Lecture 3
Overview of the LLVM Compiler

Jonathan Burket

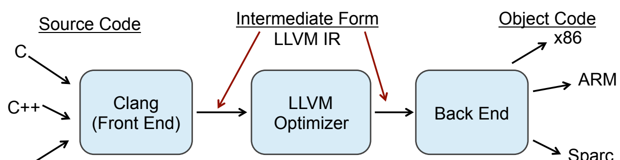
Special thanks to Deby Katz,
Gennady Pekhimenko, Olatunji Ruwase,
Chris Lattner, Vikram Adve, and David Koes for their slides

LLVM Compiler System

The LLVM Compiler Infrastructure
- Provides reusable components for building compilers
- Reduce the time/cost to build a new compiler
- Build different kinds of compilers
- Our homework assignments focus on static compilers
- There are also JITs, trace-based optimizers, etc.

The LLVM Compiler Framework
- End-to-end compilers using the LLVM infrastructure
- Support for C and C++ is robust and aggressive:
- Java, Scheme and others are in development
- Emit C code or native code for X86, Sparc, PowerPC

Visualizing the LLVM Compiler System

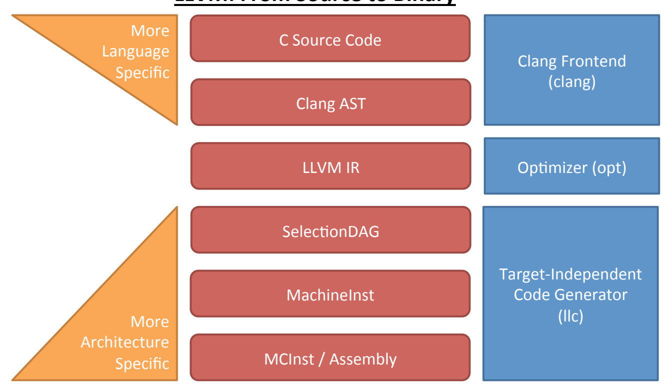


Three-Phase Design

The LLVM Optimizer is a series of "passes"
- –**Analysis and optimization passes, run one after another**
- –***Analysis* passes do not change code, *optimization* passes do**

LLVM Intermediate Form is a *Virtual Instruction Set*
- –Language- and target-independent form
  - Used to perform the same passes for all source and target languages
- –Internal Representation (IR) and external (persistent) representation

LLVM: From Source to Binary



C Source Code

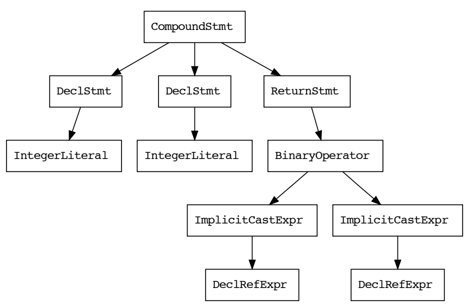
```c
int main() {
   int a = 5;
   int b = 3;
   return a - b;
}
```

Read “Life of an instruction in LLVM”:
http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

Clang AST

```
TranslationUnitDecl 0xd8185a0 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xd818870 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'char *'
`-FunctionDecl 0xd8188e0 <example.c:1:1, line:5:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xd818a90 <col:12, line:5:1>
    |-DeclStmt 0xd818998 <line:2:5, col:14>
    | `-VarDecl 0xd818950 <col:5, col:13> col:9 used a 'int' cinit
    |   `-IntegerLiteral 0xd818980 <col:13> 'int' 5
    |-DeclStmt 0xd818a08 <line:3:5, col:14>
    | `-VarDecl 0xd8189c0 <col:5, col:13> col:9 used b 'int' cinit
    |   `-IntegerLiteral 0xd8189f0 <col:13> 'int' 3
    `-ReturnStmt 0xd818a80 <line:4:5, col:16>
      `-BinaryOperator 0xd818a68 <col:12, col:16> 'int' '-'
        |-ImplicitCastExpr 0xd818a48 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xd818a18 <col:12> 'int' lvalue Var 0xd818950 'a' 'int'
        `-ImplicitCastExpr 0xd818a58 <col:16> 'int' <LValueToRValue>
          `-DeclRefExpr 0xd818a30 <col:16> 'int' lvalue Var 0xd8189c0 'b' 'int'
```


