Abstract (Last updated 6/03/18)

Abstract: In this talk, Michael Shah ("Mike") will be presenting an introduction to the LLVM Compiler Infrastructure. A discussion of what LLVM is, who is using it, and why you might be interested in using LLVM will be presented during the first part of the talk. The second part of the talk will show interactive examples, taking us through installation to the point where we build and run our first function pass. We will build on top of our first function pass, to begin outputting some program metrics about programs. Mike will also be presenting some steps on how to proceed further and what resources are available for working with LLVM.

Materials:

- Please bring a laptop with LLVM 8.0 setup if you want to follow along
- Otherwise materials will be posted to www.mshah.io

Resources:

- Downloading and setting up LLVM: http://llvm.org/docs/GettingStarted.html#checkout
- A really good introduction guide: http://adriansampson.net/blog/llvm.html

Contact: mshah.475@gmail.com
Twitter: @MichaelShah

Terminology (Open in a second browser if you like)

- LLVM - The name of the project (not an acronym)
- IR - Intermediate representation (Human-readable, 3 address, assembly like representation)
- Bitcode (.bc) - LLVM binary format of the IR
- JIT - Just-In-Time Compiler
- SSA - Single Static Analysis

Introduction to LLVM (Tutorial)

Mike Shah, Ph.D.
@MichaelShah | mshah.io

June 3, 2019

60-75 Minutes for talk (plenty of time for questions)

Demo Time! Right from the start!

- So you know what to pay attention to!
  - In case you (or maybe I) walked into the wrong room by accident!
  - (Or if you are deciding to commit to an hour long talk online in the *distant future*)
- For those attending this talk live
  - Take a moment to introduce yourself to someone next to you .

- demo1.sh - Print functions from program
- demo2.sh - Print out stats
- demo3.sh - Print out direct function callees
- demo4.sh - Instrument code

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

This is an introduction to LLVM

We have some specific goals

1. Figure out what is LLVM
2. Understand how to obtain LLVM
   a. (This can be a major bottleneck for students)
3. Do a little example with clang++
4. Understand how to produce the demos I have already shown

Goals for Tomorrow (1/2)

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future

Goals for Tomorrow (2/2)

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future
- Be able to run through these slides again with confidence and excitement!

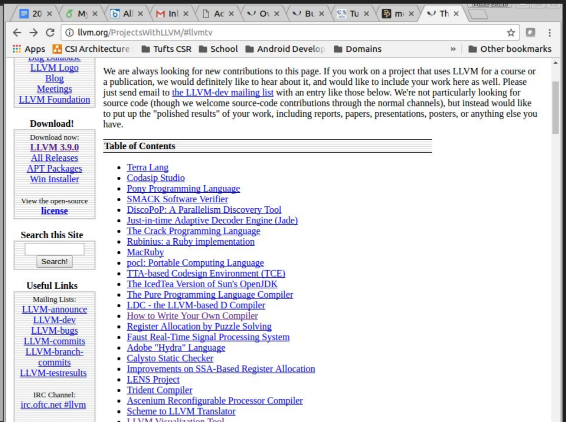
Slides and code are at the following location

www.mshah.io

What is LLVM?

LLVM (Formerly known as Low Level Virtual Machine--but it's more!) (1/2)

- Started at The University of Illinois in 2000.
- Chris Lattner is the lead architect
- Backed by companies like Apple, Google, Microsoft, Intel, and more!
- And of course--open source!

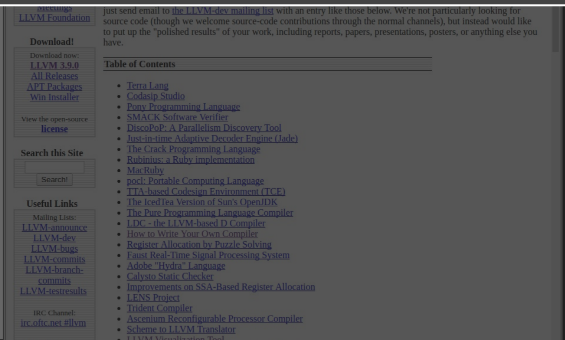


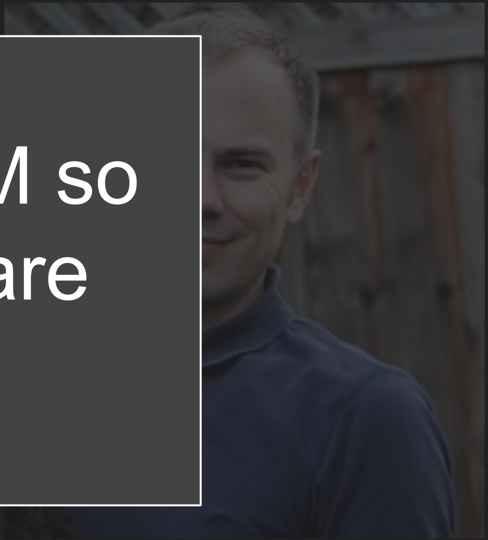
http://nondot.org/sabre/

LLVM (Formerly known as Low Level Virtual Machine--but it's more!) (2/2)

- Started a
- Chris La
- Backed b
  Intel, and
- And of c

What is it that makes LLVM so great that programmers are paying attention to it?



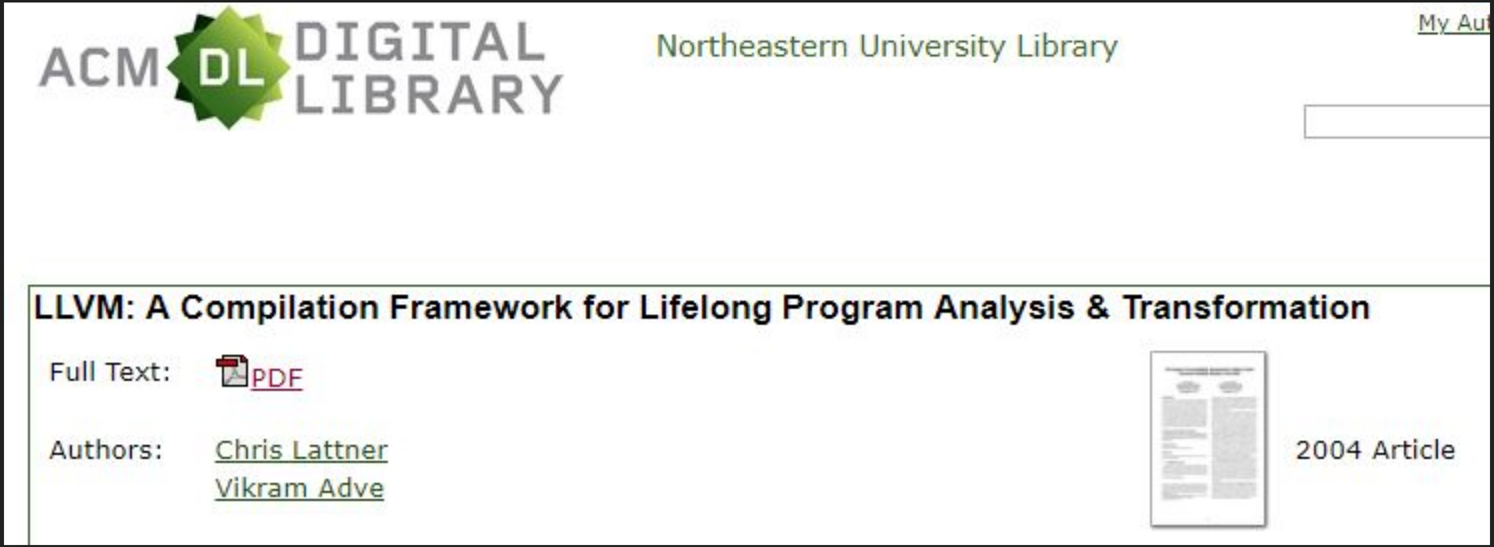


http://nondot.org/sabre/

What is it that makes LLVM so great that programmers are paying attention to it?

The Secret Recipe

- The exact details are listed in the research paper: https://dl.acm.org/citation.cfm?id=977673



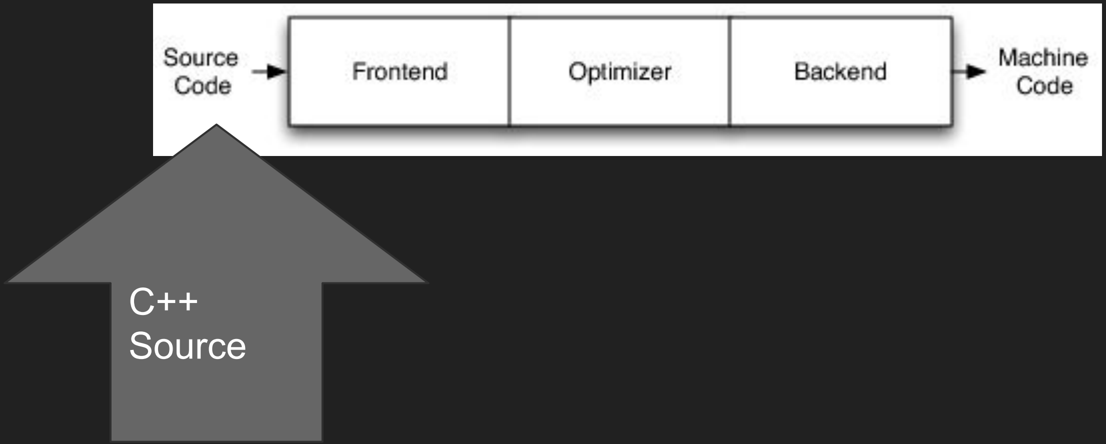
Chris Lattner's big idea (1/6)

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code

sources: LLVM The early Days Developer Meeting talk | AOSA Book

Chris Lattner's big idea (2/6)

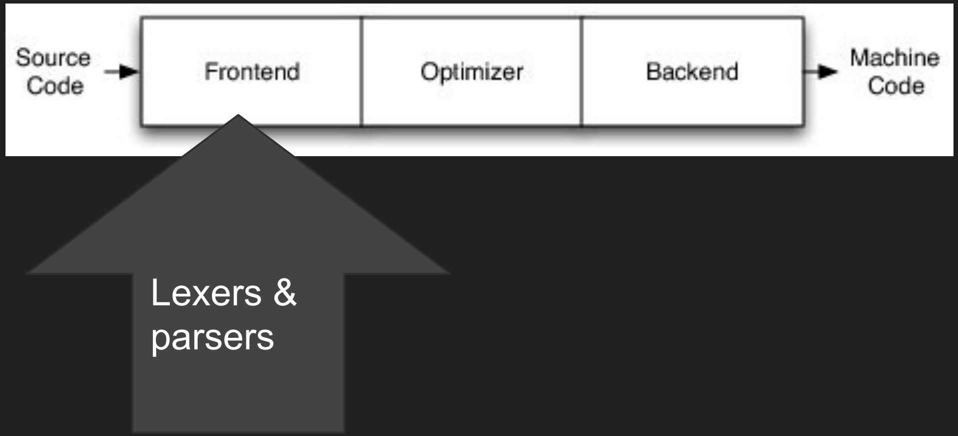
- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



sources: LLVM The early Days Developer Meeting talk | AOSA Book

Chris Lattner's big idea (3/6)

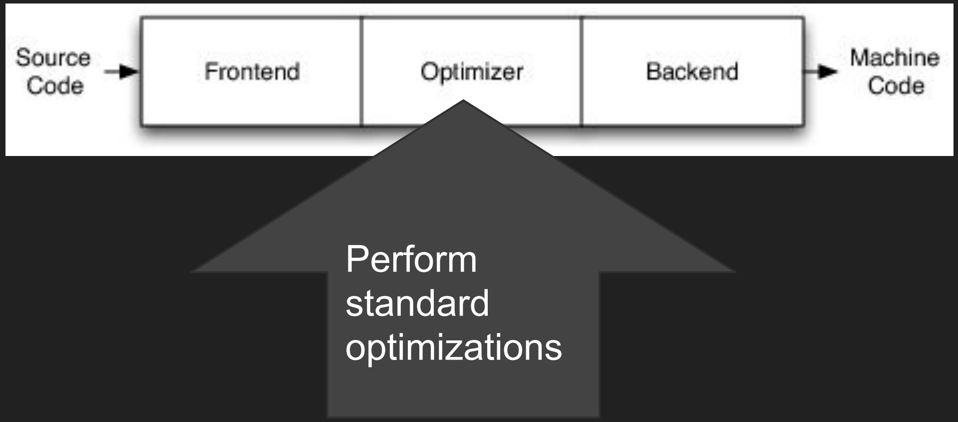
- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



sources: [LLVM The early Days Developer Meeting talk] | [AOSA Book]

