Introduction to LLVM

Eric Christopher (@echristo) and Johannes Doerfert (jdoerfert@anl.gov)

Why are you here?

- Brief overview of LLVM
- Understand some of the various pieces
- Want to know where to go for more

- Everything in this talk easily could get its own 45 minute tutorial!

Source Layout

`llvm-project`	LLVM mono-repo (https://github.com/llvm/llvm-project)
`llvm-project/clang`	C/C++/Cuda/OpenCL… language front-end
`llvm-project/llvm`	“core” LLVM (middle-end & backends)
`llvm-project/{lld, lldb, …}`	“LLVM-based” tools, linker, debugger, …

all subprojects look similar: `include, lib, tests, utils`

(see laso http://llvm.org/docs/GettingStarted.html#general-layout)

Configure & Build

Single command often suffices to configure:

```
cmake /src/llvm-project/llvm -DLLVM_ENABLE_PROJECTS='clang;openmp'
make -j
```

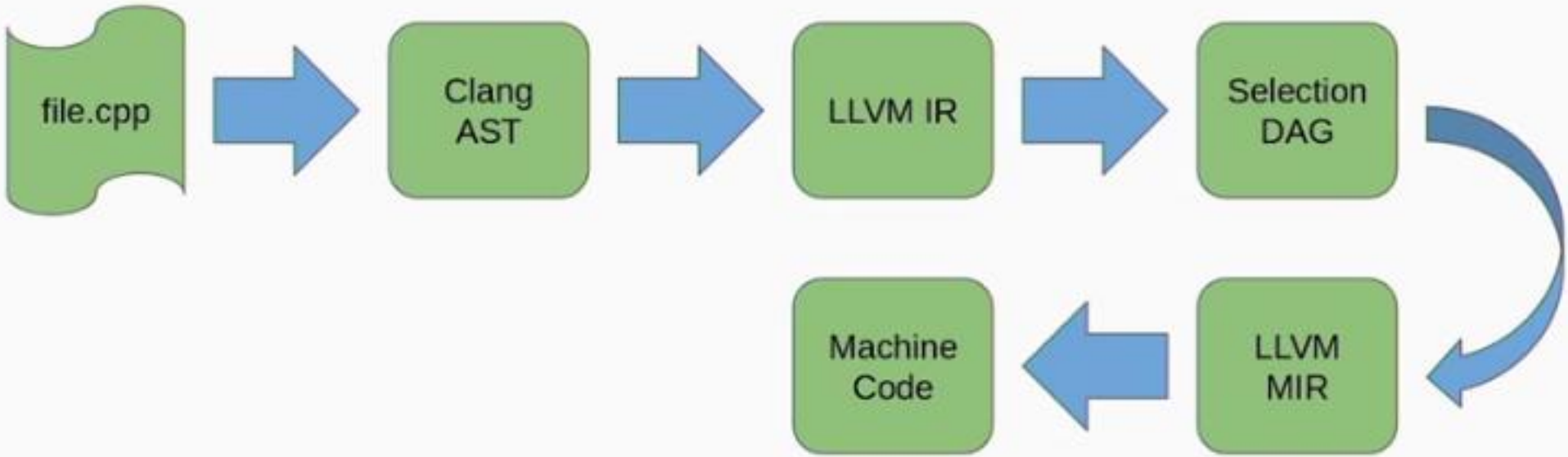
Useful options include:

```
CMAKE_BUILD_TYPE={Release,Asserts,…}
LLVM_ENABLE_ASSERTIONS={ON,OFF}
LLVM_CCACHE_BUILD={ON,OFF}
-G Ninja
```

