different outputs)
  - Templated descriptive language, good for composition and pattern matching
  - Back-ends generate multiple tables/enums with header guards + supporting code
- Back-ends describe their registers, instructions, schedules, patterns, etc.
  - Definition files generated at compile time, included in CPP files using *define-include* trick
  - Most matching, cost and code generating patterns are done via TableGen
- Clang also uses it for diagnostics and command line options
- Examples:
  - Syntax
  - Define-include trick
  - Language introduction and formal definition

Libraries

- LibC++
  - Complete Standard C++ library with native C++11/14 compatibility (no abi_tag necessary)
  - Production in FreeBSD, Darwin (MacOS)
- LibC++abi (similar to `libgcc_eh`)
  - Exception handling (cxa_*)
- Libunwind (similar to `libgcc_s`)
  - Stack unwinding (Dwarf, SjLj, EHABI)
- Compiler-RT (similar to `libgcc` + “stuff”)
  - Builtins + sanitizers + profile + CFI + etc.
  - Some inter/intra-dependencies (with clang, libc++abi, libunwind) being resolved
  - Generic C implementation + some Arch-specific optimized versions (build dep.)

Sanitizers

- Not static analysis
  - The code needs to be compiled with instrumentation (`-fsanitize=address`)
  - And executed, preferably with production workloads
- Not Valgrind
  - The instrumentation is embedded in the code (orders of magnitude faster)
  - But needs to re-compile code, work around bugs in compilation, etc.
- Compiler instrumentation
  - In Clang and GCC
  - Add calls to instrumentation before load/stores, malloc/free, etc.
- Run-time libraries
  - Arch-specific instrumentation on how memory is laid out, etc.
  - Maps loads/stores, allocations, etc. into a shadow memory for tagging
  - Later calls do sanity checks on shadow tags and assert on errors

Sanitizers: Examples

- ASAN: Address Sanitizer (~2x slower)
  - Out-of-bounds (heap, stack, BSS), use-after-free, double-free, etc.
- MSAN: Memory Sanitizer (no noticeable penalty)
  - Uninitialised memory usage (suggestions to merge into ASAN)
- LSAN: Leak Sanitizer (no noticeable penalty)
  - Memory leaks (heap objects losing scope)
- TSAN: Thread Sanitizer (5~10x slower on x86_64, more on AArch64)
  - Detects data races
  - Needs 64-bit pointers, to use the most-significant bits as tags
  - Due to multiple VMA configurations in AArch64, additional run-time checks are needed
- UBSAN: Undefined Behaviour Sanitizer (no noticeable penalty)
  - Integer overflow, null pointer use, misaligned reads