Chris Lattner's big idea (4/6)

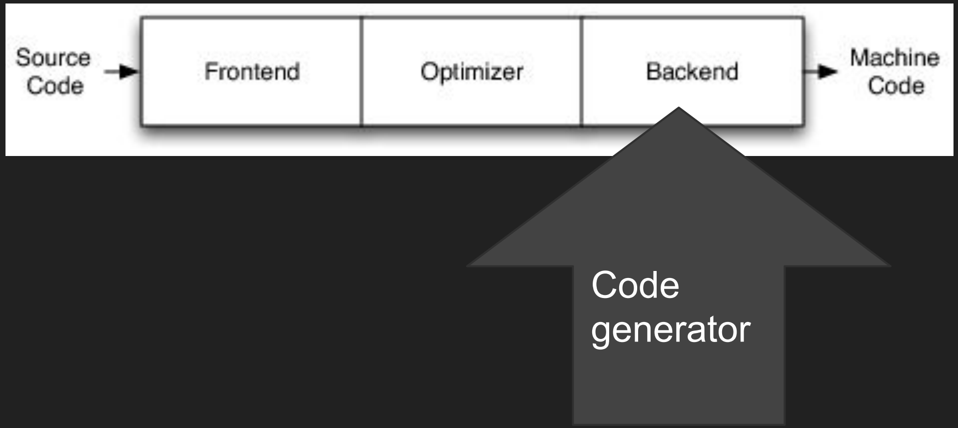
- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



sources: [LLVM The early Days Developer Meeting talk] | [AOSA Book]

Chris Lattner's big idea (5/6)

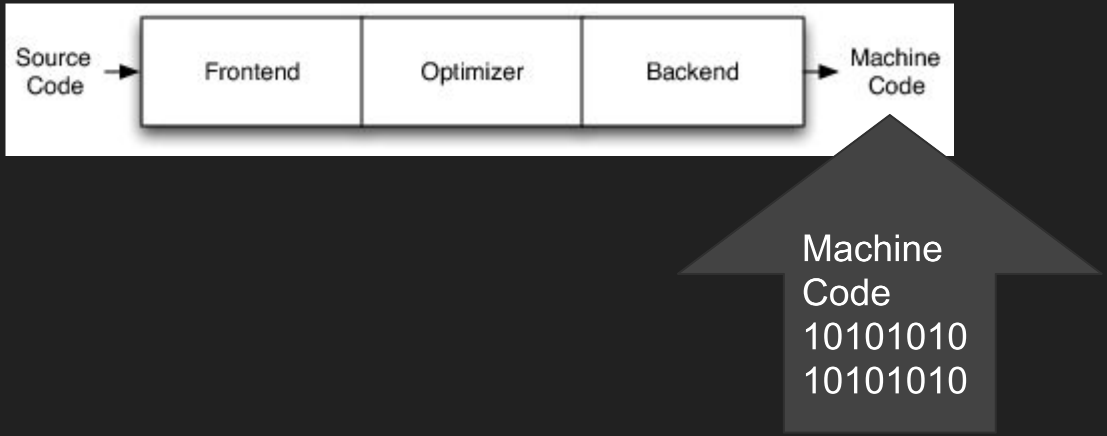
- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



sources: [LLVM The early Days Developer Meeting talk] | [AOSA Book]

Chris Lattner's big idea (6/6)

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
    - Generate a high level language to machine code



sources: LLVM The early Days Developer Meeting talk | AOSA Book

The big idea | Around the year 2000 (1/2)

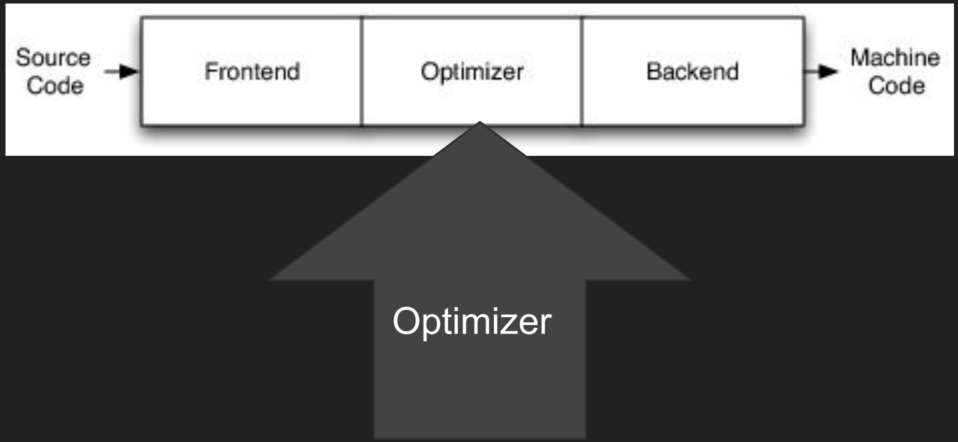
- JIT compilers were and continue to gain traction
  - A virtual machine compiles code online
  - This online compilation means performing optimizations over and over again
- So Lattner et al. big idea was to perform optimizations at compile-time that could do the heavy lifting.
  - Perhaps using some low level virtual machine

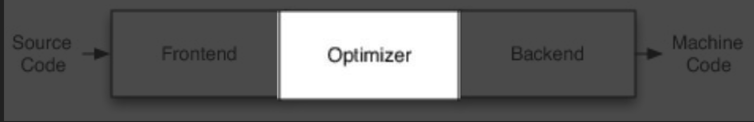
The big idea | Around the year 2000 (2/2)

- JIT compilers were and continue to gain traction
  - A virtual machine compiles code online
  - This online compilation means performing optimizations over and over again
- So Lattner et al. big idea was to perform optimizations at compile-time that could do the heavy lifting.
  - Perhaps using some Low Level Virtual Machine

The Optimizer

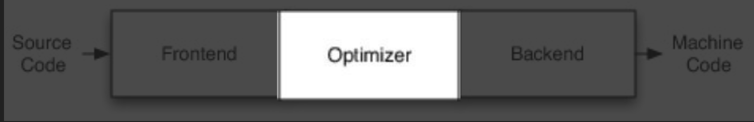
- So in the middle of our compiler pipeline, the optimizer (or optimization of code) is the focus.





The optimization stage of compilers (1/2)

- Typically programs are optimized by manipulating an intermediate representation (IR) of the high level language.
  - The intermediate representation (IR) is more ‘regular’ structurally
    - That means it is easier to analyze and manipulate.
      - (Just think about how many ways you can write and interpret the same program in a high-level language)



The optimization stage of compilers (2/2)

- Typically programs are optimized by manipulating an intermediate representation (IR) of the high level language.
  - The intermediate representation (IR) is more ‘regular’ structurally
    - That means it is easier to analyze and manipulate.
      - (Just think about how many ways you can write and interpret the same program in a high-level language)

Example of what IR instructions look like

```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>                ; Unconditional branch
```

source: https://llvm.org/docs/LangRef.html

How to get LLVM

(And all the tools)

I am actually going to run through this section very quick!

Use it as a reference for how to setup and run examples from this slide deck

The LLVM project evolves at a good pace.

That is why you will want to know how to build from source to get the latest changes.

Where the instructions always will be

- http://llvm.org/docs/GettingStarted.html#checkout

Checkout LLVM from Git

You can also checkout the source code for LLVM from Git. While the LLVM project's official source-code repository is Subversion, we are in the process of migrating to git. We currently recommend that all developers use Git for day-to-day development.

Note

Passing `--config core.autocrlf=false` should not be required in the future after we adjust the .gitattribute settings correctly, but is required for Windows users at the time of this writing.

Simply run:

```
% git clone https://github.com/llvm/llvm-project.git
```

or on Windows,

```
% git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git
```

Checkout via SVN (deprecated)

Until we have fully migrated to Git, you may also get a fresh copy of the code from the official Subversion repository.

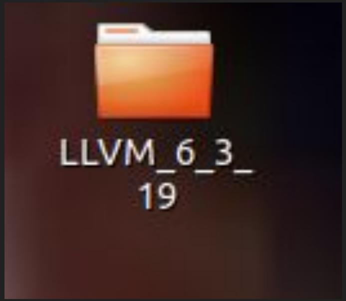
- `cd where-you-want-llvm-to-live`
- Read-Only: `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
- Read-Write: `svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm`

Downloading LLVM 8.0

- For this talk, I am using and have tested the code with LLVM 8.0
- This tutorial is for an x86 based Ubuntu 18 machine
  - A similar process should work on Mac
    - (Windows users may need some different tools, I have not built LLVM on windows)
- Tools you will need
  - svn
  - Cmake
  - Make
  - A C compiler (Mine is GNU 5.4.0)

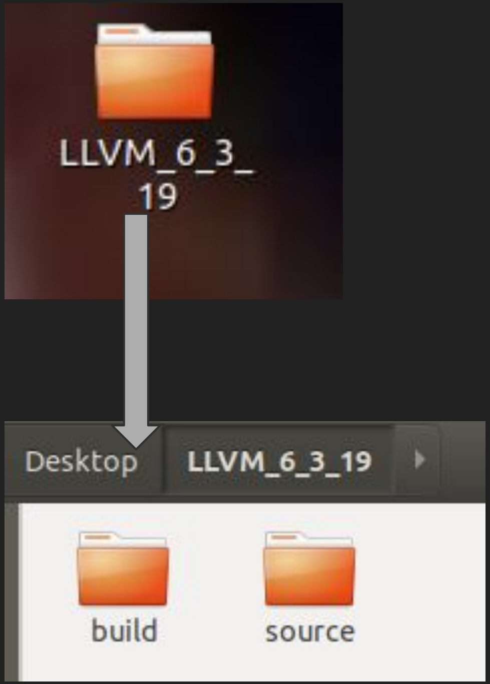
Create a directory on your desktop

- I typically append a date to this directory



Subdirectories

- Within the folder
  - A build directory where our compiled LLVM tools will go
    - (i.e. all the binaries)
  - A source directory where all of the LLVM source files live.



From a Terminal (1/2)

1. cd ~/Desktop/LLVM_6_3_2019/source # cd into your llvm source directory
2. svn co https://user@llvm.org/svn/llvm-project/llvm/tags/RELEASE_800/final llvm
3. cd llvm/tools
4. svn co http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_800/final clang
5. cd clang/tools # (To be clear, you are now in llvm/tools/clang/tools)
6. svn co http://llvm.org/svn/llvm-project/clang-tools-extra/tags/RELEASE_800/final extra
7. cd ../../../../llvm/projects # (To be clear, you are now in llvm/projects)
8. svn co http://llvm.org/svn/llvm-project/compiler-rt/tags/RELEASE_800/final compiler-rt
9. cd ../../.. # (You are now in your desktop directory)
10. mkdir build # (if you have not already done so)
11. cd build (You are now in your build directory)
12. cmake -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_TARGET_ARCH=X86 -DCMAKE_BUILD_TYPE="Release" -G "Unix Makefiles" ../source/llvm/
    - # (alternative to above step if you want to build more examlpes)
    - cmake -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_TARGET_ARCH=X86 -DCMAKE_BUILD_TYPE="Release" -DLLVM_BUILD_EXAMPLES=1 -DCLANG_BUILD_EXAMPLES=1 -G "Unix Makefiles" ../source/llvm/
13. make -j 8 (from within the build directory to start the process)

From a Terminal (2/2)

```
1.  cd ~/Deskt
2.  svn co htt                                         l llvm
3.  cd llvm/to
4.  svn co htt
5.  cd clang/t
6.  svn co htt                                         00/final extra
7.  cd ../../.                                         s)
8.  svn co htt                                         al compiler-rt
9.  cd ../../.
10. mkdir buil
11. cd build (
12. cmake -DLL
    -DCMAKE_BU                                         LES=1 -G "Unix
    Makefiles" ../source/llvm/
13. 'make -j 8' (from within the build directory to start the process)
```

Now get lunch/dinner/breakfast depending on speed of your cpu.

How will we know it worked?