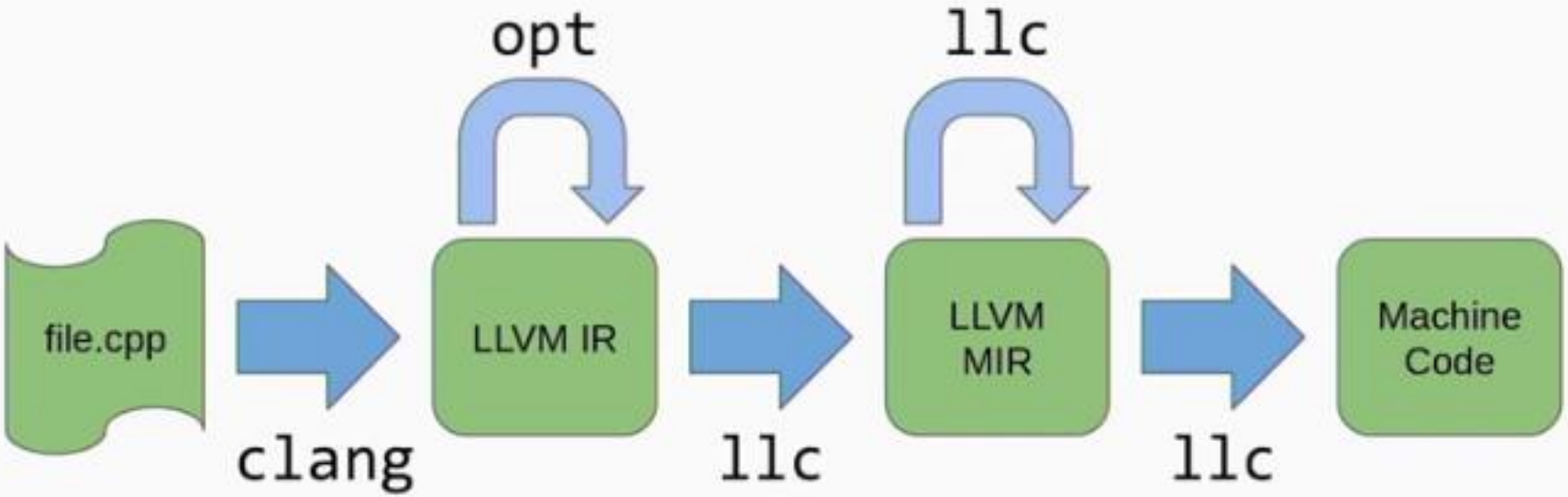
Various resources available online! Start here:

