LLVM Tutorial

Introduction

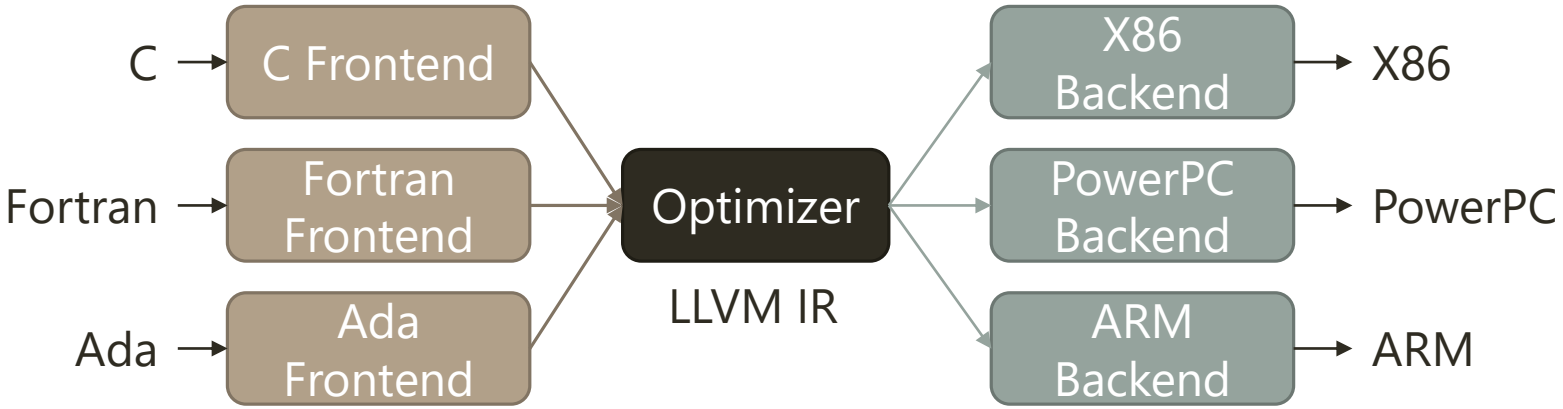
2019. 04. 10

Course Information

- Content
  - How to implement front-end compilation
  - How to implement IR optimization
  - How to implement back-end compilation
- Based on LLVM 8.0.0
- Reference
  - Mayur Pandey and Suyog Sarda, LLVM Cookbook
  - https://llvm.org/docs/tutorial/
  - https://llvm.org/docs/WritingAnLLVMBackend.html

LLVM

- **L**ow **L**evel **V**irtual **M**achine
- Compiler Infrastructure
  - Source- and target-independent code generation

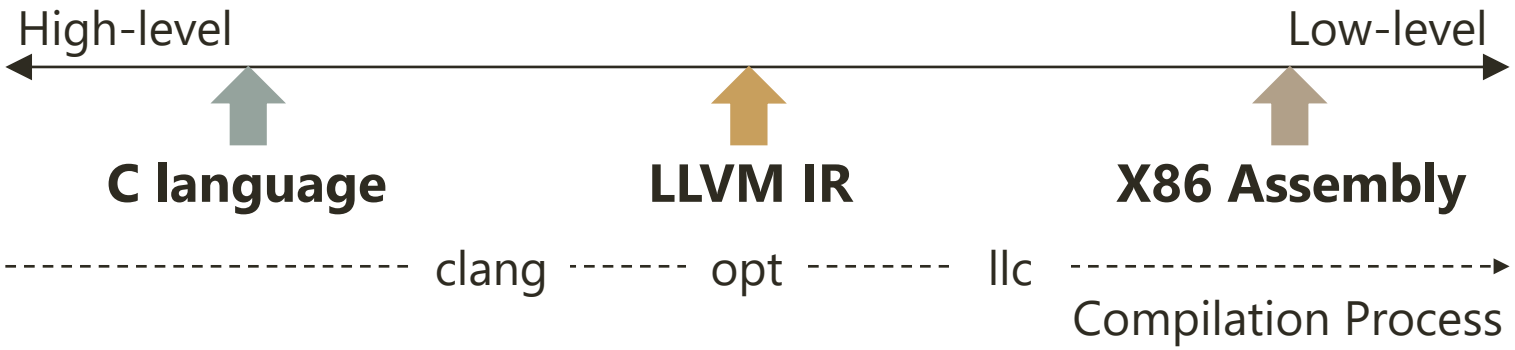


LLVM

- (Mostly) Written in C++
- The LLVM project includes
  - Core libraries
    - Code optimizer (opt)
    - Code generation (llc)
  - Clang (≈gcc)
    - Native C/C++/Object-C compiler
  - LLDB (≈ gdb)
    - Native debugger