- Check your `build/bin` directory
- It should look something like this
- Note that for the examples, clang++, and other tools are referenced from here!
    - If your system already has clang++ installed from a package manager, it may have a different version!

```
mike:build$ cd bin/
mike:bin$ ls
arcmt-test                  find-all-symbols       llvm-mt
BrainF                      hmaptool               llvm-nm
bugpoint                    HowToUseJIT            llvm-objcopy
BuildingAJIT-Ch1            Kaleidoscope-Ch2       llvm-objdump
BuildingAJIT-Ch3            Kaleidoscope-Ch3       llvm-opt-fuzz
BuildingAJIT-Ch4            Kaleidoscope-Ch4       llvm-opt-repo
BuildingAJIT-Ch5            Kaleidoscope-Ch5       llvm-pdbutil
BuildingAJIT-Ch5-Server     llc                    llvm-PerfectS
c-arcmt-test                lli                    llvm-profdata
c-index-test                lli-child-target       llvm-ranlib
clang                       llvm-ar                llvm-rc
clang++                     llvm-as                llvm-readelf
clang-8                     llvm-bcanalyzer        llvm-readobj
clang-apply-replacements    llvm-cat               llvm-rtdyld
clang-change-namespace      llvm-cfi-verify        llvm-size
clang-check                 llvm-config            llvm-special-
clang-cl                    llvm-cov               llvm-split
clang-cpp                   llvm-c-test            llvm-stress
clangd                      llvm-cvtres            llvm-strings
clang-diff                  llvm-cxxdump           llvm-strip
clangd-indexer              llvm-cxxfilt           llvm-symboliz
clang-doc                   llvm-cxxmap            llvm-tblgen
clang-extdef-mapping        llvm-diff              llvm-undname
```

(Expect ~15-45 or more minutes to build from source depending on your cpu and internet connection)

Assumption: We all have a working LLVM at this point

Our first example | Emitting LLVMs intermediate form (1/3)

- We can output and actually look at LLVM’s intermediate form.
- We are going to use the ‘clang++’ compiler
  - clang and clang++ are frontends for the C/C++ language.
  - The code they generate targets the LLVM intermediate form.
    - Let us try!

Our first example | Emitting LLVMs intermediate form (2/3)

- Here is some code we can use
  - hello.cpp

```cpp
#include <stdio.h>

int main(){
	printf("Bonjour!\n");
	return 0;
}
```

Our first example | Emitting LLVMs intermediate form (3/3)

- Here is some code we can use
  - hello.cpp
- I will be working in my `build/bin` folder in a directory I created called ‘examples’ to make life easy in these examples.
  - (See below)

```
#include <stdio.h>

int main(){
	printf("Bonjour!\n");
	return 0;
}
```

```
mike:examples$ pwd
/home/mike/Desktop/LLVM_6_3_19/build/bin/examples
mike:examples$ ls
hello.cpp
```

Compile and run (1/2)

```
mike:examples$ pwd
/home/mike/Desktop/LLVM_6_3_19/build/bin/examples
mike:examples$ ./../clang++ hello.cpp -o hello
mike:examples$ ./hello
Bonjour!
```

Compile and run (2/2)

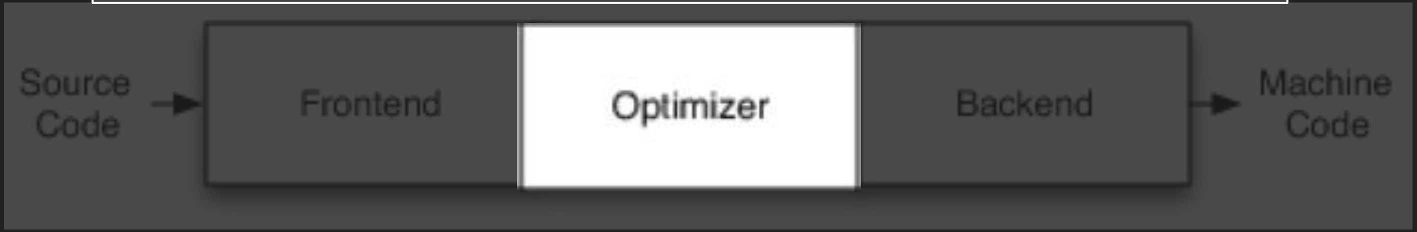
Again, make sure you are using the correct version of clang++ that we built!

```
mike:examples$ ./../clang++ --version
clang version 8.0.0 (tags/RELEASE_800/final 362351)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /home/mike/Desktop/LLVM_6_3_19/build/bin/examples/./..
```

Now we can use clang++ to emit LLVM IR (1/3)

Our goal: Get an intermediate representation

Then we can talk more about this step:



```
mike:examples$ ./../clang++ -S -emit-llvm hello.cpp
```

Now we can use clang++ to emit LLVM IR (2/3)

```
mike:examples$ ./../clang++ -S -emit-llvm hello.cpp
```

Now we can use clang++ to emit LLVM IR (3/3)

- (Compiler arguments explained)
  - `-S` -- only run preprocessor and compilation steps
  - `-emit-llvm` -- Use the LLVM Representation for assembler and object files
  - (Use `clang++ -help` to see options)
- If you are successful, you should see a ‘`hello.ll`’ file.

```
mike:examples$ ls
hello.cpp  hello.ll
```

Aside: Clang++, isn’t this an LLVM talk?

- The news my friends is that LLVM has expanded since the early 2000s!
- LLVM is an umbrella of tools

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "Apache 2.0 License with LLVM exceptions"

LLVM Tools

LLVM Tools - clang/clang++

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
   - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
   - http://llvm.org/docs/GettingStarted.html#llvm-tools

What a second Mike!

So clang or perhaps other tools can work with this “LLVM”

- ☐ Yes
- ☐ No

What a second Mike!

So clang or perhaps other tools can work with this “LLVM”

- [x] Yes
- [ ] No

Modularity (1/2)

- A key feature is that language frontends can all target the same IR
- The optimizer can optimize that IR
- And the code generator can just the same target many other targets

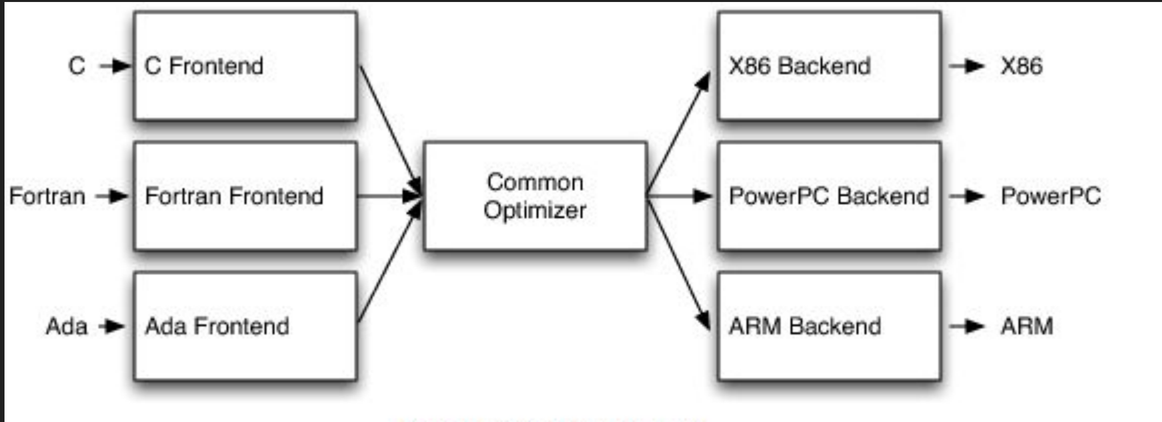


Figure 11.2: Retargetablity

sources: [AOSA Book]

Modularity (2/2)

- A key fea… R
- The opti…
- And the … ets

Okay, now let us take a closer look at that IR

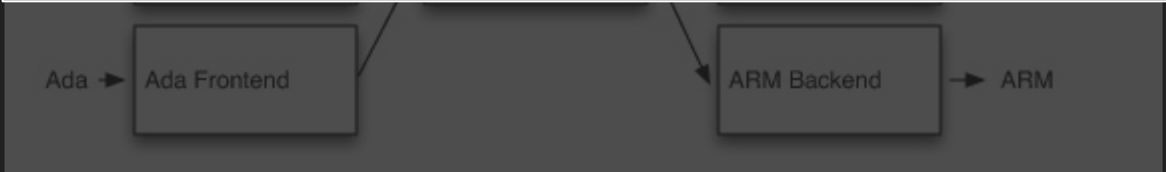


Figure 11.2: Retargetablity

sources: [AOSA Book]

[Pop Quiz] What does this function do? (1/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

[Pop Quiz] What does this function do? (2/

Guesses from the audience?

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

[Pop Quiz] What does this function do? (3/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 =            %a, %b
  ret i32
}
```

Well it is named “add1”

[Pop Quiz] What does this function do? (4/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a,
  ret i32 %tmp1
}
```

There are 2 i32 arguments

[Pop Quiz] What does this function do? (5/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a,
  ret i32 %tmp1
}
```

i32 = int

[Pop Quiz] What does this function do? (6/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
     mp1 = add i32 %a, %b
        2 %tmp1
```

Every function has a starting point

[Pop Quiz] What does this function do? (7/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
      i32 %tmp1
```

We store a result of an ‘add’ operation

[Pop Quiz] What does this function do? (8/9)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

Then return the result as an int

[Pop Quiz] What does this function do? (9/9)

If you can read assembly (or even C!) you can understand LLVM Intermediate Representation

LLVM’s Secret Sauce -- the IR

(IR = Intermediate Representation)

LLVM IR

- The LLVM IR can be targeted by many languages (we have discussed that)
  - It is fairly readable
  - It is also fairly writeable, considered a first-class language!
    - It is well-defined! (You have an alternative to targeting ‘C’ as your IR language :))
- Other takeaways
  - The IR is strongly typed (e.g. i32 or even with pointers such as i32*)
  - There are an infinite number of registers
    - You did not see a finite amount of registers like %rax, %rdx, %r15 if you are use to x86
    - Rather, anything that starts with ‘%’ is a temporary register
    - IR uses Single Static Assignment (SSA) form.
      - Aides in program analysis and compiler optimizations
        - Constant Propagation
        - Dead Code Elimination
        - etc.

sources: AOSA Book

(Quick Aside: SSA example from wikipedia)

https://en.wikipedia.org/wiki/Static_single_assignment_form

$$
\begin{aligned}
y &:= 1 \\
y &:= 2 \\
x &:= y
\end{aligned}
$$

Not SSA

$$
\begin{aligned}
y_1 &:= 1 \\
y_2 &:= 2 \\
x_1 &:= y_2
\end{aligned}
$$

Uses SSA

(Quick Aside: SSA example from wikipedia)

https://en.wikipedia.org/wiki/Static_single_assignment_form

```
y := 1
y := 2
x := y
```

Not SSA

$y_1 := 1$

$y_2 := 2$

$x_1 := y_2$

Uses SSA

Quickly notice we can eliminate an extra variable

(Again, more examples from AOSA book from Lattner himself)

11.3. LLVM's Code Representation: LLVM IR

With the historical background and context out of the way, let's dive into LLVM: The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represen code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a `.ll` file:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Using Clang++ and Generating IR

```
mike:examples$ ./../clang++ -S -emit-llvm hello.cpp
```

Example 1 | `hello.cpp`

- Returning to our example of ‘hello world’
- This command generated a `.ll` file (two lower-case L’s).
  - .ll files are the ‘textual’ form of LLVM’s IR.

```
mike:examples$ ls
hello.cpp  hello.ll
```

(Note ubuntu users: if the above failed, try adding `-fno-use-cxa-atexit` link)

And here it is:

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

@.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1

; Function Attrs: noinline norecurse optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1

attributes #0 = { noinline norecurse optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "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" }
```

Audience, what stands out?

Pause -- Really take a second to look at the IR
What jumps out at you in this snippet?

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

@.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1

; Function Attrs: noinline norecurse optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1

attributes #0 = { noinline norecurse optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "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" }
```

My Findings

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

@.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1

; Function Attrs: noinline norecurse optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1

attributes #0 = { noinline norecurse optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "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" }
```

- Source filename
- Data layout
- Target Triple
- Functions, Structure Types
- Lots of % signs - These are registers (Remember the thing about SSA?)

- Other important things (not in this IR--phi nodes)
- Attributes
- type information! Cool--better than assembly!
- Meta data (At the end with the “!”)