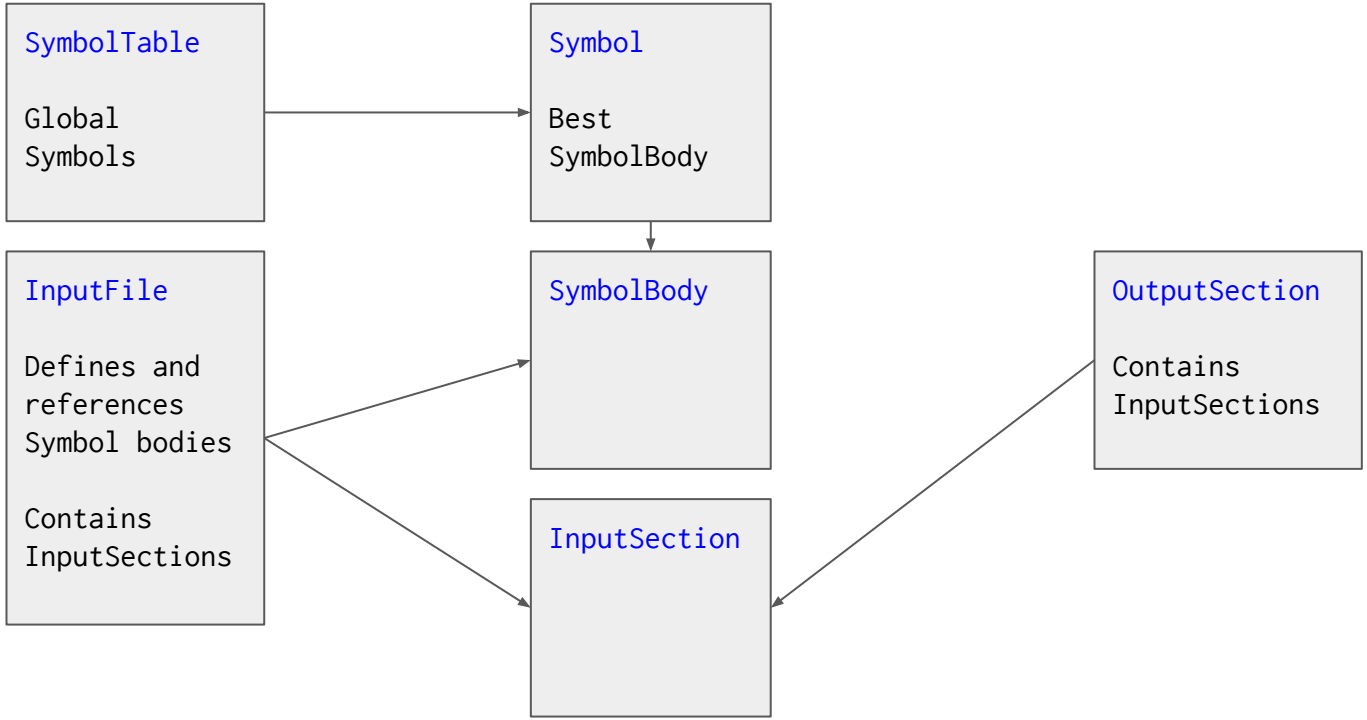
LLD the llvm linker

- Since May 2015, 3 separate linkers in one project
  - ELF, COFF and the Atom based linker (Mach-O)
  - ELF and COFF have a similar design but don't share code
  - Primarily designed to be system linkers
    - ELF Linker a drop in replacement for GNU ld
    - COFF linker a drop in replacement for link.exe
  - Atom based linker is a more abstract set of linker tools
    - Only supports Mach-O output
  - Uses llvm object reading libraries and core data structures
- Key design choices
  - Do not abstract file formats (c.f. BFD)
  - Emphasis on performance at the high-level, do minimal amount as late as possible.
  - Have a similar interface to existing system linkers but simplify where possible