LLVM IR

- IR = Intermediate Representation

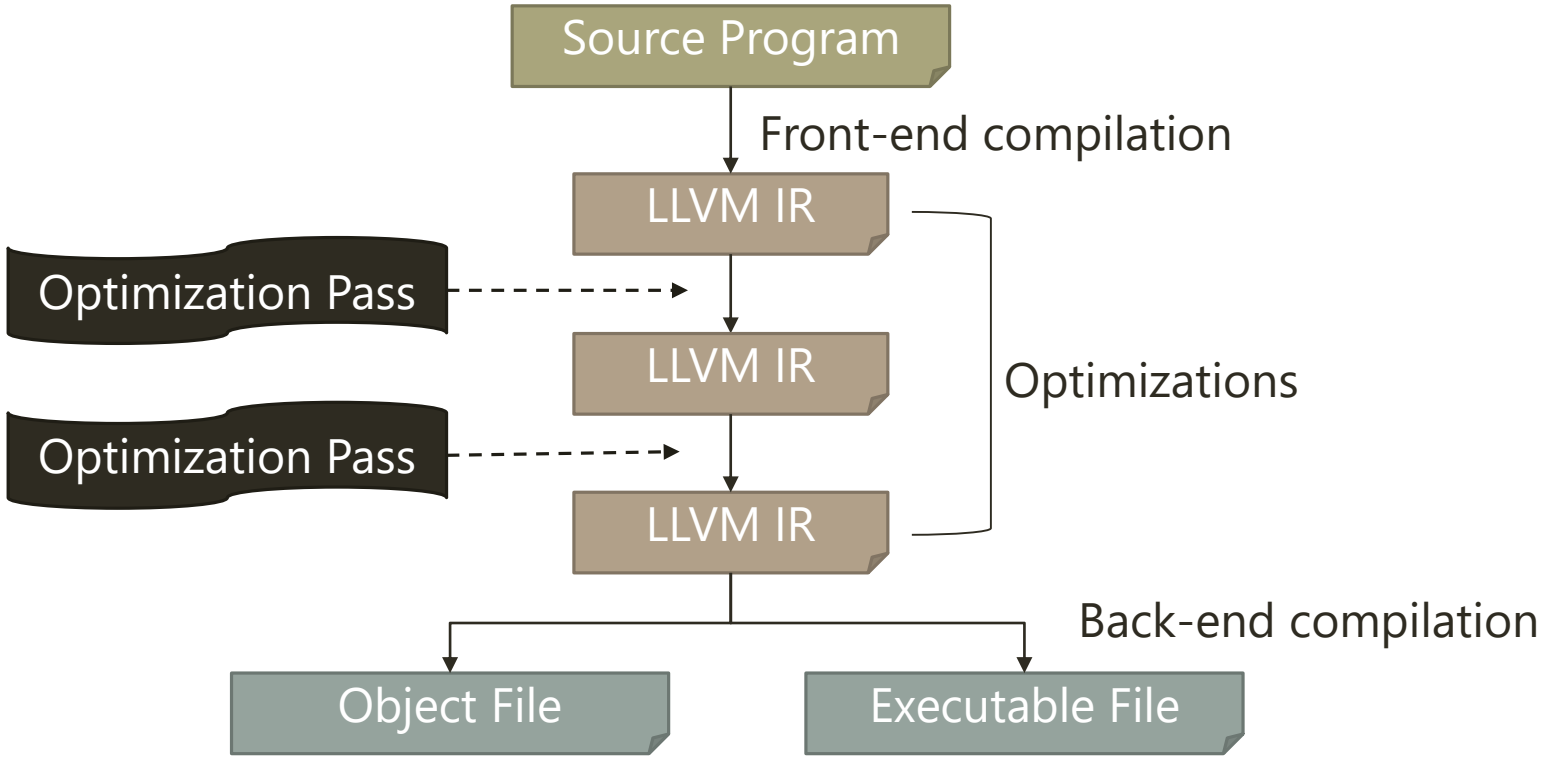


- File Extensions
  - *.bc: Bitcode IR
  - *.ll: Human-readable IR

`llvm-dis *.bc`

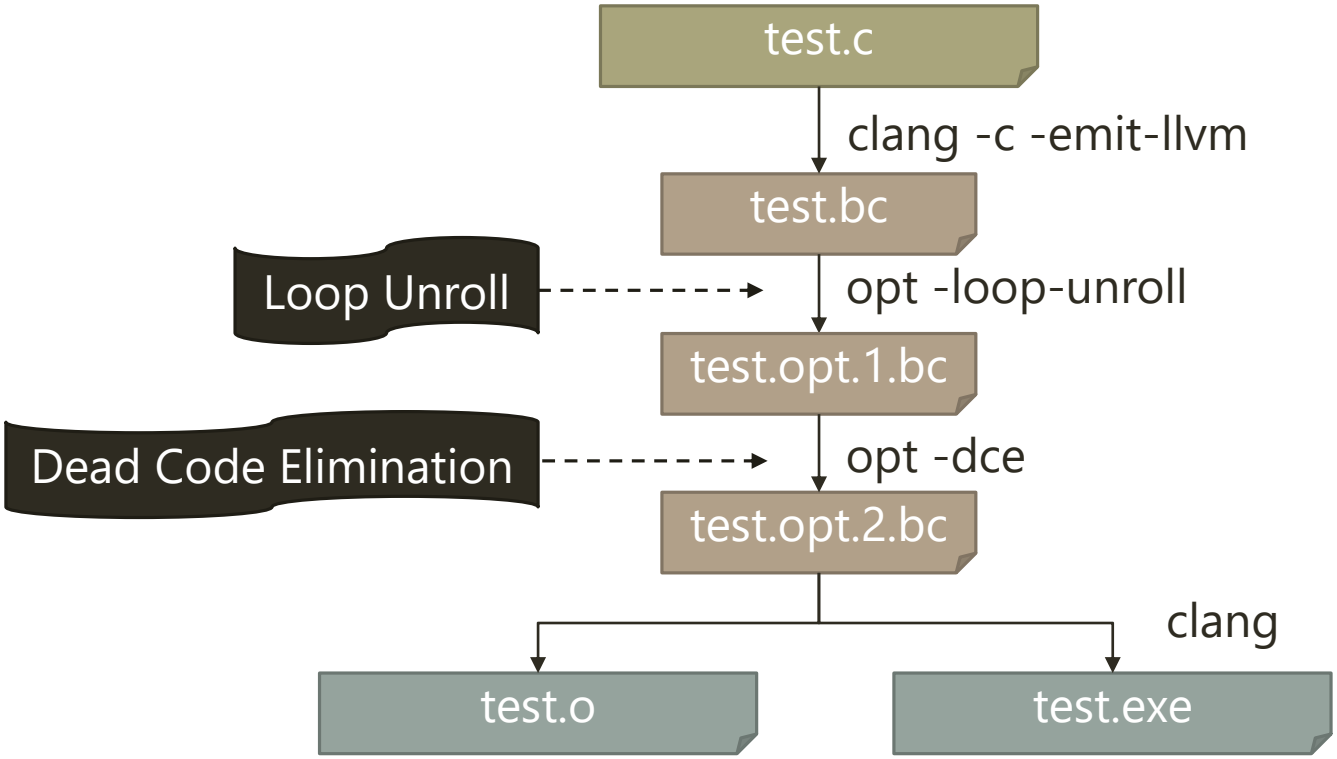
Compilation Process

- General compilation process



Compilation Process

- Example (Diagram)



Compilation Process

- Example (Command Line)
  - Front-end compilation

```
$ clang -c -emit-llvm test.c -o test.bc
```

  - Optimizations

```
$ opt -loop-unroll test.bc -o test.opt.1.bc
$ opt -dce test.opt.1.bc -o test.opt.2.bc
```

  - Back-end compilation

```
$ clang test.opt.2.bc -o test.exe
```

Practice 1: First Compilation

- Goal
  - Learn how to generate and optimize LLVM IR from a C program

- Steps
  1) Write a simple program in C (test.c)
  2) Generate test.bc from test.c
  3) Optimize test.bc with any optimization pass (test.opt.bc)
     - Tip: opt -help to see available passes
  4) Generate test.ll and test.opt.ll from test.bc and test.opt.bc
  5) Generate test.exe from test.opt.bc