Targeting different backends (1/2)

- Source filename
- Data layout
- Target Triple
- Functions, Structure Types
- Lots of % signs - These are registers
- Other important things (not in this IR--phi nodes)
- Attributes
- type information! Cool--better than assembly!
- Meta data (At the end with the “!”)

Looks like good information to have for this stage (which we will not get to today)

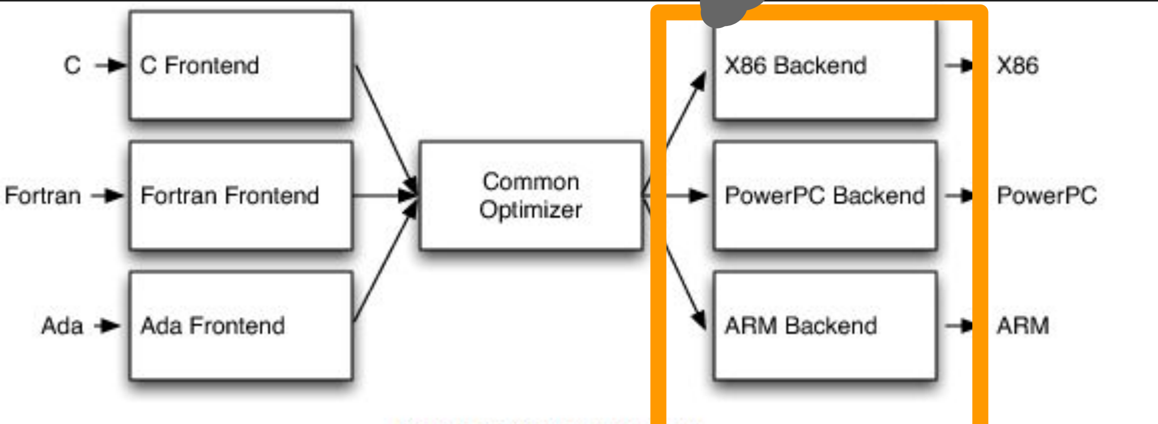


Figure 11.2: Retargetability

Targeting different backends (2/2)

- Source file
- Data layou
- Target Tripl
- Functions,
- Lots of % s registers
- Other impo this IR--phi
- Attributes
- type inform than assem
- Meta data (At the end with the “!”)

Are you enjoying the readability of IR yet?

Good news, machines like IR too

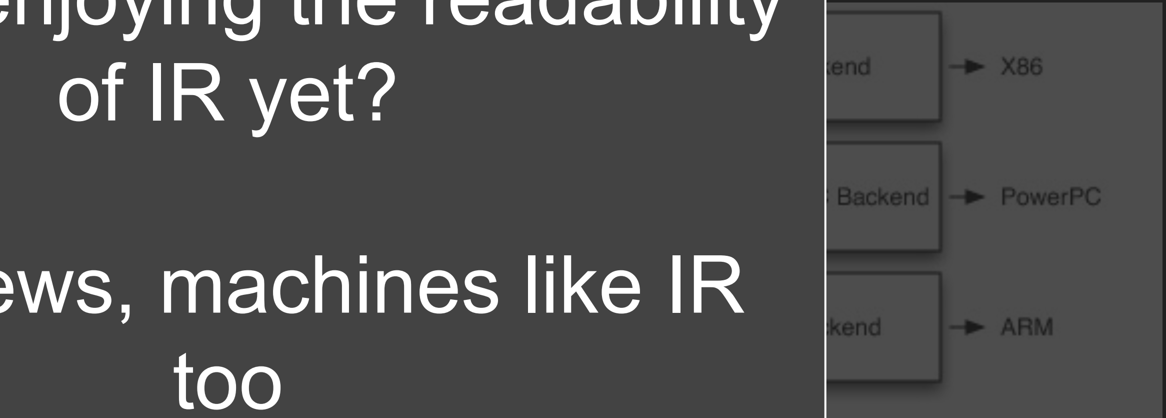


Figure 11.2: Retargetablity

LLVM Tools - lli

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
   - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
   - http://llvm.org/docs/GettingStarted.html#llvm-tools

The IR is very assembly like -- very readable! (1/2)

- In fact the machine can read it, and the machine can directly execute the IR using it's Just-in-time (JIT compile for current architecture) execution engine.
- Let's do it now using lli ("`L L I`")
- What do you see?
  - Program should execute -- even though you did not see executable!
  - LLI can directly execute IR!

```
mike:examples$ ./../lli hello.ll
Bonjour!
```

- (If you're on Ubuntu 16.04--you may need an additional flag)
  - `./../llvm_build/bin/clang++ -S -emit-llvm hello.cpp -fno-use-cxa-atexit`

The IR is very assembly like -- very readable! (2/2)

- In fact th ... cute the IR using it's ... tion engine.
- Let's do
- What do
  - Progr
  - LLI ca

```
mike ... o.ll
Bonj
```

IR has a binary form called bitcode (.bc).
Binary data will be more compact and thus to run through a JIT!

- (If you're on Ubuntu 16.04--you may need an additional flag)
  - `./../llvm_build/bin/clang++ -S -emit-llvm hello.cpp -fno-use-cxa-atexit`

LLVM Tools - `llvm-as`

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
    - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
    - http://llvm.org/docs/GettingStarted.html#llvm-tools

Let's convert `.ll` to a `.bc` file | `llvm-as`

The llvm assembler converts the textual (or readable) IR to bitcode and now we have "`hello.bc`".

```
mike:examples$ ./../llvm-as hello.ll
mike:examples$ ls
hello.bc  hello.cpp  hello.ll
mike:examples$ vim hello.bc
```

Same result, as expected!

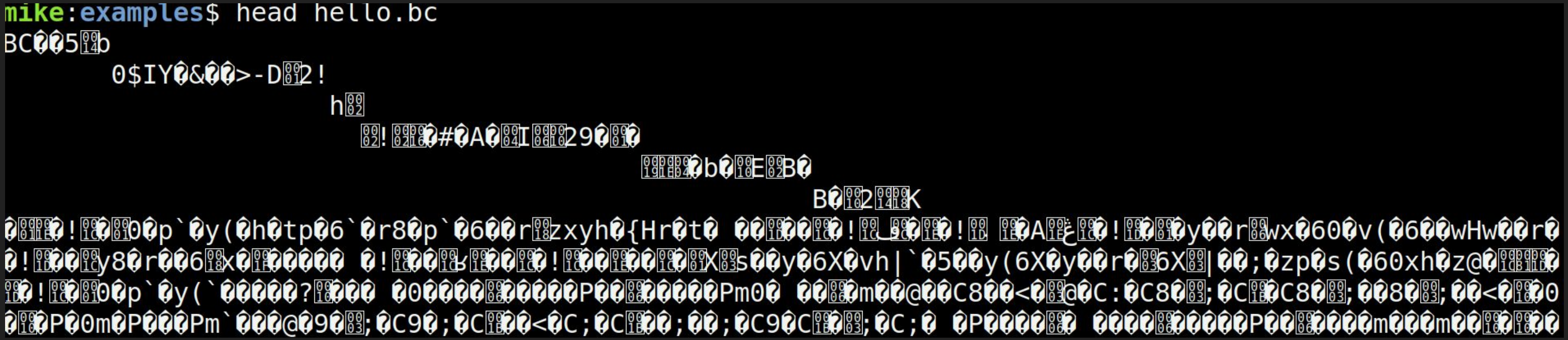
```
mike:examples$ ./../lli hello.bc
Bonjour!
```

lli executes bitcode (binary format of IR) (1/2)

My claim is the JIT engine can execute more efficiently (Why?).

lli executes bitcode (binary format of IR) (2/2)

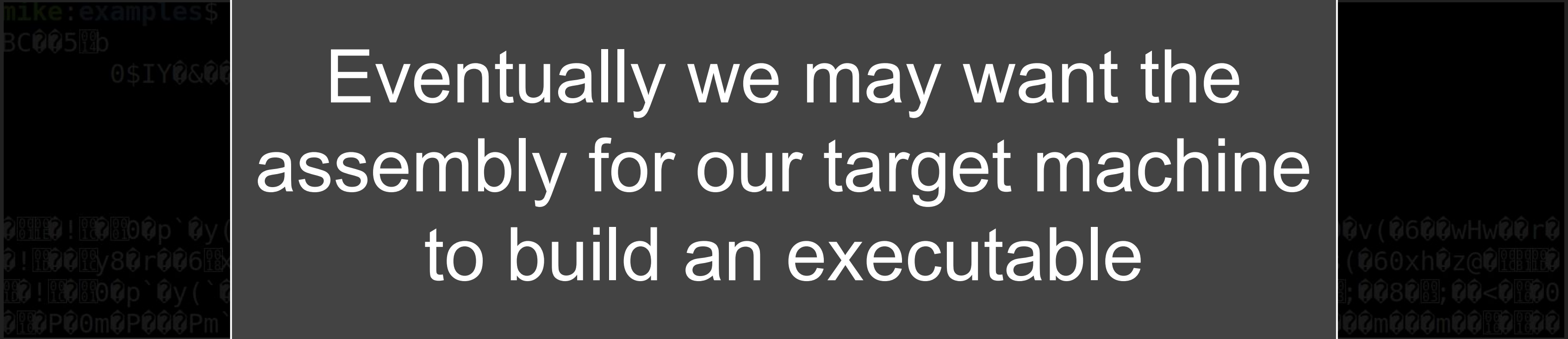
My claim is the JIT engine can execute more efficiently (Why?).



^binary representation of the textual .ll format we previously saw. A little more compressed, smaller file size.

lli executes bitcode (binary format of IR)

My claim is th



Eventually we may want the assembly for our target machine to build an executable

^binary repr ttle more compressed

LLVM Tools - llc

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
   - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
   - http://llvm.org/docs/GettingStarted.html#llvm-tools

The full circle -- compile our IR to assembly (.s file)

Run llc on our .bc file which creates an assembly file (hello.s)

```
mike:examples$ ./../llc hello.bc
mike:examples$ ls
hello.bc  hello.cpp  hello.ll  hello.s
```

The full circle -- compile our IR to assembly (.s file)

Run llc on our .bc file which creates an assembly file (hello.s)

```
mike:examples$ ./../llc hello.bc
mike:examples$ ls
hello.bc  hello.cpp  hello.ll  hello.s
```

hello.s

```
mike:examples$ nl hello.s
     1          .text
     2          .file   "hello.cpp"
     3          .globl  main                            # -- Begin function main
     4          .p2align        4, 0x90
     5          .type   main,@function
     6  main:                                   # @main
     7          .cfi_startproc
     8  # %bb.0:
     9          pushq   %rbp
```

The full circle -- compile our IR to assembly (.s file)

A wide variety of targets are available for you to generate assembly code.

```
mike:examples$ ./../llvm-as < /dev/null | ./../llc -march=x86 -mattr=help
Available CPUs for this target:

  amdfam10      - Select the amdfam10 processor.
  athlon        - Select the athlon processor.
  athlon-4      - Select the athlon-4 processor.
  athlon-fx     - Select the athlon-fx processor.
  athlon-mp     - Select the athlon-mp processor.
  athlon-tbird  - Select the athlon-tbird processor.
```

The full circle -- compile our IR to assembly (.s file)

A wide variety ... ode.

```
mike:examples$ ... 6 -mattr=help
Available CPUs ...

  amdfam10
  athlon
  athlon-4
  athlon-fx
  athlon-mp
  athlon-tbird
```

At this point in the talk, we have played with IR and gotten familiar with some tools.