http://llvm.org/docs/GettingStarted.html

Compilers 101

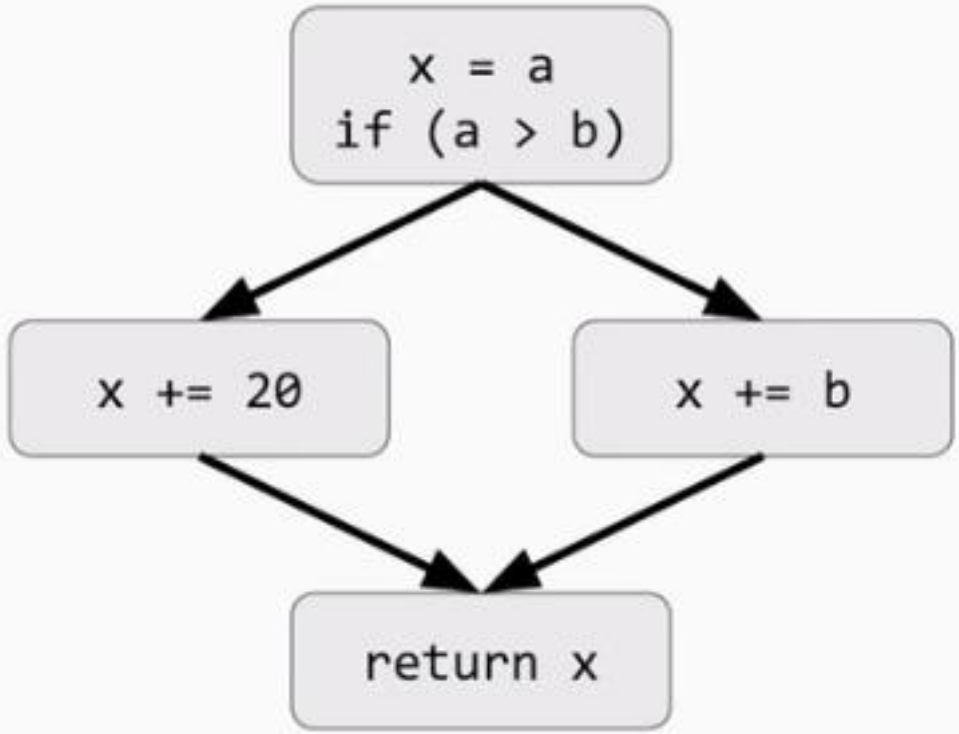


Compilers 101



Life of a Program: Control Flow Graph (CFG)

```
long f(long a, long b) {
  long x = a;
  if (a > b)
    x += 20;
  else
    x += b;
  return x;
}
```



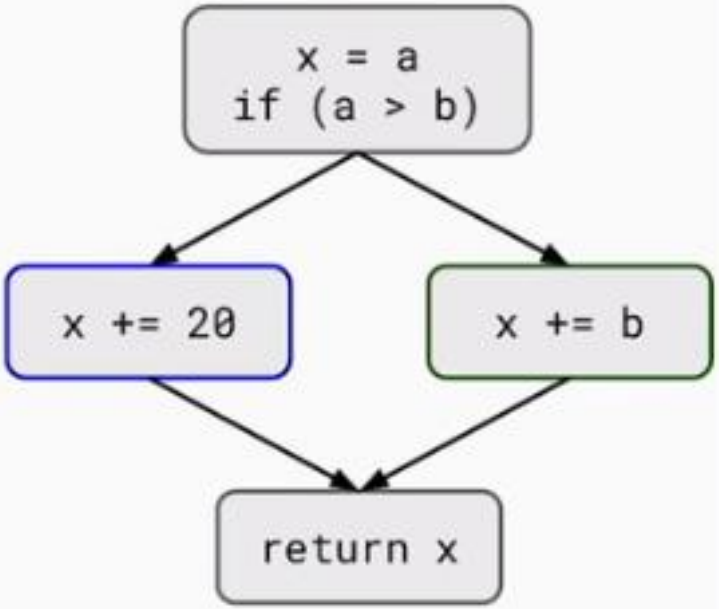
Life of a Program: C -> LLVM-IR

```
clang -O3 -Xclang -disable-llvm-passes                    \ # prepare for O3 but don't run it
      -S -emit-llvm code.c -o code.ll                     \ # emit LLVM IR for code.c into code.ll
opt -S -mem2reg -instnamer code.ll -o code_before_opt.ll  \ # slight cleanup
```

```
; ModuleID = 'f.ll'
source_filename = "f.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define dso_local i64 @f(i64 %a, i64 %b) #0 {
```

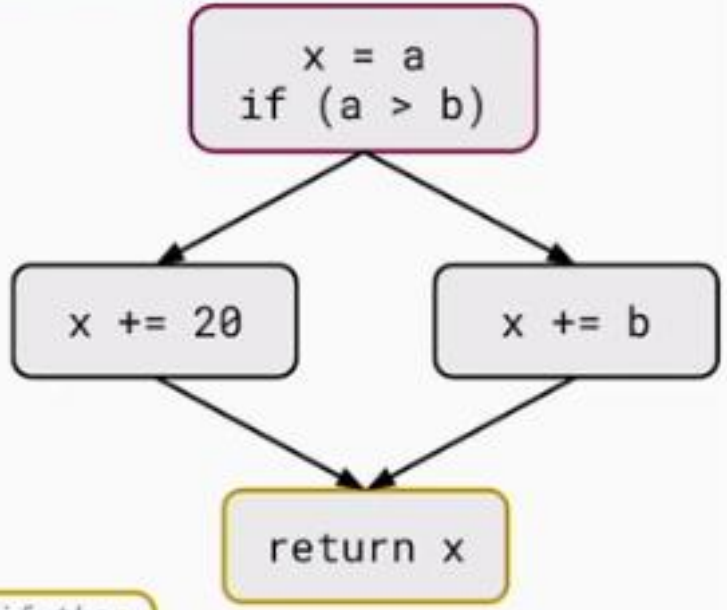
```
if.then:                                          ; preds = %entry
  %add = add nsw i64 %a, 20
  br label %if.end
```

```
if.else:                                          ; preds = %entry
  %add1 = add nsw i64 %a, %b
  br label %if.end
```




```
}

attributes #0 = { nounwind uwtable }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0"}
```

```
; ModuleID = 'f.ll'
source_filename = "f.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define dso_local i64 @f(i64 %a, i64 %b) #0 {
entry:
  %cmp = icmp sgt i64 %a, %b
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %add = add nsw i64 %a, 20
  br label %if.end

if.else:                                          ; preds = %entry
  %add1 = add nsw i64 %a, %b
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.0 = phi i64 [ %add, %if.then ], [ %add1, %if.else ]
  ret i64 %x.0
}

attributes #0 = { nounwind uwtable }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0"}
```