Clang AST



LLVM IR

In-Memory Data Structure

Bitcode (.bc files)

```
42 43 C0 DE  21 0C 00 00
06 10 32 39  92 01 84 0C
0A 32 44 24  48 0A 90 21
18 00 00 00  98 00 00 00
E6 C6 21 1D  E6 A1 1C DA
          …
```

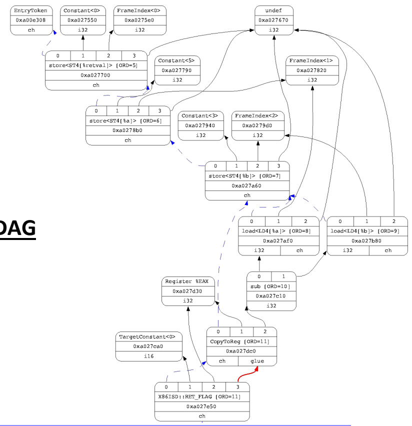
llvm-dis ↔ llvm-asm

Text Format (.ll files)

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
...
```

Bitcode files and LLVM IR text files are **lossless serialization formats**!
We can pause optimization and come back later.

SelectionDAG



Carnegie Mellon

Machine Inst

BB#0: derived from LLVM BB %entry
  Live Ins: %EBP
    PUSH32r %EBP<kill>, %ESP<imp-def>, %ESP<imp-use>; flags: FrameSetup
    %EBP<def> = MOV32rr %ESP; flags: FrameSetup
    %ESP<def,tied1> = SUB32ri8 %ESP<tied0>, 12, %EFLAGS<imp-def,dead>; flags: FrameSetup
    MOV32mi %EBP, 1, %noreg, -4, %noreg, 0; mem:ST4[%retval]
    MOV32mi %EBP, 1, %noreg, -8, %noreg, 5; mem:ST4[%a]
    MOV32mi %EBP, 1, %noreg, -12, %noreg, 3; mem:ST4[%b]
    %EAX<def> = MOV32rm %EBP, 1, %noreg, -8, %noreg; mem:LD4[%a]
    %EAX<def,tied1> = ADD32ri8 %EAX<kill,tied0>, -3, %EFLAGS<imp-def,dead>
    %ESP<def,tied1> = ADD32ri8 %ESP<tied0>, 12, %EFLAGS<imp-def,dead>
    %EBP<def> = POP32r %ESP<imp-def>, %ESP<imp-use>
    RETL %EAX

Carnegie Mellon

MCInst

```
#BB#0:                      # %entry
pushl   %ebp                # <MCInst #2191 PUSH32r
                            #  <MCOperand Reg:20>>
movl    %esp, %ebp          # <MCInst #1566 MOV32rr
                            #  <MCOperand Reg:20>
                            #  <MCOperand Reg:30>>
subl    $12, %esp           # <MCInst #2685 SUB32ri8
                            #  <MCOperand Reg:30>
                            #  <MCOperand Reg:30>
                            #  <MCOperand Imm:12>>
movl    $0, -4(%ebp)        # <MCInst #1554 MOV32mi
                            #  <MCOperand Reg:20>
                            #  <MCOperand Imm:1>
                            #  <MCOperand Reg:0>
                            #  <MCOperand Imm:-4>
                            #  <MCOperand Reg:0>
                            #  <MCOperand Imm:0>>
                ….
```

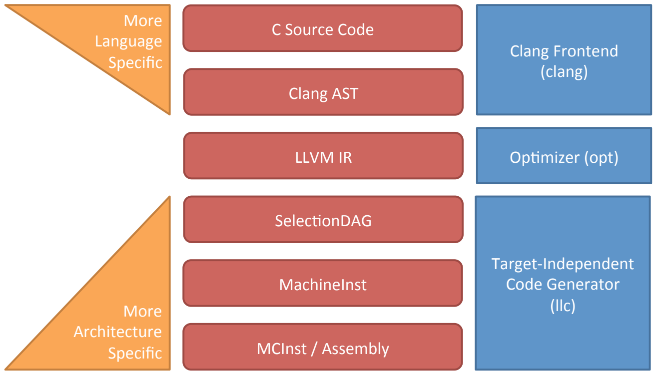
Carnegie Mellon

Assembly