We have not utilized the optimizer, (i.e. Lattner’s big idea)

LLVM Tools - opt

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
    - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
    - http://llvm.org/docs/GettingStarted.html#llvm-tools

Lets run opt | `./../opt hello.ll --time-passes`

```
mike:examples$ ./../opt hello.ll --time-passes
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0000 seconds (0.0000 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Module Verifier
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Total

===-------------------------------------------------------------------------===
                          LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0002 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Parse IR
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Total
```

Passes with ‘opt’ (1/2)

- Opt is the ‘optimizer’
- It works by making several passes through a module of code looking for opportunities to ‘optimize’ the code.
- There exists several ways to ‘pass’ through the code and gather information or make code changes.

```
mike:examples$ ./../opt hello.ll --time-passes
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0000 seconds (0.0000 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Module Verifier
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Total

===-------------------------------------------------------------------------===
                          LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0002 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Parse IR
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Total
```

Passes with ‘opt’ (2/2)

- Opt is the ‘optimizer’
- It works by making several **passes** through a module of code looking for opportunities to ‘optimize’ the code.
- There exists several ways to ‘pass’ through the code and gather information or make code changes.

```
mike:examples$ ./../opt hello.ll --time-passes
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0000 seconds (0.0000 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Module Verifier
   0.0000 (100.0%)   0.0000 (100.0%)   0.0000 (100.0%)  Total

===-------------------------------------------------------------------------===
                          LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0002 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Parse IR
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Total
```

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

Different

- Levels o
  - Modu
  - Call C
  - Func
  - Basic
  - (Imm ... day)
- Analysis
  - Analy ... g
  - Trans

Our next task:

Learn how to analyze IR with passes. This can lead toward paths of:

1. Code optimization
2. Code understanding
3. etc.

Goal - Print all of the Functions in a program

- What do we need? (Question for the audience)
- a.) Module Pass - Can think of this as a single source file
- b.) Call Graph Pass - Traverses a program bottom-up
- c.) Function Pass - Runs over individual functions
- d.) Basic Block Pass - Runs over individual basic blocks within a function
- e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

Goal - Print all of the Functions in a p

Guesses from the audience?

- What do we need? (Question for the audience)
- a.) Module Pass - Can think of this as a single source file
- b.) Call Graph Pass - Traverses a program bottom-up
- c.) Function Pass - Runs over individual functions
- d.) Basic Block Pass - Runs over individual basic blocks within a function
- e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

Goal - Print all of the Functions in a program

- What do we need?
- a.) Module Pass - Can think of this as a single source file
- b.) Call Graph Pass - Traverses a program bottom-up
- c.) Function Pass - Runs over individual functions
- d.) Basic Block Pass - Runs over individual basic blocks within a function
- e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

Goal - Print all of the Functions in a

Maybe I would accept other answers as well, but “Function Pass” is the easiest route

- What do we need?
- a.) Module Pass - Can think of this as a single source file
- b.) Call Graph Pass - Traverses a program bottom-up
- c.) **Function Pass - Runs over individual functions**
- d.) Basic Block Pass - Runs over individual basic blocks within a function
- e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

Writing Our First Function Pass

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
  static char ID;
  Hello() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n';
    return false;
  }
}; // end of struct Hello
}  // end of anonymous namespace

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

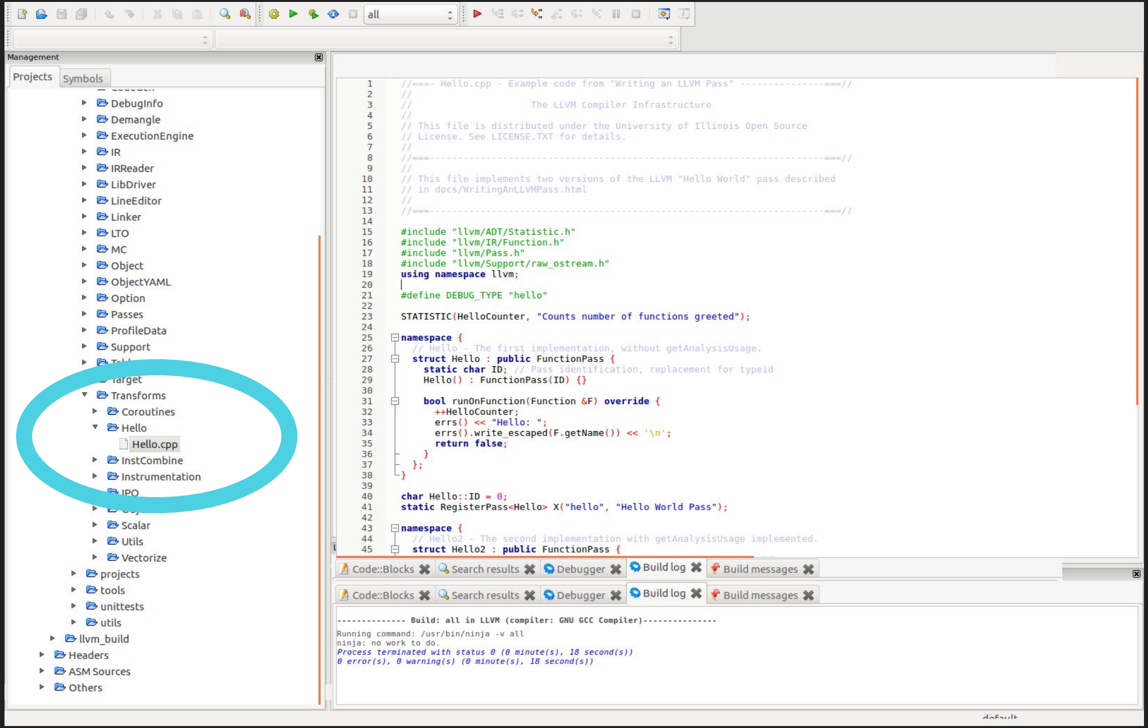
We will be working in: `llvm/lib/Transforms/Hello/Hello.cpp`

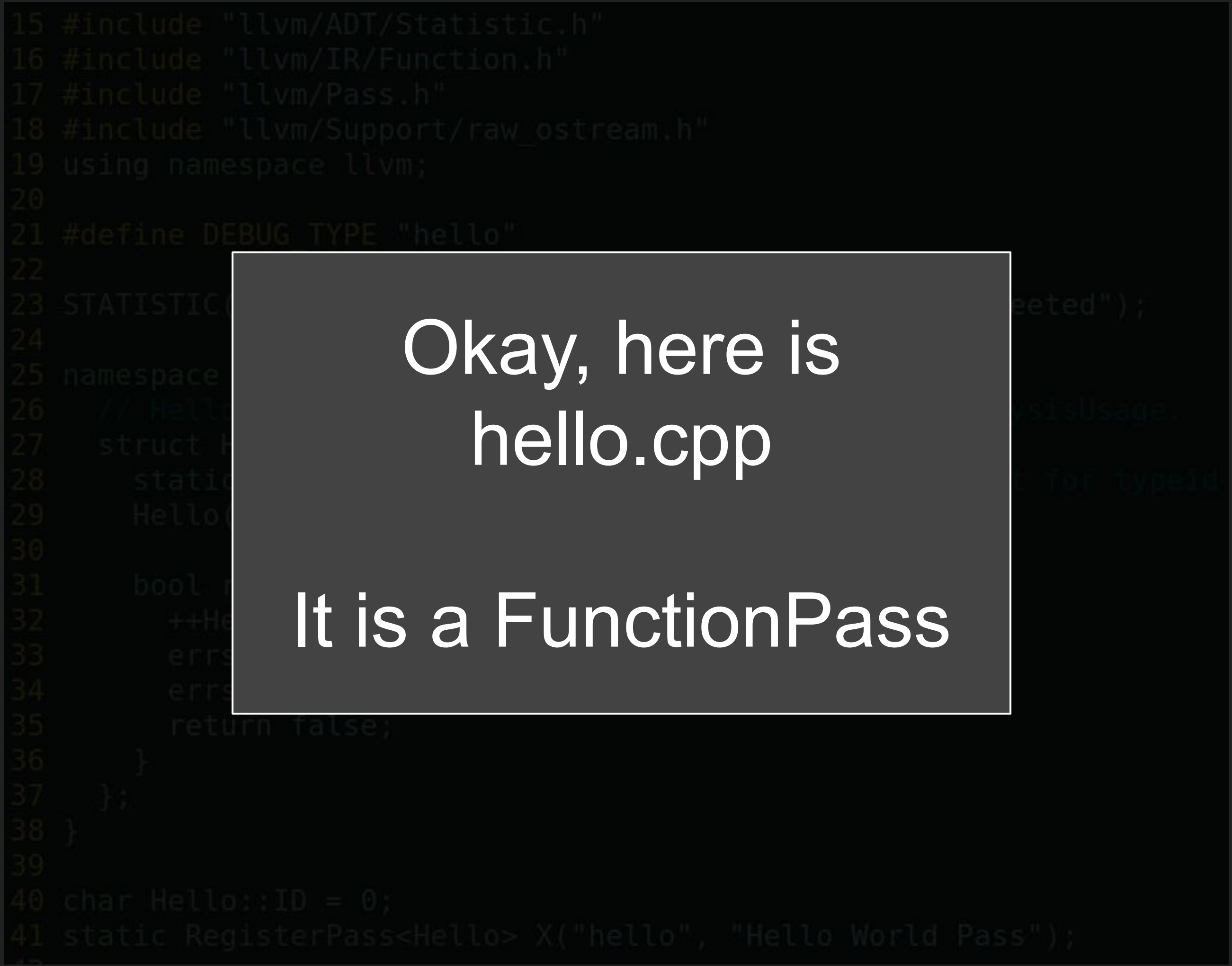
```
mike:Hello$ ls
CMakeLists.txt  Hello.cpp  Hello.exports
mike:Hello$ pwd
/home/mike/Desktop/LLVM 6 3 19/source/llvm/lib/Transforms/Hello
```

- This is given to you when you download LLVM
  - You can learn how to add more passes here
    - http://llvm.org/docs/WritingAnLLVMPass.html

(A visual if anyone setup Codeblocks)

- This is given to you when you download LLVM
- You can learn how to add more passes [here](http://llvm.org/docs/WritingAnLLVMPass.html)
  - http://llvm.org/docs/WritingAnLLVMPass.html





Okay, here is hello.cpp

It is a FunctionPass

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(                                    eeted");

namespace
  // Hello                                    ysisUsage.
  struct H
    static                                    t for typeid.
    Hello(

    bool r
      ++He
      errs
      errs
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

(This code is included with LLVM)

```cpp
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

```cpp
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

The piece we care about for now

Building our hello pass

- Navigate to the build directory
- In the ‘`lib/Transforms/Hello`’ folder you’ll find a make file
- type: ‘`make`’
- Any changes we have made will build.

```
mike:Hello$ cd /home/mike/Desktop/LLVM_6_3_19/build/lib/Transforms/Hello/
mike:Hello$ make
[  0%] Built target LLVMHello_exports
[ 33%] Built target obj.llvm-tblgen
[ 33%] Built target LLVMDemangle
[100%] Built target LLVMSupport
[100%] Built target LLVMTableGen
[100%] Built target llvm-tblgen
[100%] Built target intrinsics_gen
[100%] Built target LLVMHello
mike:Hello$ ls
CMakeFiles  cmake_install.cmake  LLVMHello.exports  Makefile
```

Our pass is then compiled in `build/lib/` as `LLVMHello.so`

```
libLLVMDebugInfoMSF.a          libLLVMX86AsmPrinter.a
libLLVMDebugInfoPDB.a          libLLVMX86CodeGen.a
libLLVMDemangle.a              libLLVMX86Desc.a
libLLVMDlltoolDriver.a         libLLVMX86Disassembler
libLLVMExecutionEngine.a       libLLVMX86Info.a
libLLVMExegesis.a              libLLVMX86Utils.a
libLLVMExegesisX86.a           libLLVMXRay.a
libLLVMFuzzMutate.a            libLTO.so
libLLVMGlobalISel.a            libLTO.so.8
libLLVMInstCombine.a           libOptRemarks.so
libLLVMInstrumentation.a       libOptRemarks.so.8
libLLVMInterpreter.a           LineEditor
libLLVMipo.a                   Linker
libLLVMIRReader.a              LLVMHello.so
libLLVMLibDriver.a             LTO
libLLVMLineEditor.a            Makefile
libLLVMLinker.a                MC
libLLVMLTO.a                   MCA
libLLVMMC.a                    Object
libLLVMMCA.a                   ObjectYAML
libLLVMMCDisassembler.a        Option
libLLVMMCJIT.a                 OptRemarks
libLLVMMCParser.a              Passes
libLLVMMIRParser.a             PrintFunctionNames.so
```

Run our first pass with opt on hello.bc

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc
```

opt tool which we have used before

Run our first pass with opt on hello.bc

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc
```

We load the library which contains our passes

Run our first pass with opt on hello.bc

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc
```

Path to our LLVMHello pass library

Run our first pass with opt on hello.bc

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc
```

The particular function pass we want to run

Run our first pass with opt on hello.bc

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc
```

Our input file (.bc or .ll file)

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < hello.bc > /dev/null
Hello: main
```

Run our first pass with opt on hello.bc

- Neat--we see all of the functions!
  - Or rather, we have one ‘main’ function in our program.