LLVM-IR Hierarchy

- llvm::Module
  - llvm::GlobalVariable
  - llvm::Function (declarations & definitions)
    - llvm::BasicBlock
      - llvm::Instruction
        - llvm::ICmpInst
        - llvm::BranchInst

LLVM Intermediate Representation (LLVM-IR)

```
%sum = add i32 4, %var                    ; Binary operations
%cmp = icmp sge i32 %a, %b

%value = load i32, i32* %ptr              ; Memory operations
store i32 %value, i32* %ptr2

br label %next-block                      ; Terminator operations
br i1 %cmp, label %then-block, label %else-block
ret float %a
ret void

%X = trunc i32 257 to i8                  ; Cast operations
%Y = sext i32 %V to i64
%Z = bitcast i8* %q to i32*

%ptr = alloca [4 x double]                ; Other operations
%ret = call i32 @foo(i8* %fmt, i16 %val)
%phi = phi float [ %valA, %predBlockA ], [ %valB, %predBlockB ]
%elemI = getelementptr float, float* %A, i64 %I
```

LLVM Intermediate Representation (LLVM-IR)

assembly-like language **always** in SSA-form with infinite registers.

Organization:

`module`	= list of global symbols (functions, global variables, …)
`function`	= list of basic blocks that form a control-flow-graph (CFG)
`basic block`	= list of instructions, terminated by a branch/return/…
`instruction`	= typed assembly instruction with operands
`constant`	= constant literals, globals, ...
`value`	= almost anything, mostly constants and instructions

LLVM Intermediate Representation (LLVM-IR)

assembly-like language **always** in SSA-form with infinite registers.

- “global symbols” start with “@”
- “local symbols” start with “%”
- Basic block names start with “%” when used
- Basic block names end with “:” when defined

LLVM Intermediate Representation (LLVM-IR)

Common pitfalls:

- control-flow can be irreducible and irreducible loops are “not recognized”
- only *reachable* LLVM-IR has to be *well-formed*, e.g., wrt. SSA properties
- address spaces are part of the pointer type (and not to be ignored!)
- `null` (= `0`) can be a valid pointer, even in address space 0 (=default)
- types are sign agnostic
- llvm:ontants include values “constant at startup time”, e.g., addr of globals

Useful command line options

```
clang -mllvm <option for opt>

--help & --help-hidden

--debug & --debug-only=attributor,must-execute

--debug-pass={Arguments,Structure,Executions,Details}

--print-before-all/--print-after-all/--print-XXXX/--dot-XXXX
```

Targets and Code Generation

Codegen

- Instruction Selectors
- Register Allocation
- Instruction Scheduling
- Machine IR
- Machine Optimizations
- Machine Specific Features (stack protection, etc)
- Exception Handling
- “Assembly Printing”
- Target Dependency Breaking
- Calls heavily into Target code