LLVM IR

- Example Code

C

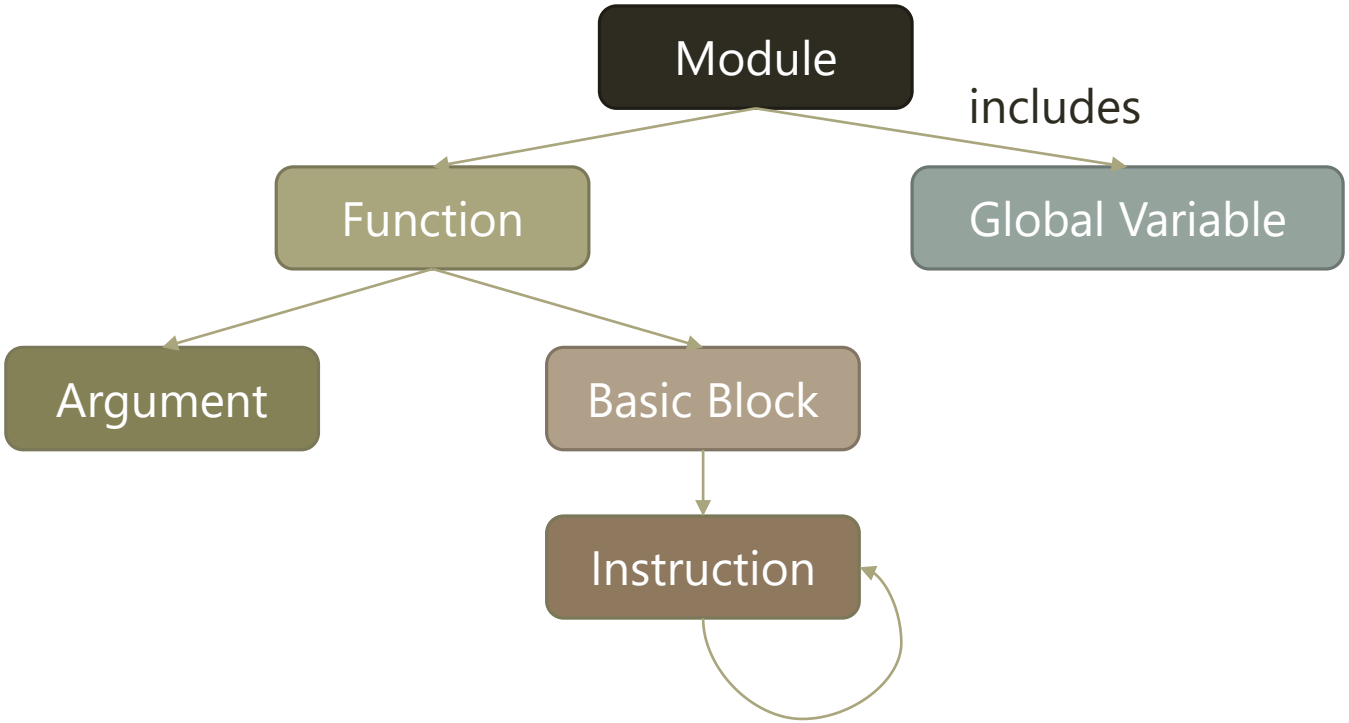
```c
int add (int a, int b) {
  return a+b;
}
```

LLVM IR

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

LLVM IR

- Structure of a program
  - Has-a relationship of objects

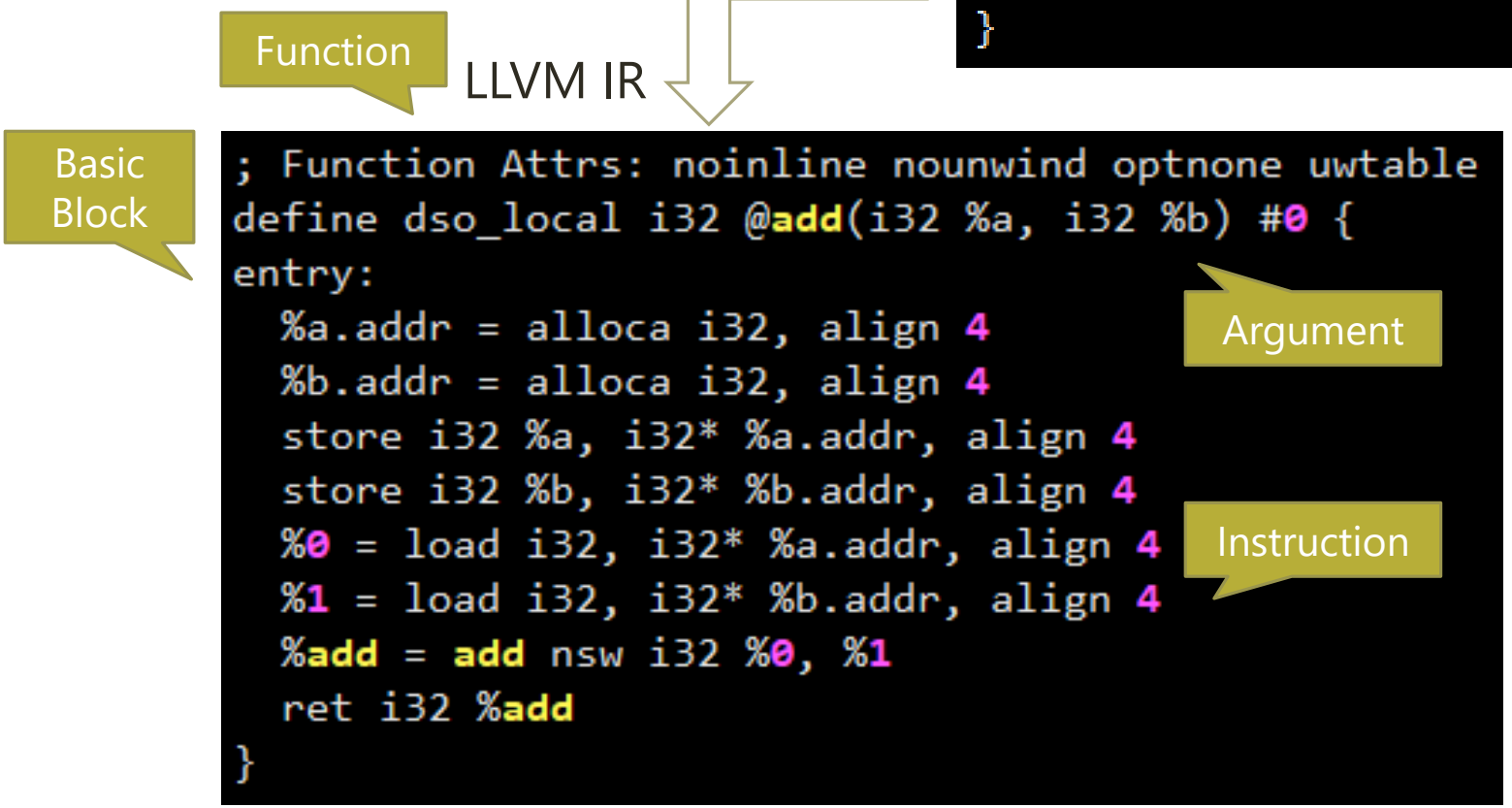


LLVM IR

- Example

C

```c
int add (int a, int b) {
  return a+b;
}
```



LLVM IR

- Features of LLVM IR
  - Strongly typed: No implicit type casting

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

LLVM IR

- Features
  - Single Static Assignment (SSA): No redefinition of value

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

Not $0 nor $1

Single Static Assignment (SSA)