Anatomy of a “Pass”

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

piece of code that does the work

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

We are not 'mutating code' so return false.

Inherit from the ‘FunctionPass’ class

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

Register the pass. This is how the pass is built

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "hello"

STATISTIC(HelloCounter, "Counts number of functions greeted");

namespace {
  // Hello - The first implementation, without getAnalysisUsage.
  struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      ++HelloCounter;
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

i.e. how I knew what to type in the command line in our example

Congratulations on writing/running your first pass

LLVM is properly configured--let’s move on to more analysis!

Static Analysis

Goal of Static Analysis: What information/bugs/performance errors can we uncover before we run the program.

Pros: Gives us full coverage of program
Cons: No real runtime data, overly conservative

Our Second pass -- This time we collect some program stats

1. It will print the function name
2. It will count basic blocks and instruction counts.

Our Second pass -- This time we collect some program stats

1. It will print the function name
2. It will count basic blocks and instruction counts.
3. We'll use this new sample source code -- or even better use one of your own!

```c
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

Compile and Test loops.cpp and use `loops.ll` on `-hello` pass

1. Compile program to IR
   a. `./../clang++ -S -emit-llvm loops.cpp`
   b. Test opt with our old pass (note we can just use the .ll version for this sample)
      i. `./../opt -load ./../../lib/LLVMHello.so -hello < loops.ll > /dev/null`

```
mike:examples$ ./../clang++ -S -emit-llvm loops.cpp
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello < loops.ll > /dev/null
Hello: _Z9countDownv
Hello: _Z7addFuncii
Hello: main
```

The Stats Pass source code

michaeldshah.net/LLVM/Intro/hello.cpp

Okay, here is our second pass

It is a FunctionPass that collects stats

www.

```
namespace {
  // Hello2 - The second pass in our tutorial
  struct Hello2 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello2() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        unsigned int basicBlockCount  = 0;
        unsigned int instructionCount = 0;
        for(BasicBlock &bb : F){
            ++basicBlockCount;
            for(Instruction &i : bb){
                ++instructionCount;
            }
        }
        errs() << "Hello2 is running: ";
        errs().write_escaped(F.getName())   << "Basic Blocks:"  << basicBlockCount
                                            << "Instruction:"   << instructionCount << "\n";
    }

    // We don't modify the program, so we preserve all analyses.
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello2::ID = 0;
static RegisterPass<Hello2>
Y("hello2", "Hello World Pass (with getAnalysisUsage implemented)");
```

Here is where we will accumulate the basic blocks and instructions within our function

www.

```cpp
namespace {
  // Hello2 - The second pass in our tutorial
  struct Hello2 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello2() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        unsigned int basicBlockCount  = 0;
        unsigned int instructionCount = 0;
        for(BasicBlock &bb : F){
            ++basicBlockCount;
            for(Instruction &i : bb){
                ++instructionCount;
            }
        }
        errs() << "Hello2 is running: ";
        errs().write_escaped(F.getName())   << "Basic Blocks:"  << basicBlockCount
                                            << "Instruction:"   << instructionCount << "\n";
    }

    // We don't modify the program, so we preserve all analyses.
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello2::ID = 0;
static RegisterPass<Hello2>
Y("hello2", "Hello World Pass (with getAnalysisUsage implemented)");
```

michaeldshah.net/LLVM/Intro/hello.cpp

Here notice, that within a function, we can iterate through its basic blocks, and every instruction within each basic block

www.

```
namespace {
  // Hello2 - The second pass in our tutorial
  struct Hello2 : public FunctionPass {
    static char ID; // Pass identification, replacement fo
    Hello2() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        unsigned int basicBlockCount  = 0;
        unsigned int instructionCount = 0;
        for(BasicBlock &bb : F){
            ++basicBlockCount;
            for(Instruction &i : bb){
                ++instructionCount;
            }
        }
        errs() << "Hello2 is running: ";
        errs().write_escaped(F.getName())   << "Basic Blocks:"  << basicBlockCount
                                            << "Instruction:"   << instructionCount << "\n";
    }

    // We don't modify the program, so we preserve all analyses.
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello2::ID = 0;
static RegisterPass<Hello2>
Y("hello2", "Hello World Pass (with getAnalysisUsage implemented)");
```

michaelc ello.cpp

And finally we output this information

(Don't forget to save, and rebuild our pass)

```
mike:Hello$ make
[  0%] Built target LLVMHello_exports
[ 33%] Built target obj.llvm-tblgen
[ 33%] Built target LLVMDemangle
[100%] Built target LLVMSupport
[100%] Built target LLVMTableGen
[100%] Built target llvm-tblgen
[100%] Built target intrinsics_gen
Scanning dependencies of target LLVMHello
[100%] Building CXX object lib/Transforms/Hello/CMakeFiles/LLVMHello.dir/Hello.cpp.o
```

Results of pass 2 (with loops.ll)

- `./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2 is running: _Z9countDownvBasic Blocks:4Instruction:11
Hello2 is running: _Z7addFunciiBasic Blocks:1Instruction:8
Hello2 is running: mainBasic Blocks:1Instruction:6
```

Results of pass 2 (with loops.ll)

- `./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/nul
Hello2 is running: _Z9countDownvBasic Blocks:4Instruction:11
Hello2 is running: _Z7addFunciiBasic Blocks:1Instructi
Hello2 is running: mainBasic Blocks:1Instruction:6
```

Same library, but different pass that's it!

Results of pass 2 (with loops.ll)

- `./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2 is running: _Z9countDownvBasic Blocks:4Instruction:11
Hello2 is running: _Z7addFunciiBasic Blocks:1Instruction:8
Hello2 is running: mainBasic Blocks:1Instruction:6
```

Observe here, same pass runs on every function. There is no “memory” here of previous runs. Need a data structure, analysis pass, or perhaps “module pass”

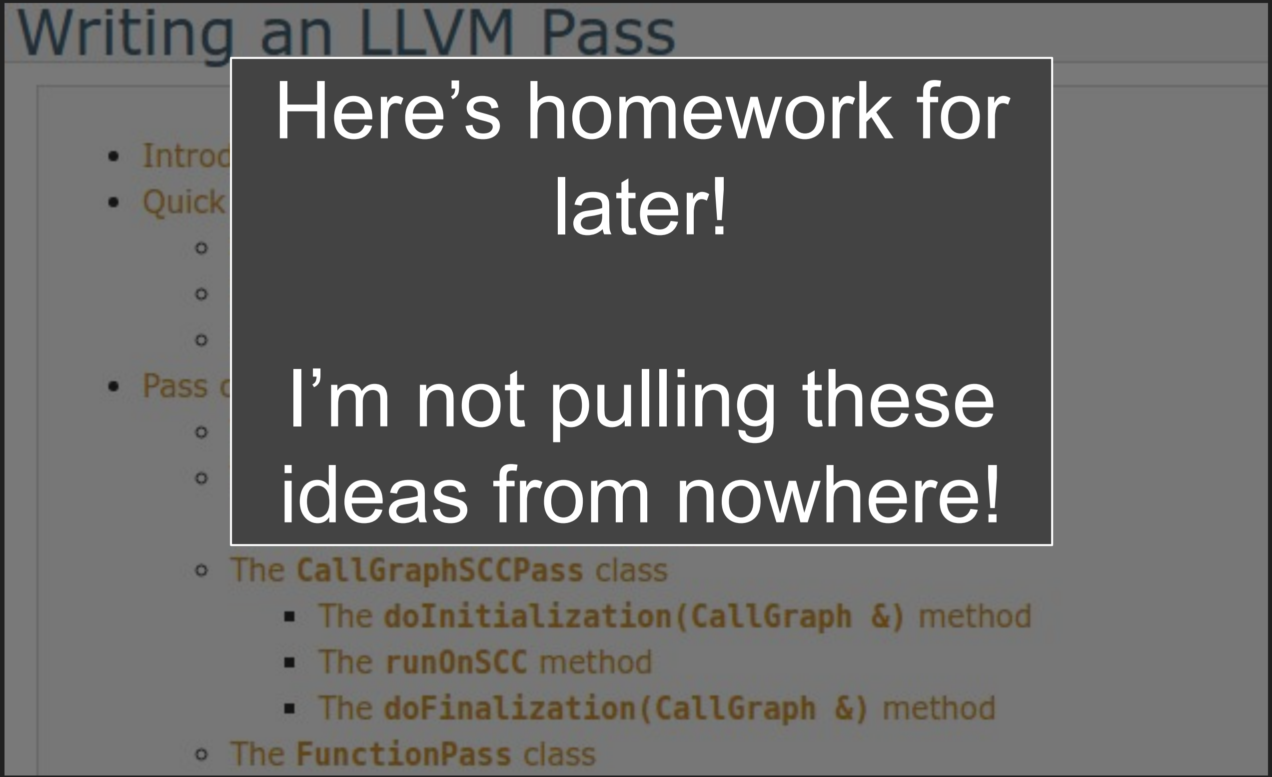
Results of pass 2 (with loops.ll)

- `./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2 is running: _Z9countDownvBasic Blocks:4Instruction:11
Hello2 is running: _Z7addFunciiBasic Blocks:1Instruction:8
Hello2 is running: mainBasic Blocks:1Instruction:6
```

- Let’s add more!
- What can we do with instruction information?

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


Writing an LLVM Pass

- Introd
- Quick
- Pass c
  - The CallGraphSCCPass class
    - The doInitialization(CallGraph &) method
    - The runOnSCC method
    - The doFinalization(CallGraph &) method
  - The FunctionPass class

Here's homework for later!

I'm not pulling these ideas from nowhere!

```
#include "llvm/IR/CallSite.h"
namespace {
  // Hello3 - The third part of our tutorial
  struct Hello3 : public FunctionPass {
```

Okay, here is our third pass

It is a FunctionPass that shows direct function calls

```
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello3::ID = 0;
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

```cpp
#include "llvm/IR/CallSite.h"
namespace {
  // Hello3 - The third part of our tutorial
  struct Hello3 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello3() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of our instruction
                CallSite cs(&i);
                if(!cs.getInstruction()){
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs()  << "\tDirect call to function:" << f->getName()
                            << " from "                     << F.getName() << "\n";
                }
            }
        }

        return false;
    }

    // We don't modify the program, so we preserve all analyses.
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello3::ID = 0;
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

```
#include "llvm/IR/CallSite.h"
namespace {
  / Hello3 - The third part of our tutorial
  ct Hello3 : public FunctionPass {
   ic char ID; // Pass identification, replacement for typeid
   () : FunctionPass(ID) {}

       Function(Function &F) override {
       cBlock &bb: F){
          for(Instruction &i: bb){
              // Find where callsite is of our instruction
              CallSite cs(&i);
              if(!cs.getInstruction()){
                  continue;
              }
              Value *called = cs.getCalledValue()->stripPointerCasts();
              if(Function* f = dyn_cast<Function>(called)){
                  errs()  << "\tDirect call to function:" << f->getName()
                          << " from "                     << F.getName() << "\n";
              }
          }

        return false;
    }

    // We don't modify the program, so we preserve all analyses.
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesAll();
    }
  };
}

char Hello3::ID = 0;
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

Note the new header included

Find Direct Calls

```
    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of our instruction
                CallSite cs(&i);
                if(!cs.getInstruction()){
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs()  << "\tDirect call to function:" << f->getName()
                            << " from "                     << F.getName() << "\n";
                }
            }
        }

        return false;
    }
```

```
#include "llvm/IR/CallSite.h"
namespace {
  // Hello3 - The third part of our tutorial
  struct Hello3 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello3() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of our instruction
                CallSite cs(&i);
```

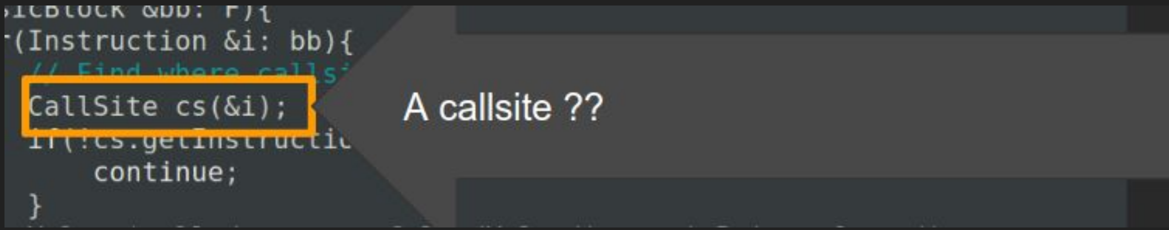
```
    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsi
                CallSite cs(&i);
                if(!cs.getInstructio
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs()  << "\tDirect call to function:" << f->getName()
                            << " from "                     << F.getName() << "\n";
                }
            }
        }

        return false;
    }
```

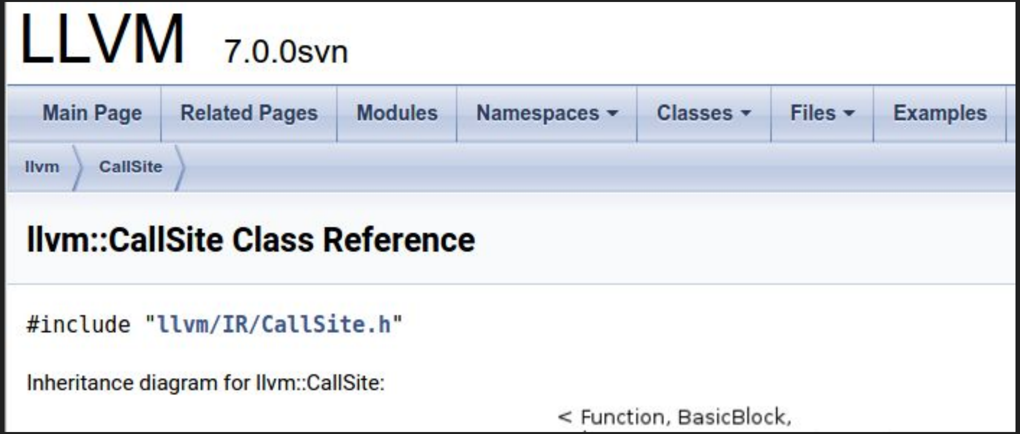
A callsite ??