```
main:                       # @main
# BB#0:                     # %entry
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
        movl    $0, -4(%ebp)
        movl    $5, -8(%ebp)
        movl    $3, -12(%ebp)
        movl    -8(%ebp), %eax
        addl    $-3, %eax
        addl    $12, %esp
        popl    %ebp
        retl
```

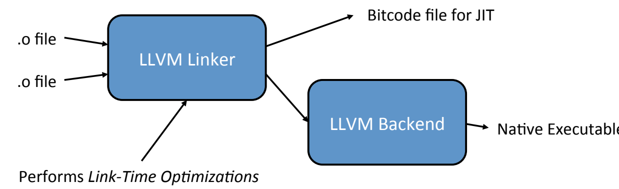
Carnegie Mellon

LLVM: From Source to Binary



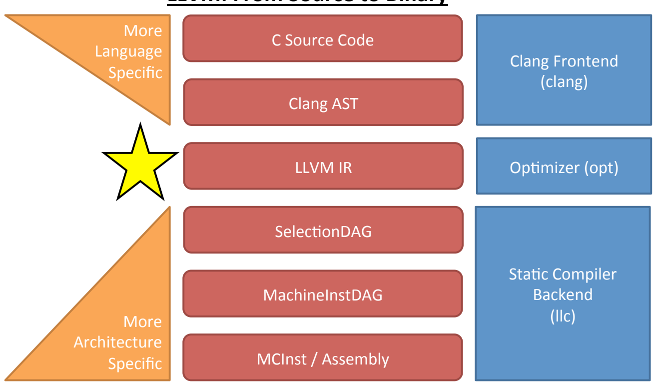
Carnegie Mellon

Linking and Link-Time Optimization



Carnegie Mellon

LLVM: From Source to Binary



Carnegie Mellon

Goals of LLVM Intermediate Representation (IR)

- Easy to produce, understand, and define
- Language- and Target-Independent
- One IR for analysis and optimization
- Supports high- *and* low-level optimization
- Optimize as much as early as possible

Carnegie Mellon

LLVM Instruction Set Overview

- Low-level and target-independent semantics
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers

```
for (i = 0; i < N; i++)
        Sum(&A[i], &P);
```

```
loop:                                    ; preds = %bb0, %loop
%i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
%AiAddr = getelementptr float* %A, i32 %i.1
call void @Sum(float %AiAddr, %pair* %P)
%i.2 = add i32 %i.1, 1
%exitcond = icmp eq i32 %i.1, %N
br i1 %exitcond, label %outloop, label %loop
```


LLVM Instruction Set Overview (continued)

- High-level information exposed in the code
  - Explicit dataflow through SSA form (more later in the class)
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
  - Preserves array subscript and structure indexing

```
for (i = 0; i < N; i++)
        Sum(&A[i], &P);
```

```
loop:                                    ; preds = %bb0, %loop
%i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
%AiAddr = getelementptr float* %A, i32 %i.1
call void @Sum(float %AiAddr, %pair* %P)
%i.2 = add i32 %i.1, 1
%exitcond = icmp eq i32 %i.1, %N
br i1 %exitcond, label %outloop, label %loop
```

Nice syntax for calls is preserved

Lowering Source-Level Types to LLVM

- Source language types are *lowered*:
  - Rich type systems expanded to **simple types**
  - Implicit & abstract types are made explicit & concrete
- Examples of lowering:
  - Reference turn into pointers: T& -> T*
  - Complex numbers: complex flood -> {float, float}
  - Bitfields: struct X { int Y:4; int Z:2; } -> { i32 }
- The entire type system consists of:
  - **Primitives**: label, void, float, integer, …
    - Arbitrary bitwidth integers (i1, i32, i64, i1942652)
  - **Derived**: pointer, array, structure, function (unions get turned into casts)
  - No high-level types
- Type system allows arbitrary casts

Example Function in LLVM IR