- Every value must be defined **only once**

```
%x = 1 + 2
%x = %x + 3
```

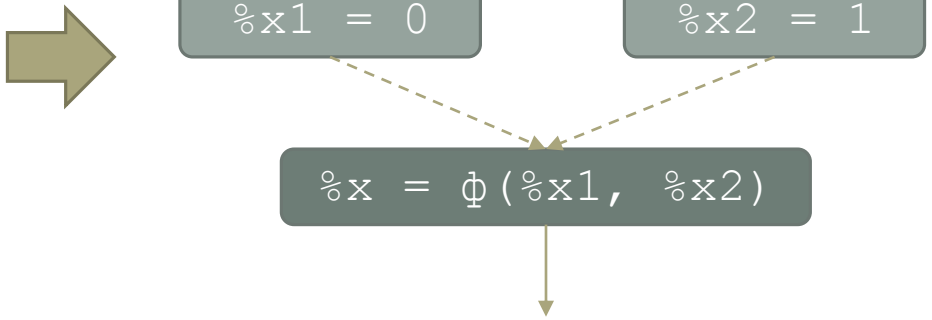
```
%x = 1 + 2
%y = %x + 3
```

- Facilitate program analyses
  - Liveness Analysis: From **DEF** to last **USE**
  - Constant Propagation: If **DEF** is constant, then **USE** is also constant

Single Static Assignment (SSA)

- Phi(Φ) Node
  - Choose a variable according to the control flow
  - Require “remembering” previous basic block

```
if (k == 0) {
  x = 0;
} else {
  x = 1;
}
printf(...,x);
```



LLVM IR

- Type System
  - Void type (void)
  - First Class Types
    - Single Value Types
      - Integer Type
        - i***N***: ***N***-bit integer type
        - ex) i1, i8, i16, i32, i64, …
      - Floating-point Types
        - half (16-bit), float (32-bit), double (64-bit), fp128 (128-bit)
        - x86_fp80 (80-bit), ppc_fp128 (128-bit)

LLVM IR

- Type System
  - First Class Types
    - Pointer Type
      - Format: <type> *
      - ex) [4 x i32] *
    - Vector Type
      - Format: < <# of elements> x <element type> >
      - ex) <4 x i32>, <8 x float>

LLVM IR

- Type System
  - Aggregate Types
    - Array Type
      - Format: [<# of elements> x <element type>]
      - ex) [40 x i32], [3 x [4 x i32]]
    - Structure Type
      - Formats
        - *Normal* struct type: $T1 = type { <type list> }
        - *Packed* struct type: $T2 = type <{ <type list>}>
      - ex) { i32, i32, i32 }, <{ i8, i32 }>

LLVM IR

- Type System
  - Function Type
    - Format: <return type> (<parameter list>)
      - ex)
        - i32 (i32)
        - float (i16, i32*) *
        - i32 (i8*, ...)

Variable argument

LLVM IR

- Instructions
  - Binary Operations
    - add, fadd, sub, fsub…
  - Memory Access and Addressing Operations
    - **alloca**: Allocate memory on the stack frame
      - ex) %ptr = alloca i32
    - load, store
    - **getelementptr:** Get the address of a subelement of an aggregate data structure
      - Pointer dereference
      - Structure member access
      - ex) %iptr = getelementptr [10 x i32], [10 x i32]* @arr, i16 0, i16 0

LLVM IR

- Instructions
  - Terminator Instructions
    - ret**:** Return control flow from a function
    - br: Transfer control flow to a different basic block
    - **invoke**: Transfer control flow to a function (exception handling)
  - Other operations
    - icmp: Compare two integers
    - **phi**: Implement Φ node
    - call: Call a function
- Full reference at https://llvm.org/docs/LangRef.html

LLVM IR

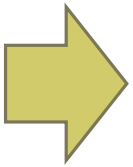
- Q. What does the function *foo* do?

```
; Function Attrs: norecurse nounwind readonly uwtable
define dso_local i32 @foo(i32* nocapture readonly %a) {
entry:
  %0 = load i32, i32* %a, align 4, !tbaa !2
  %arrayidx1 = getelementptr inbounds i32, i32* %a, i64 1
  %1 = load i32, i32* %arrayidx1, align 4, !tbaa !2
  %cmp = icmp eq i32 %0, %1
  %spec.store.select = zext i1 %cmp to i32
  ret i32 %spec.store.select
}
```

LLVM IR

- Full LLVM IR Code

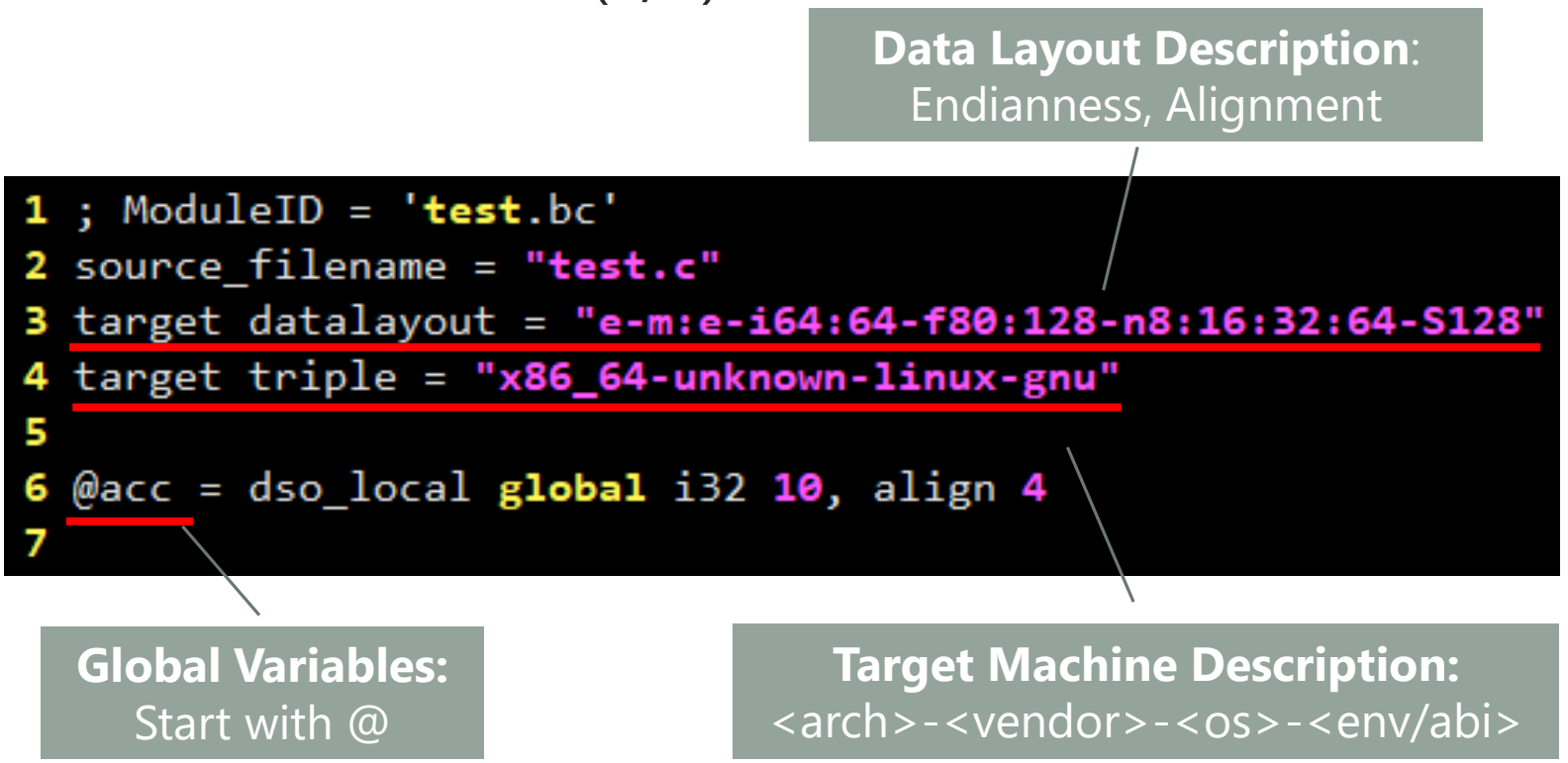
```c
int acc = 10;

int add (int a, int b) {
  return a + b + acc;
}
```