Target

- x86, ARM, AArch64, WebAssembly, Power, Hexagon, CPUs, GPUs, Bytecodes
- Assembly Parsing/Disassembly
- Instruction Descriptions
- Optimization Passes
- CPU pipeline descriptions
- Calling Conventions and ABI
- Machine Code and Object File

Targets and Code Generation - An Example

```
def LEA64_32r : I<0x8D, MRMSrcMem,
                  (outs GR32:$dst), (ins lea64_32mem:$src),
                  "lea{l}\t{$src|$dst}, {$dst|$src}",
                  [(set GR32:$dst, lea64_32addr:$src)]>,
                  OpSize32, Requires<[In64BitMode]>;
```

Targets and Subtargets

`Target/`

- TargetMachine contains Module level lowering, data, and transform information
- Module level means object file/OS ABI

- Subtarget contains Function level lowering information.
- Primarily initialized with CPU and Features
- SubtargetFeatures include things like ISA level

Requires<[In64BitMode]>

`X86.td:`

```
def In64BitMode  :
Predicate<"Subtarget->is64Bit()">
```

`X86Subtarget.h,.cpp`

```
bool is64Bit() { return In64BitMode;}

In64BitMode(TargetTriple.getArch() ==
Triple::x86_64)
```

Machine IR

`CodeGen/MachineInstr.h`

`Target/`

- Follows a similar hierarchy to the rest of LLVM IR
- Dependent upon LLVM IR constructs for
- Extensive APIs to construct MIR
- Contains target-dependent and target-independent opcodes, registers
- Can be produced out of the `llc` program

X86FastISel.cpp:

```
unsigned Opc = TLI.getPointerTy(DL) ==
MVT::i32 ?
(Subtarget->isTarget64BitILP32() ?
X86::LEA64_32r : X86::LEA32r)
: X86::LEA64r;

addFullAddress(BuildMI(*FuncInfo.MBB,
FuncInfo.InsertPt, DbgLoc,
TII.get(Opc), ResultReg), AM);
```

Instruction Selection and Register Allocation

`CodeGen/`

- Instruction pattern describes what is happening.
- 3 Instruction Selectors
  - SelectionDAG
  - FastISel
  - GlobalISel
- Operands describe what and where
- 2* Register Allocators
  - RegAllocFast
  - RegAllocGreedy

`SelectionDAG View`

```
(set GR32:$dst,
lea64_32addr:$src)
```

`X86InstrInfo.td`

```
def lea64_32addr : ComplexPattern<i32,
5, "selectLEA64_32Addr"...
```

`X86ISelDAGToDAG.cpp:`

```
bool
X86DAGToDAGISel::selectLEA64_32Addr
```

Object Generation - Assembly “Printing”

`CodeGen/AsmPrinter`

- Target independent MIR to Assembly translation
- Calls into backends

`X86AsmPrinter::PrintLeaMemReference`

`Target/*AsmPrinter`

- Handles target specific opcode and operand printing

LEA64_32r : I<0x8D

leal 485498096(%edx,%eax,4), %eax

Object Generation - Assembly “Parsing”

`AsmParser/`

encoding: [0x8d,0x84,0x82,0xf0,0x1c,0xf0,0x1c]

`MC/`

- Handles target-independent object file construction and encoding

`Target/AsmParser`

`Target/MCTargetDesc`

- Parses target specific assembly and encodings
- Handles target-dependent details

Object Reading

`Object/`

`DebugInfo/`

- Parses object files and target-independent data
- Calls into the target for instruction decoding

`Target/Disassembler`

- Handles target specific instruction decoding

llvm-objdump.cpp:

```
Binary &Binary = *OBinary.getBinary()

ObjectFile *O = dyn_cast<ObjectFile>(&Binary)

DIContext DICtx = DWARFContext::create(*O)

DICtx->dump(outs(), DumpOpts)
```

More Tools

`tools/`

- Examples of APIs
- Canonical usage
- Updated frequently

- Can be moved into a library if we have a need for reuse