```
int main() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

clang

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

Explicit stack allocation

Explicit Loads and Stores

Explicit Types

Example Function in LLVM IR

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

mem2reg →

```
define i32 @main() #0 {
entry:
  %sub = sub nsw i32 5, 3
  ret i32 %sub
}
```

Not always possible: Sometimes stack operations are too complex

Carnegie Mellon

LLVM Instruction Hierarchy

Carnegie Mellon

LLVM Instructions <--> Values

```
int main() {
  int x;
  int y = 2;
  int z = 3;
  x = y+z;
  y = x+z;
  z = x+y;
}
```

clang + mem2reg →

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

Instruction I: %add1 = add nsw i32 %add, 3

Operand 1

Operand 2

You can't "get" %add1 from Instruction I.
Instruction serves as the Value %add1.

Carnegie Mellon

LLVM Instructions <--> Values

```
int main() {
  int x = 1;
  int y = 2;
  int z = 3;
  x = y+z;
  y = x+z;
  z = x+y;
}
```

clang + mem2reg →

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

Instruction I: %add1 = add nsw i32 %add, 3

outs() << *(I.getOperand(0)); → "%add = add nsw i32 2, 3"

outs() << *(I.getOperand(0)->getOperand(0)); → "2"

Only makes sense for an SSA Compiler [More on this Later]

Carnegie Mellon

LLVM Instructions <--> Values

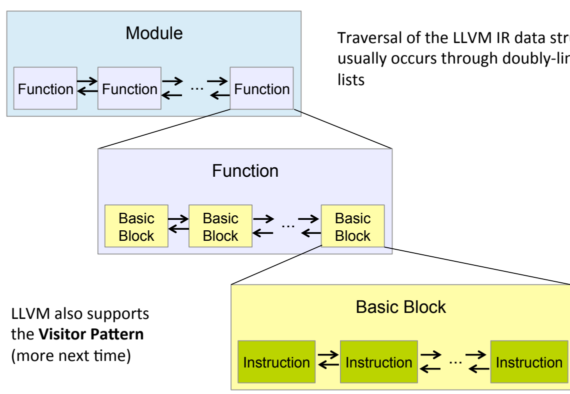
```
int main() {
    int x = 1;
    int y = 2;
    int z = 3;
    x = y+z;
    y = x+z;
    z = x+y;
}
```

clang + no m2r

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %z = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 1, i32* %x, align 4
  store i32 2, i32* %y, align 4
  store i32 3, i32* %z, align 4
  %0 = load i32* %y, align 4
  %1 = load i32* %z, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %x, align 4
  …
```


LLVM Program Structure

- **Module contains Functions and GlobalVariables**
  - Module is unit of compilation, analysis, and optimization
- **Function contains BasicBlocks and 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**



Traversal of the LLVM IR data structure usually occurs through doubly-linked lists

LLVM also supports the **Visitor Pattern** (more next time)

LLVM Pass Manager

- **Compiler is organized as a series of "passes":**
  - Each pass is one analysis or transformation
- **Four types of passes:**
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time
- **Constraints imposed (e.g. FunctionPass):**
  - FunctionPass can only look at "current function"
  - Cannot maintain state across functions

LLVM Tools

- Basic LLVM Tools
  - llvm-dis: Convert from .bc (IR binary) to .ll (human-readable IR text)
  - llvm-as: Convert from .ll (human-readable IR text) to .bc (IR binary)
  - opt: LLVM optimizer
  - llc: LLVM static compiler
  - llvm-link - LLVM bitcode linker
  - llvm-ar - LLVM archiver
- Some Additional Tools
  - bugpoint - automatic test case reduction tool
  - llvm-extract - extract a function from an LLVM module
  - llvm-bcanalyzer - LLVM bitcode analyzer
  - FileCheck - Flexible pattern matching file verifier
  - tblgen - Target Description To C++ Code Generator

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:**

–When you write a pass, you must write the registration

```
RegisterPass<FunctionInfo> X("function-info",
    "15745: Function Information");
```