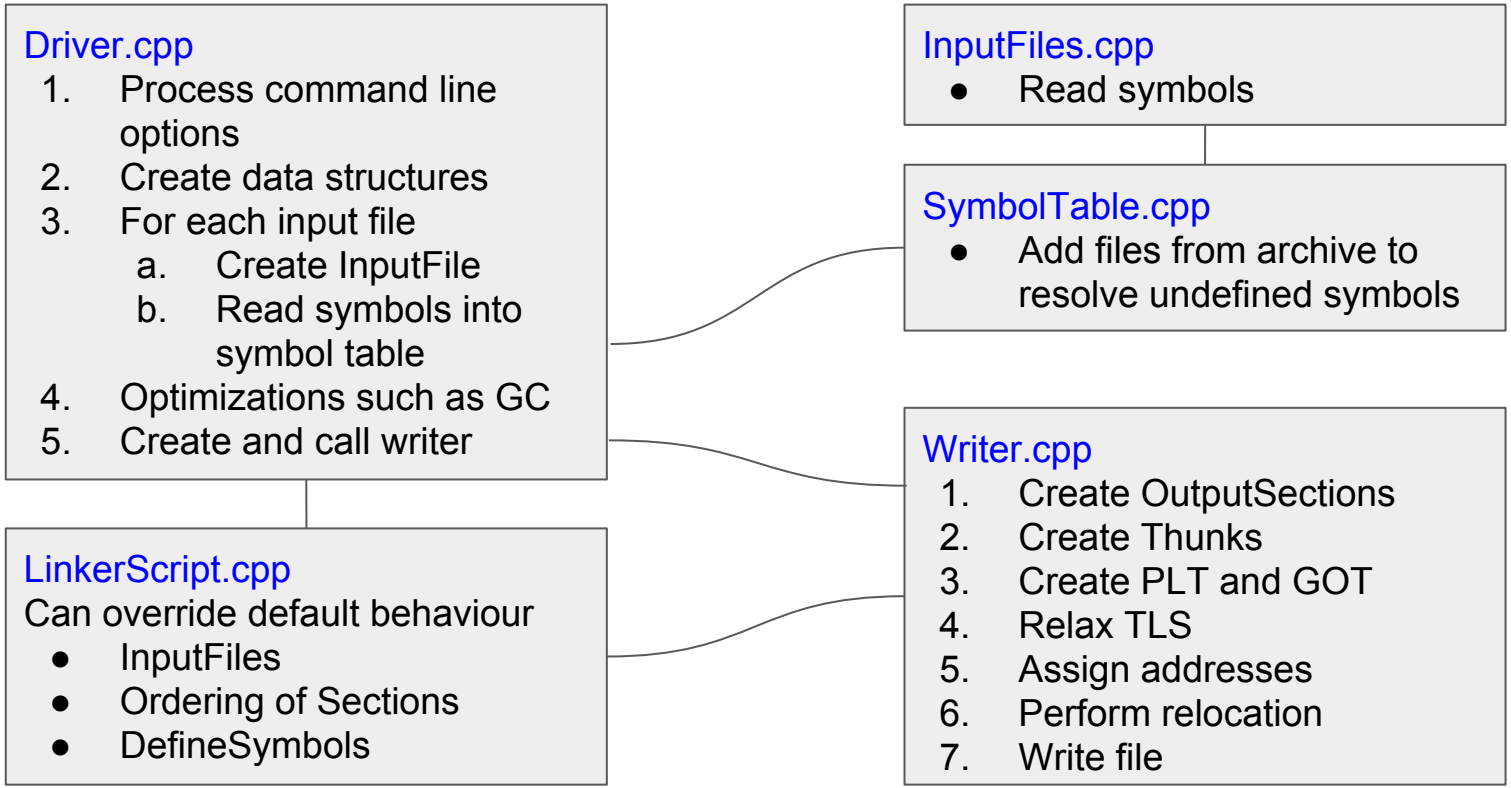
LLD ELF

- Support for AArch64, amd64, ARM (sort of), Mips, Power, X86 targets
- Focused on Linux and BSD like ELF files suitable for demand paging
- FreeBSD team aiming at using it for AARCH64 lld
- Many ld command line options supported
- Linker script support in active development
- Not ready for being a callable library yet