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

@acc = dso_local global i32 10, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  %2 = load i32, i32* @acc, align 4
  %add1 = add nsw i32 %add, %2
  ret i32 %add1
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 8.0.0 (git@git.corelab.or.kr:corelab/clang.git 7973f6c2602b1e37f00a710ffa0c798a3f321e58) (git@git.corelab.or.kr:corelab/llvm.git c55bcb2f96806a3d9e5718497cede4665b27c8a4)"}
```

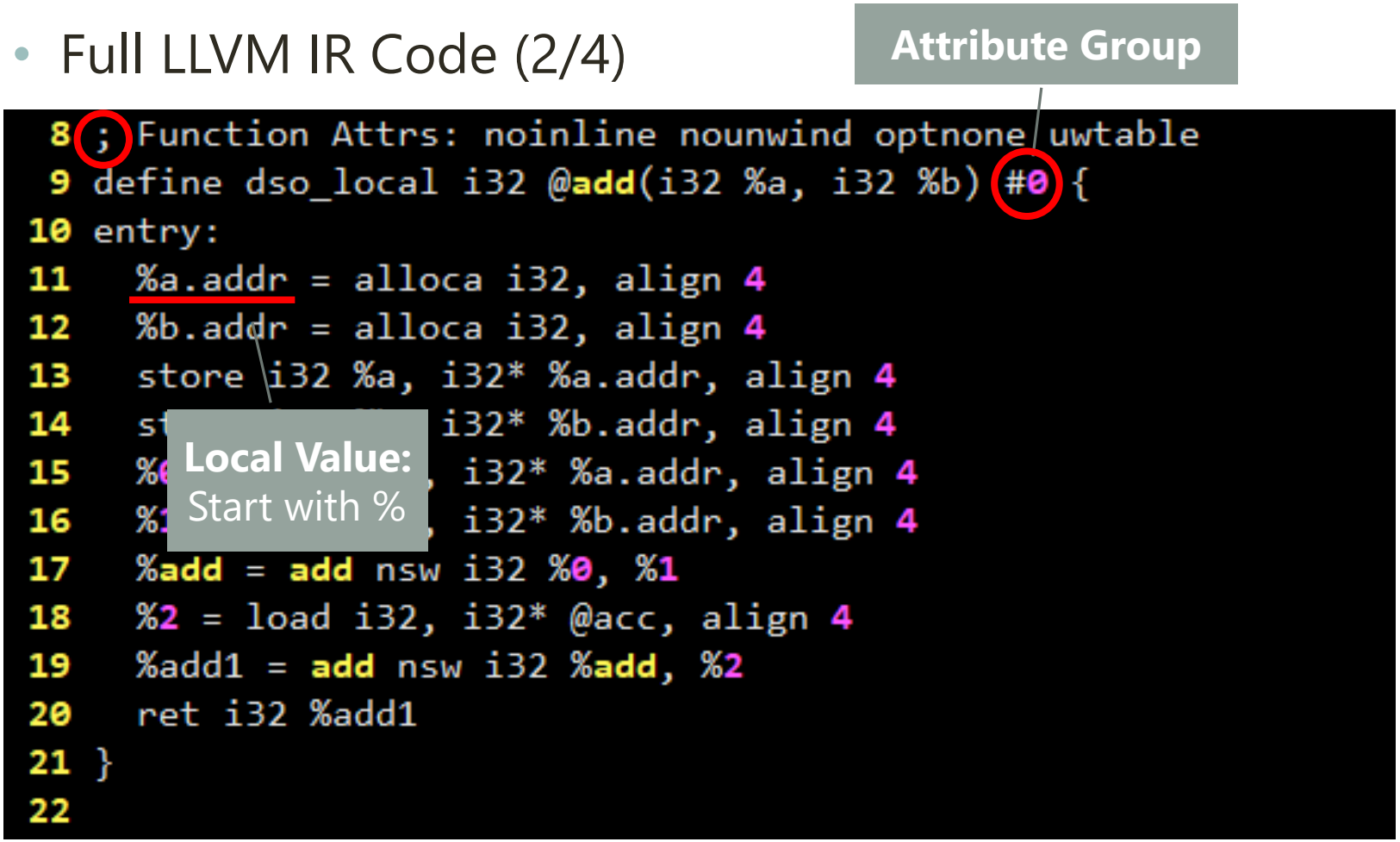
LLVM IR

- Full LLVM IR Code (1/4)



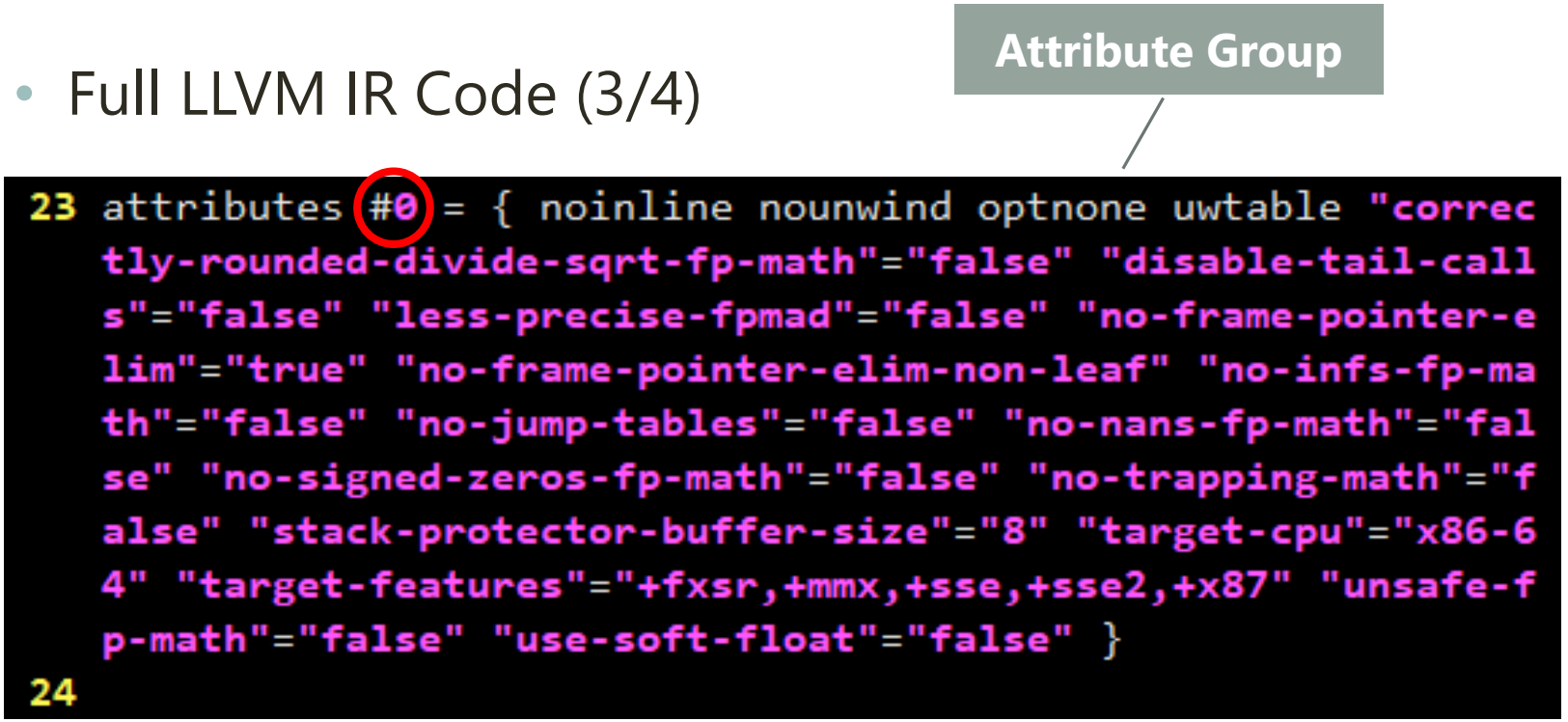
LLVM IR

- Full LLVM IR Code (2/4)



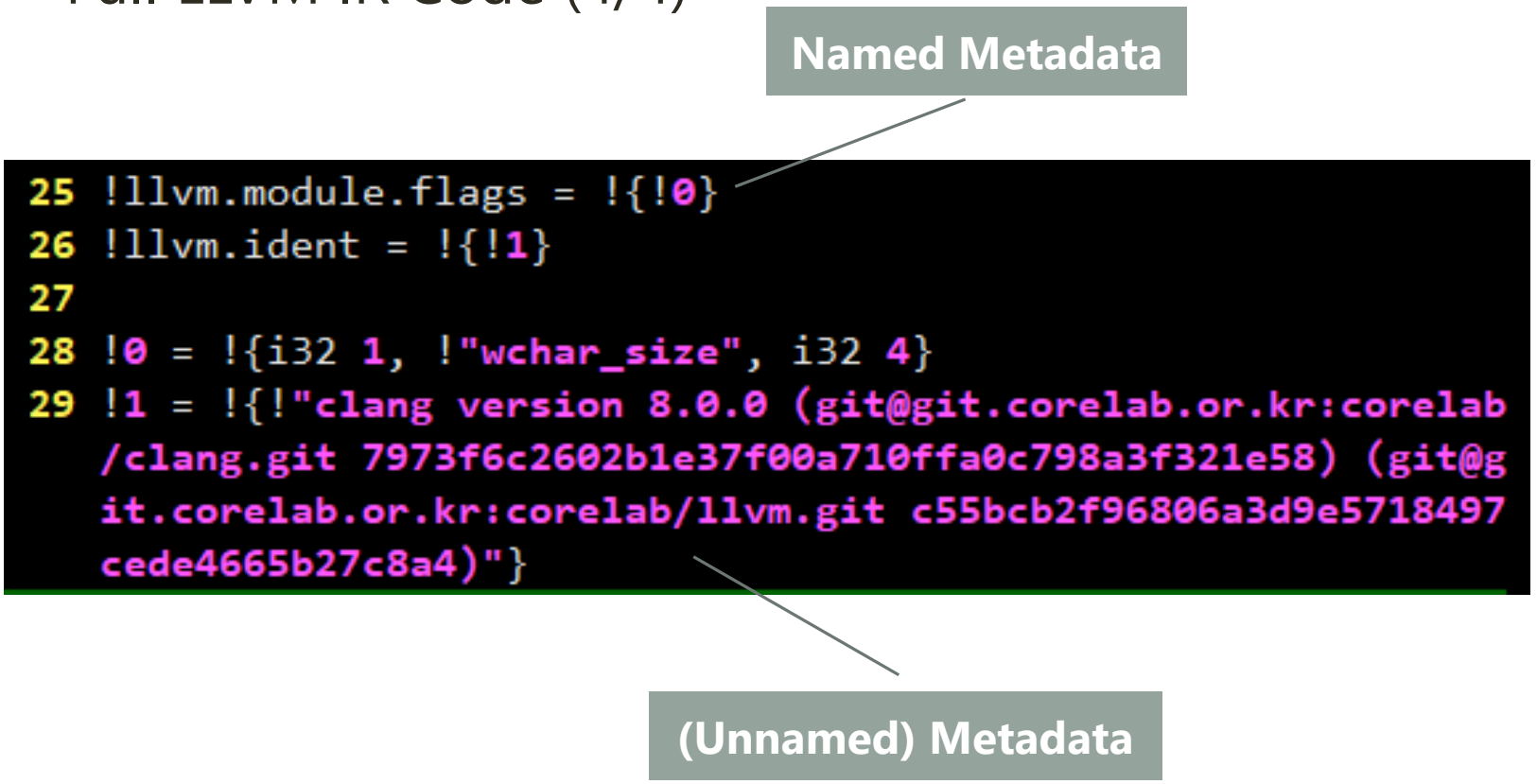
LLVM IR

- Full LLVM IR Code (3/4)



LLVM IR

- Full LLVM IR Code (4/4)



Hand Optimization

- Hand optimization is sometimes needed
  - No available optimization pass
  - Want to know expected speedup

- Modify a .ll file with a text editor, then compile