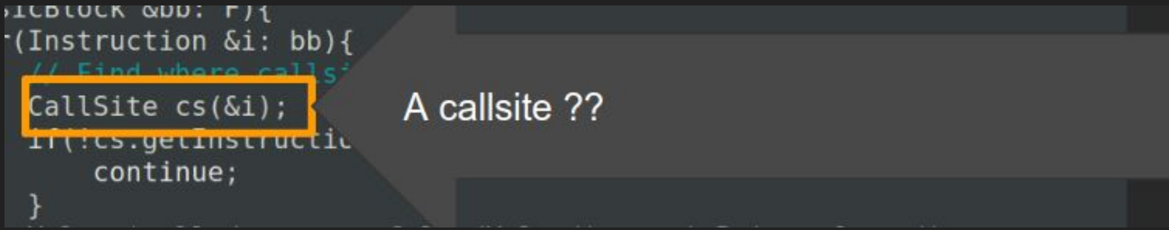
```
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

LLVM Docs



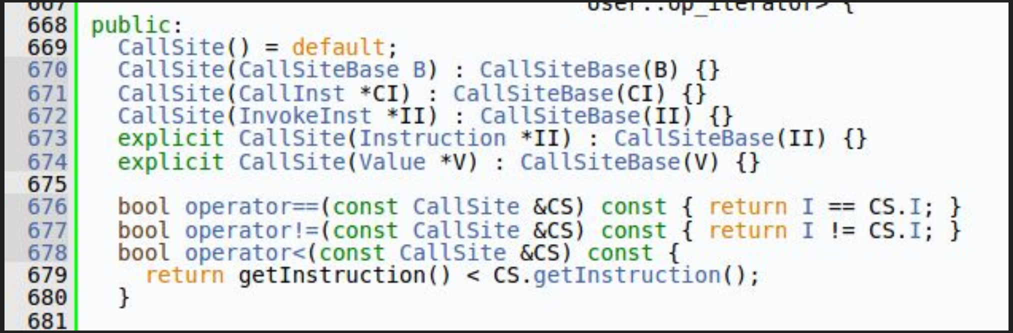
- I do not actually know all of the LLVM commands by heart.
- As you start with LLVM, it is a good idea to keep the doxygen documentation open.
- “googling LLVM _______” will lead you to the correct page most often
  - http://llvm.org/doxygen/classllvm_1_1CallSite.html





LLVM Docs

- From the documentation you can navigate to the appropriate function and even the source code



(Pssst! You have the source code as well)

Here is a sample grep

```
grep --include="*.cpp" -nr "getInstruction()" .
```

```
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:320:      Call->replaceAllUsesWith(NewCS.getInstruction());
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:824:    if (CS.getInstruction() == nullptr || !CS.isCallee(&U))
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:827:    if (CS.getInstruction()->getParent()->getParent() == F)
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:1028:        Function *Caller = OldCS.getInstruction()->getParent()->getParent();
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:163:    Instruction *Call = CS.getInstruction();
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:187:      cast<CallInst>(NewCS.getInstruction())
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:200:      Call->replaceAllUsesWith(NewCS.getInstruction());
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:521:    const Instruction *TheCall = CS.getInstruction();
```

- Often times grepping through the source code gives you ideas of how to use instructions
- I myself do not pretend to be compared with the LLVM experts!

(continued) Find Direct Calls

```
#include "llvm/IR/CallSite.h"
namespace {
```

```
    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of
                CallSite cs(&i);
                if(!cs.getInstruction()){
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs()  << "\tDirect call to function:" << f->getName()
                            << " from "                     << F.getName() << "\n";
                }
            }
        }

        return false;
    }
```

If our instruction is not a 'callable' (i.e. a function)

```
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

(continued) Find Direct Calls

```
#include "llvm/IR/CallSite.h"
namespace {
  // Hello3 - The third part of our tutorial
  struct Hello3 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello3() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is
                CallSite cs(&i);
```

```
    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is
                CallSite cs(&i);
                if(!cs.getInstruction()){
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs() << "\tDirect call to function: " << f->getName()
                           << " from "                      << F.getName() << "\n";
                }
            }
        }

        return false;
    }
```

```
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

Find out if our 'callee' is a direct function call (not a function pointer or anything)

The Result!

- Simple little function pass
- Now you can use this information to build a data structure
    - The function “F” is the caller, and “f” the callee.
    - Each of these forms an edge and could be put into a graph data structure.
    - Then output static graphs!
- `./../opt -load ./../../lib/LLVMHello.so -hello3 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello3 < loops.ll > /dev/null
        Direct call to function:_Z7addFuncii from main
        Direct call to function:printf from main
        Direct call to function:_Z9countDownv from main
```

Bonus Trick: Outputting graphs

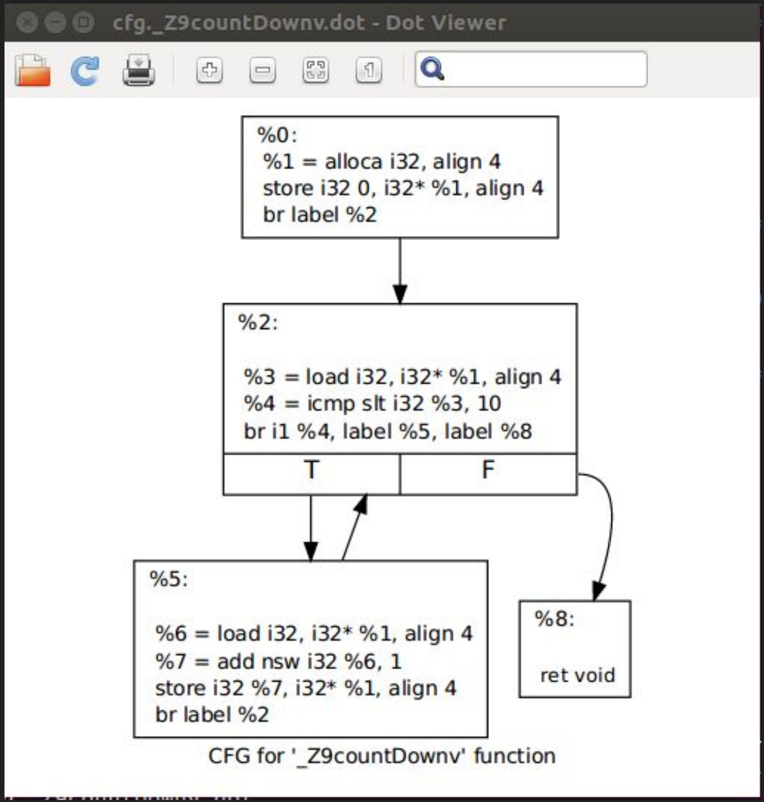
LLVM actually provides a pass that can output control flow graphs

- Install a dot file viewer
  - `sudo apt install xdot` (for linux)
- Generate a dot file with
  - `./../opt -dot-cfg-only loops.ll > /dev/null`
- View dot file with
  - `xdot cfg._Z9countDownv.dot`

```
mike:examples$ ./../opt -dot-cfg-only loops.ll > /dev/null
Writing 'cfg._Z9countDownv.dot'...
Writing 'cfg._Z7addFuncii.dot'...
Writing 'cfg.main.dot'...
```

Here is the ‘countdown function’ from loops.cpp

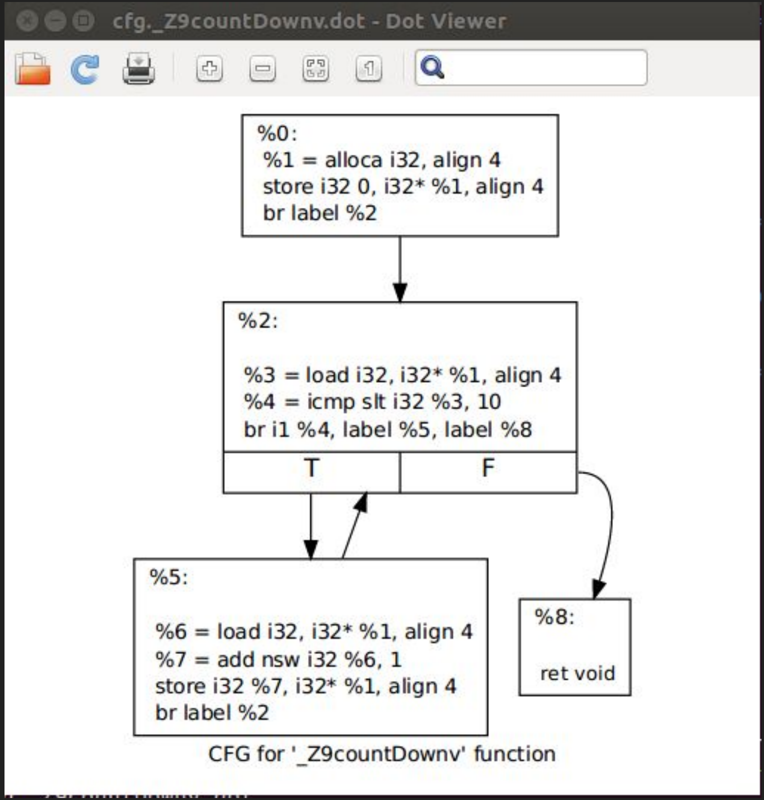
```
void countDown(){
    int x = 0;
    while(x<10){
        ++x;
    }
}
```



Here is the ‘countdown function’ from loops.cpp

- You can slowly map each basic block from the visualization to the C++ code in this way.

```
void countDown(){
    int x = 0;
    while(x<10){
        ++x;
    }
}
```



Here is the ‘countdown function’ from loops.cpp

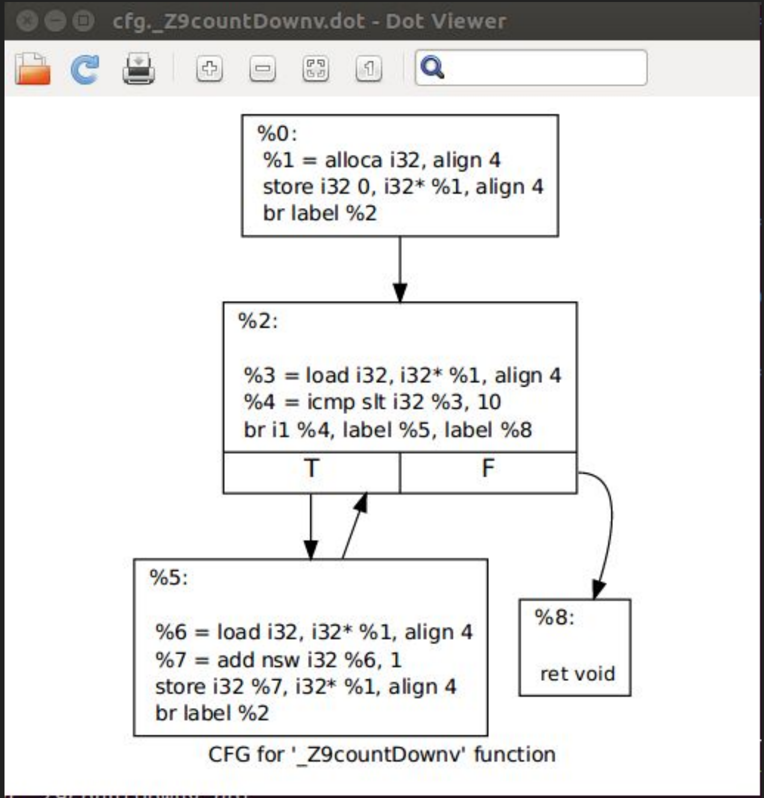
- You can slowly map each basic block from the visualization or directly to the IR

```
; Function Attrs: noinline nounwind optnone uwtable
define void @_Z9countDownv() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %2

; <label>:2:                                      ; preds = %5, %0
  %3 = load i32, i32* %1, align 4
  %4 = icmp slt i32 %3, 10
  br i1 %4, label %5, label %8

; <label>:5:                                      ; preds = %2
  %6 = load i32, i32* %1, align 4
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* %1, align 4
  br label %2

; <label>:8:                                      ; preds = %2
  ret void
}
```



Dynamic Analysis

Goal of Dynamic Analysis: What information/bugs/performance errors can we uncover when we run the program.