LLD key data structure relationship



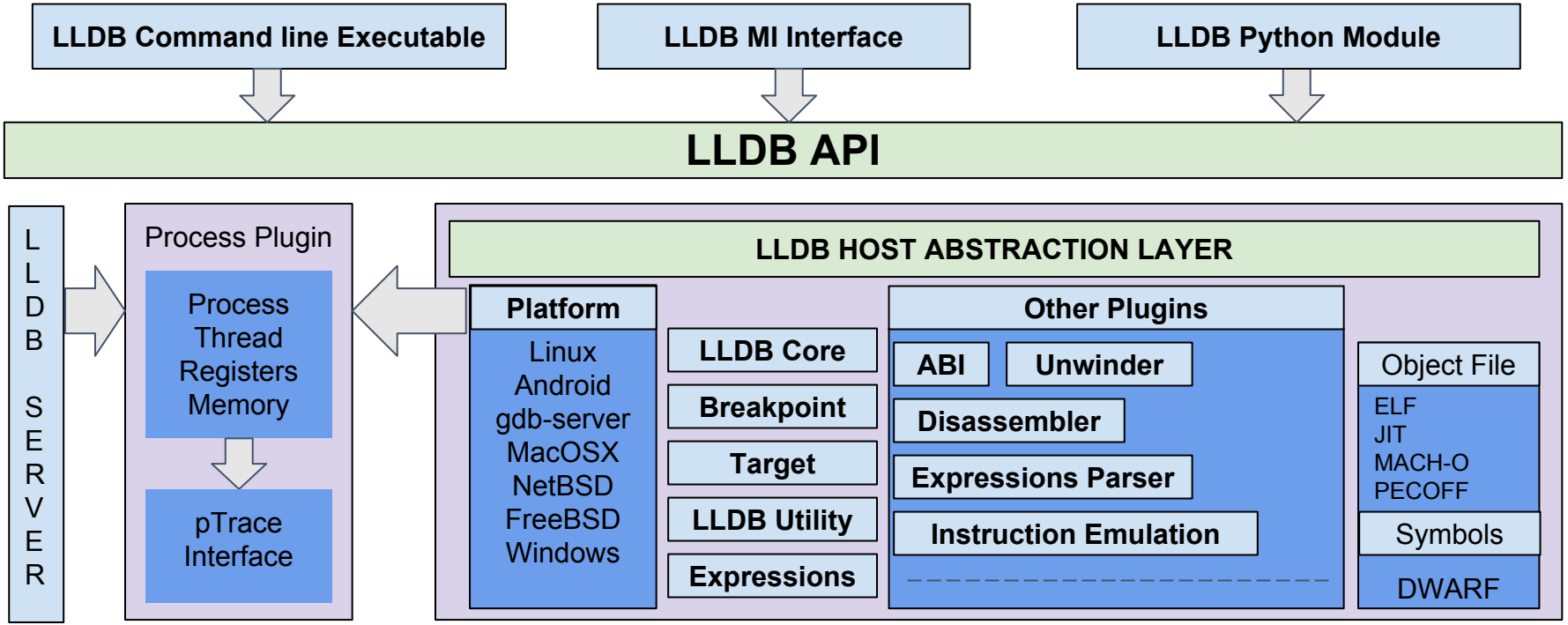
LLD control flow



LLDB

- A modern, high-performance source-level debugger written in C++
- Extensively under development for various use-cases.
- Default debugger for OSX, Xcode IDE, Android Studio.
- Re-uses LLVM/Clang code JIT/IR for expression evaluation, disassembly etc.
- Provides a C++ Debugger API which can be used by various clients
- **Supported Host Platforms**
  - OS X, Linux/Android, FreeBSD, NetBSD, and Windows
- **Supported Target Architectures**
  - i386/x86_64, Arm/AArch64, MIPS/MIPS64, IBM s390
- **Supported Languages**
  - Fully support C, C++ and Objective-C while SWIFT and GoLang (under development)

LLDB Architecture



References

- Official docs
  - LLVM docs (LangRef, Passes, CodeGen, BackEnds, TableGen, Vectorizer, Doxygen)
  - Clang docs (LangExt, SafeStack, LTO, AST)
  - LLDB (Architecture, GDB to LLDB commands, Doxygen)
  - LLD (New ELF/COFF backend)
  - Sanitizers (ASAN, TSAN, MSAN, LSAN, UBSAN, DFSAN)
  - Compiler-RT / LibC++ (docs)
- Blogs
  - LLVM Blog
  - LLVM Weekly
  - Planet Clang
  - Eli Bendersky's **excellent** blog post: Life of an instruction in LLVM
  - Old and high level, bug good overall post by Chris Lattner
  - Not that old, but great HowTo adding a new back-end
  - libunwind is not easy!

Thank You

#LAS16

For further information: www.linaro.org

LAS16 keynotes and videos on: connect.linaro.org