```
$ clang test.ll -o test.exe
```

Hand Optimization

- Example

add.c

```c
int add (int a, int b) {
  return a + b;
}

int main () {
  int c = add(10, 30);
  printf("%d\n", c);
  return 0;
}
```

add.ll

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  %add2 = add nsw i32 %add, 1
  ret i32 %add2
}
```

Hand Optimization

- Common mistakes
  - Violate Single Static Assignment (SSA)
    - IR code

```
%add = add nsw i32 %0, %1
%add = add nsw i32 %add, 1
ret i32 %add
```

    - Error message

```
add.ll:18:3: error: multiple definition of local value named 'add'
  %add = add nsw i32 %add, 1
  ^
1 error generated.
```

Hand Optimization

- Common mistakes
  - Violate instruction numbering policy
    - IR code

```
%0 = load i32, i32* %a.addr, align 4
%1 = load i32, i32* %b.addr, align 4
%add = add nsw i32 %0, %1
%3 = add nsw i32 %add, 1
ret i32 %3
```

    - Error message

```
add.ll:18:3: error: instruction expected to be numbered '%2'
  %3 = add nsw i32 %add, 1
  ^
1 error generated.
```

Practice 2: 2MM Optimization

- Goal
  - Get used to LLVM IR by optimizing the code by hand

- Steps

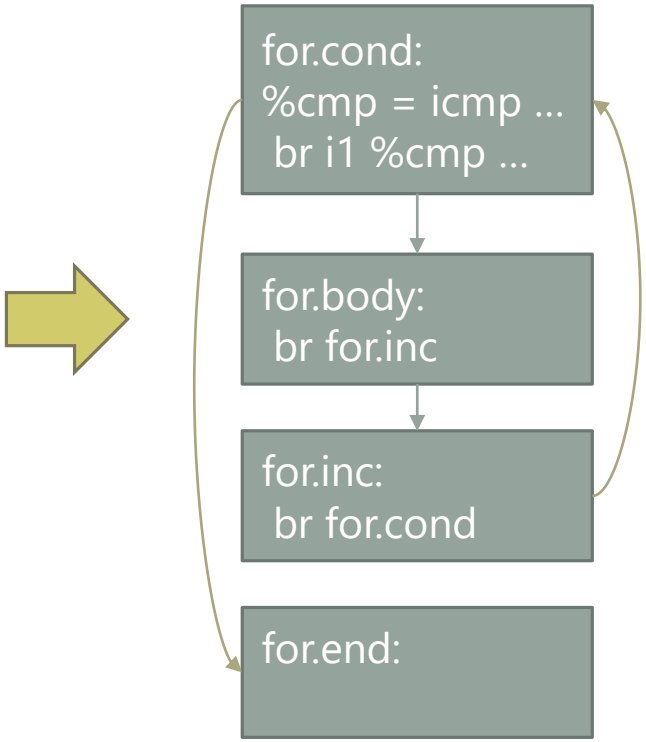
Optimization level

1) Compile a matrix-multiplication program with -O0 option
   - Target program: tutorial/basic/mm.c → mm.ll
2) Copy mm.ll as mm.opt.ll and apply **LICM** by hand
3) Generate mm.exe and mm.opt.exe from mm.ll and mm.opt.ll
4) Compare the performance
   - Tip: time -e

Practice 2: 2MM Optimization

- Structure of a loop
  - (Preheader), Header, Body, Exit

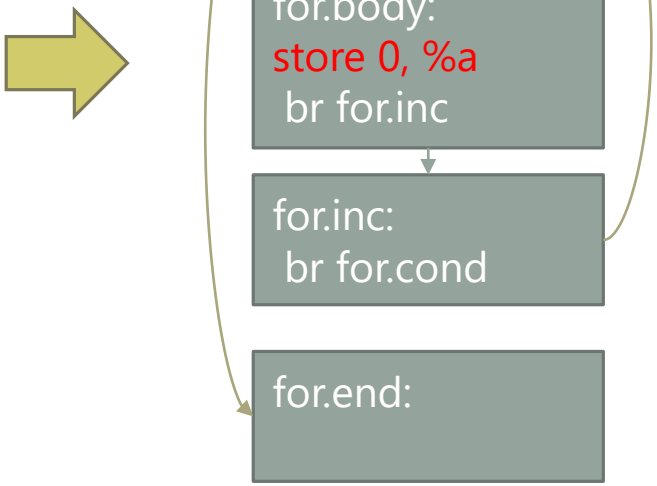
```
for (i = 0; i < 100; i++)
{
  // Body
}
```



Practice 2: 2MM Optimization

- LICM (Loop-Invariant Code Motion)
  - Loop-invariant code: Value does not change in iteration

```
for (i = 0; i < 100; i++)
{
  a = 1;
  b += a;
}
```



Practice 2: 2MM Optimization

- Q1. Performance improved?
- Q2. Any possible optimization?

Project Structure

- LLVM Core Libraries
  - Folders in **include/llvm**
    - **IR:** IR types and classes
      - Type.h
      - Module.h
      - Function.h
    - **Analysis:** LLVM IR analysis passes
      - AliasAnalysis.h
      - CallGraph.h
      - CFG.h (Control Flow Graph)

Project Structure

- LLVM Core Libraries
  - Folders in **include/llvm**
    - **Transforms**
      - **Scalar**
        - LoopRotation.h
        - IndVarSimplify.h
        - LICM.h (Loop Invariant Code Motion)
      - **Utils**
        - Cloning.h
        - Mem2Reg.h
- Full reference at http://llvm.org/doxygen/

Thank you!

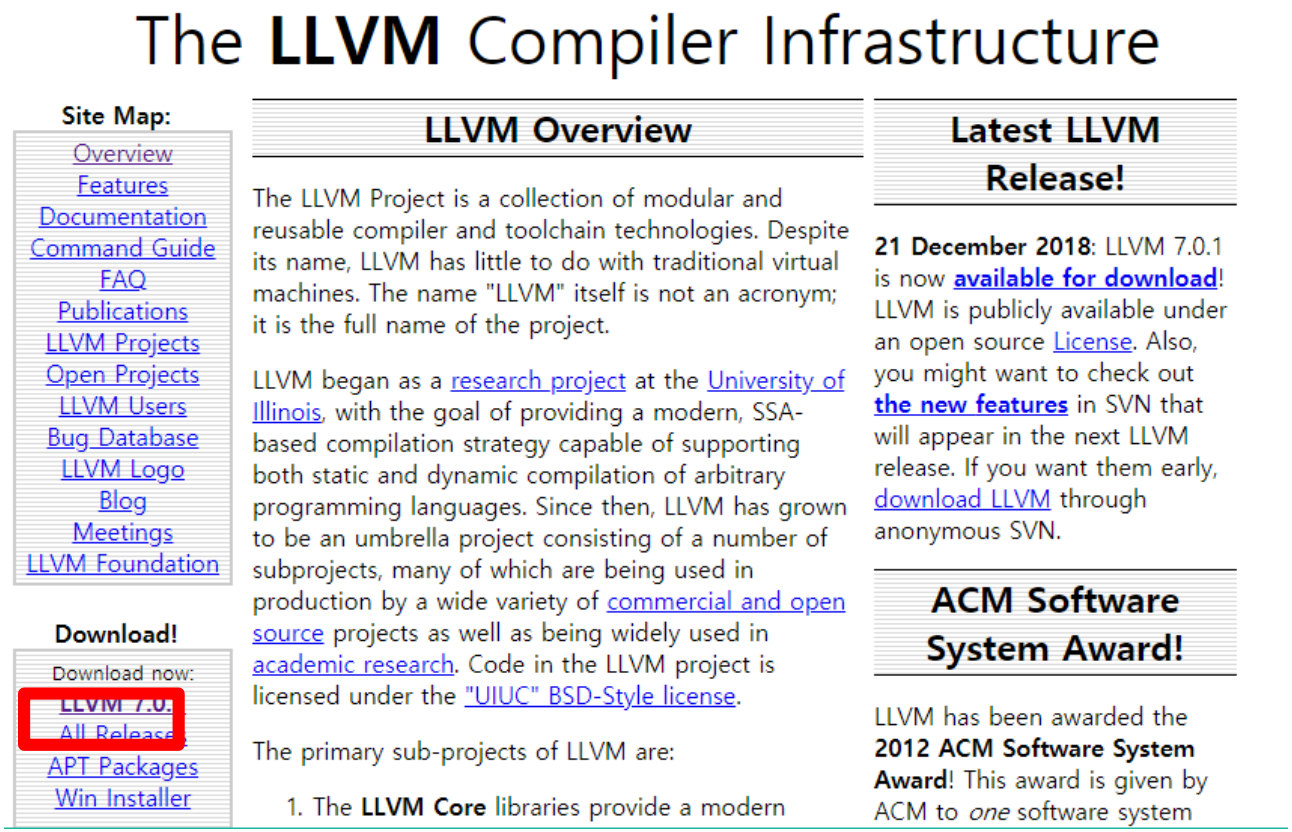
허선영

heosy@postech.ac.kr

Backup Slides

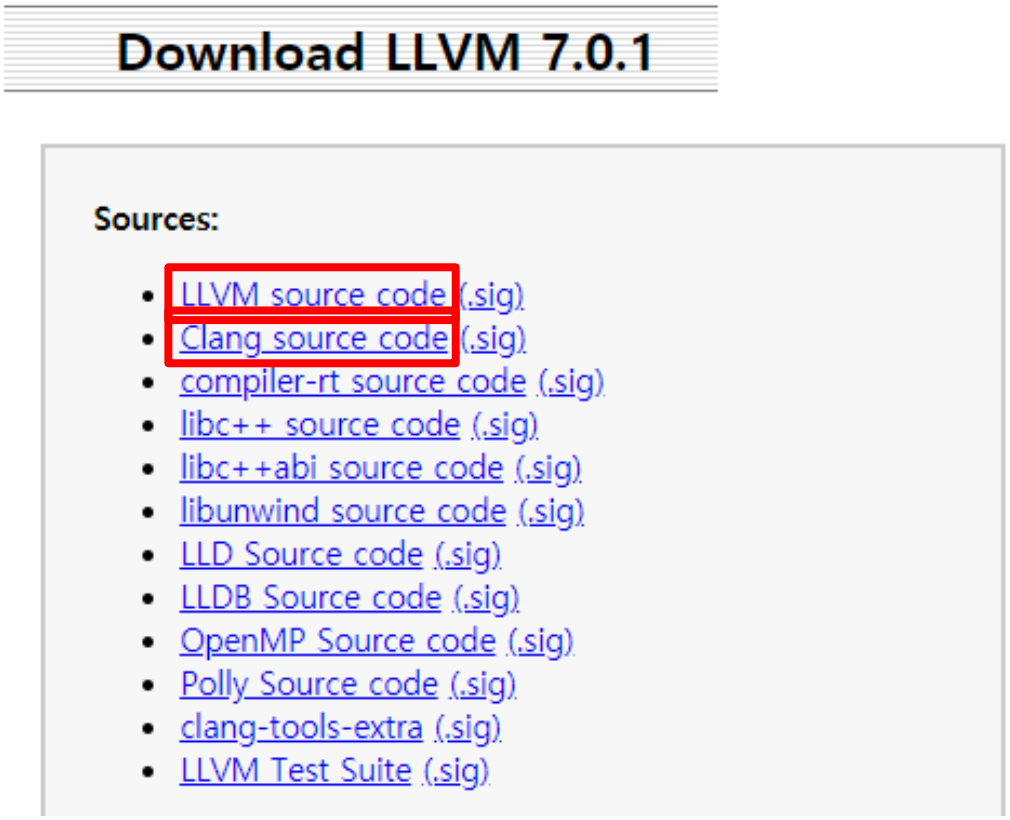
Practice 0: Install LLVM

1. Visit https://llvm.org/



Practice 0: Install LLVM

2. Download LLVM and Clang source codes



Practice 0: Install LLVM

3. Extract the LLVM source code

```
$ tar xf llvm-7.0.1.src.tar.xz
$ mv llvm-7.0.1.src llvm
```

4. Extract the Clang source code at **tools**

```
$ cd llvm/tools
$ tar xf cfe-7.0.1.src.tar.xz
$ mv cfe-7.0.1.src clang
```

Practice 0: Install LLVM

5. Build LLVM at **llvm-objects**

```
$ cd ../..
$ mkdir llvm-objects
$ cmake ../llvm && make
```

make -j4 for multi-threading

6. Install LLVM at **llvm-install**

```
$ mkdir ../llvm-install
$ cmake -DCMAKE_INSTALL_PREFIX=../llvm-
install -P cmake_install.cmake
```

Practice 0: Install LLVM

7. Add the install path to $PATH

```
$ export PATH=$PATH:$PATH_TO_LLVM_INSTALL/bin
```

LLVM IR

- Hierarchy of LLVM IR
  - Is-A relationship of IR classes