Pros: Gives us real values

Cons: Instrumentation effects results & Performance

Why use LLVM for this?

We can insert/inject code to monitor or change behavior of our code.

Dy… sis

G… at

information/… we uncover

when we run the program.

Pros: Gives us real values

Cons: Instrumentation effects results & Performance

Adding in Functions (For Dynamic Analysis)

- Typically this is done in an ad-hoc fashion
  - Either spreading in '`printf`' functions everywhere
  - Lots of `#define #endif`
- If we have our source code, we can inject code as needed.
  - No need to mess up or keep copies of various source versions.

- Fair warning, I am running through these examples fast, but you have the slides
  - (Lots of source code on slides ahead--I am breaking powerpoint rules!)

Step 1:

Let’s write some code that we want to instrument

Step 1: Write a ‘hook’ or ‘profiling code’

Let’s write some code that we want to instrument

Here is a function ‘`__initMain`’ that will be inserted in our ‘main’ function and print a message (new file called: `instrumentation.cpp`)

```
#include <stdio.h>

// This is the function that is called at
// the very start of the program.
// It will be called right after main.
// "dummyValue" does nothing except demonstrate
// how to pass a single argument in our pass.
void __initMain(int dummyValue){
    printf("Hello, you are running an instrumented binary.\nPerformance may vary while running an instrumented binary.\n");
    // Do more work here...
}
```

Step 1: Generate IR for hook

- Now let's create the intermediate representation of our code.
  - `./../clang++ -S -emit-llvm instrumentation.cpp`

instrumentation.ll

```
; Function Attrs: noinline optnone uwtable
define dso_local void @_Z10__initMaini(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([107 x i8], [107 x i8]* @.str, i32 0, i32 0))
  ret void
}
```

Step 1: Generate IR for hook

- Now let's create the intermediate representation of our code.
  - ./../clang++ -S -emit-llvm instrumentation.cpp

instrumentation.ll

```
; Function Attrs: noinline optnone uwtable
define dso_local void @_Z10__initMaini(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, al
  %3 = call i32 (i8*                    etelementptr inbounds ([107 x i8], [107 x i8]* @.str, i32 0, i32 0))
  ret void
}
```

This is our function name. Note it "looks weird". It is a mangled function name.

Step 2: Let’s find some source code to instrument

How about our hello.cpp program. And we already have hello.ll from previous examples

This is the simplest program with one function
(hello.cpp -- yes I know I am using printf)

```
#include <stdio.h>

int main(){
	printf("Bonjour!\n");
	return 0;
}
```

Now time for the Module pass

Why?

1. To show you a module pass
2. It makes a little more sense (to me) to search functions in a module I want to instrument.
   a. New headers needed: `#include "llvm/IR/Module.h"`

The Module pass | Setup in 3 parts (in my code)

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hook some source into.
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our functions in the module
    // This is where you could instrument only a subset
    // of functions.
    // Be careful just not to modify instrumentation functions in most cases!
    Module::FunctionListType &functions = M.getFunctionList();
    for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=FE; ++FI){
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentEnterFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

The Module pass

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hoo
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our fun
    // This is where you could instrument      y a subset
    // of functions.
    // Be careful just not to modify instrumentation functions in most cases!
    Module::FunctionListType &functions = M.getFunctionList();
    for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=FE; ++FI){
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentEnterFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

1.) Create a “stub” function

The Module pass

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hoo
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our fu
    // This is where you could instrument
    // of functions.
    // Be careful just not to modify instrumentation functions in most cases!
    Module::FunctionListType &functions = M.getFunctionList();
    for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=FE; ++FI){
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentEnterFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

1.) Notice it is using the ‘mangled’ c++ function name

The Module pass

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hook some source into.
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our functions in the module
    // This is where you could instrument only a subset
    // of functions.
    // Be careful just not to modify instrumentation functi
    Module::FunctionListType &functions = M.getFunctionLi
    for(Module::FunctionListType::iterator FI = functio
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentEnterFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

2.) This next chunk of code iterates through a Module to look at all of the functions

The Module pass

```
bool runOnModule(Module &M) override {
    // The setup hooks function creates
    // a 'stub' function for us to hook some source into.
    setupHooks("_Z10__initMaini",M);
    // We next loop through all our functions in the module
    // This is where you could instrument only a subset
    // of functions.
    // Be careful just not to modify instrumentation functions in most cases!
    Module::FunctionListType &functions = M.getFunctionList();
    for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=FE; ++FI){
        // Ignore our instrumentation function
        if("_Z10=initMaini"==FI->getName()){
            continue;
        }
        if("main"==FI->getName()){
            InstrumentF erFunction("_Z10__initMaini",*FI,M);
        }
    }
    return true;
}
```

3.) I am modifying code, so I return true for this pass

setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```
void setupHooks(StringRef InstrumentingFunctionName, Module &M){
    auto &Context = M.getContext();

    Type* voidTy    = Type::getVoidTy(Context);
    Type* intTy     = Type::getInt32Ty(Context);
    // Specify the return value, arguments, and if there are variable number of arguments.
    FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
    Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunctionName);
}
```

setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```
void setupHooks(StringRef InstrumentingFunctionName, Module &M){
    auto &Context = M.getContext();

    Type* voidTy    = Type::getVoidTy(Context);
    Type* intTy     = Type::getInt32Ty(Context);
    // Specify the return value, arguments, and if there are variable number of arguments.
    FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
    Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunctionName);
}
```

The observation from setupHooks() is that I am building up a function that returns void and takes in one argument

setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```cpp
void setupHooks(StringRef InstrumentingFunctionName, Module &M){
    auto &Context = M.getContext();

    Type* voidTy    = Type::getVoidTy(Context);
    Type* intTy     = Type::getInt32Ty(Context);
    // Specify the return value, arguments, and if there are variable number of arguments.
    FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
    Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunctionName);
}
```

The observation from setupHooks() is that I am building up a ‘function’ that returns void and takes in one argument

Which is exactly the signature of __initMain

```c
#include <stdio.h>

// This is the function that is called at
// the very start of the program.
// It will be called right after main.
// "dummyValue" does nothing except demonstrate
// how to pass a single argument in our pass.
void __initMain(int dummyValue){
    printf("Hello, you are running an instrumented binary.\nPerformance may vary while running an instrumented binary.\n");
    // Do more work here..
}
```

InstrumentEnterFunction

- Same idea from InstrumentEnterFunction
- I am building up a specific function to insert

```
void InstrumentEnterFunction(StringRef InstrumentingFunctionName,Function& FunctionToInstrument, Module& M){
    // Create the actual function
    // If we have a function already, then the below is very useful
    //
    // FunctionType* funcTy = M.getFunction(InstrumentingFunctionName)->getFunctionType();
    //
    // However, we are hooking into a function that we will merge later, so we instead build our function type
    // Both methods will allow us to then modify the function arguments.
    //
```

- Full code sample for reference -- the last piece you need
- (Read at your leisure)

```cpp
void InstrumentEnterFunction(StringRef InstrumentingFunctionName,Function& FunctionToInstrument, Module& M){
    // Create the actual function
    // If we have a function already, then the below is very useful
    //
    // FunctionType* funcTy = M.getFunction(InstrumentingFunctionName)->getFunctionType();
    //
    // However, we are hooking into a function that we will merge later, so we instead build our function type
    // Both methods will allow us to then modify the function arguments.
    //
    // Build out the function type
    auto &Context = M.getContext();
    // The functions return type
    Type* voidTy = Type::getVoidTy(Context);
    // The start of our parameters
    Type* intTy = Type::getInt32Ty(Context);
    // push back all of the parameters
    std::vector<llvm::Type*> params;
    params.push_back(intTy);
    // Specify the return value, arguments, and if there are variable numbers of arguments.
    FunctionType* funcTy = FunctionType::get(voidTy, params, false);
    // Create a Constant that grabs our function
    Constant* hook = M.getOrInsertFunction(InstrumentingFunctionName, funcTy);

    // We determine where we want to add our instrumentation.
    // In this instance, we want to instrument the first basic block, and
    // put the instruction at the front. Every function has at least an entry:
    // block in the LLVM IR, so this should be valid.
    BasicBlock *BB = &FunctionToInstrument.front();
    Instruction *I = &BB->front();

    // In order to set the arguments of the instrumenting function, we are going to
    // get all of our instrumenting functions arguments, and then modify them.
    std::vector<Value*> args;
    for(unsigned int i=0; i< funcTy->getNumParams(); ++i){
        Type* t = funcTy->getParamType(i);
        // We get the argument, and then we can re-assign its value
        // In this case, we are looking at our obController to see the function name in the hashmap, and then its value
        //
        // TODO: For now I know this is a constant, but perhaps this could change in the future.
        llvm::Value* foo = 0;
        //Value *newValue = dyn_cast<llvm::ConstantInt>(foo);
        Value *newValue = ConstantInt::get(t,0x1234);
        args.push_back(newValue);
        errs() << "getNumParams()" << i << "\n";
    }

    // Create our function call
    CallInst::Create(hook,args)->insertBefore(I);
}
```

InstrumentEnte

- Same idea from I
- I am building up a

```
void InstrumentEnterFuncti                                nctionToInstrument, Module& M){
    // Create the actual f
    // If we have a functi
    //
    // FunctionType* funcT                                nctionType();
    //
    // However, we are hoo                                e instead build our function type
    // Both methods will a
    //
```

Why not do something more simple?

With this approach, I can push different values as parameters based on *whatever* I need to do.

Steps to running our Module pass (Hello4)

Get our source code setup by running our pass in.

```
examples$ ./../opt -load ./../../lib/LLVMHello.so -hello4 -S hello.ll > readyToBeInstrumented.ll
```

Link in our instrumentation

```
examples$ ./../llvm-link readyToBeInstrumented.ll instrumentation.ll -S -o instrumentDemo.ll
```

LLVM Tools - llvm-link

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
    - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
    - http://llvm.org/docs/GettingStarted.html#llvm-tools

LLVM Tools - ll

1. clang - Clang is th backend)
    - Likely you have h
2. llvm-as - Takes LL it to bitcode format.
3. llvm-dis - Convert y
4. llvm-link - Links tw e.
5. lli - Directly execu
6. llc - Static compile code) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
    - http://llvm.org/docs/GettingStarted.html#llvm-tools

Now that our files are merged, there is a declaration and a definition for our instrumentation!

LLVM-Link

- Think of this like a ‘linker’ for IR code.
- Sometimes it is useful to link all of your code together, and then run your optimizations
  - We call this “whole program optimization”

```
examples$ ./../llvm-link readyToBeInstrumented.ll instrumentation.ll -S -o instrumentDemo.ll
```

Grand Finale!

Run our linked .ll file (using lli or compile to source)

```
mike:examples$ ./../lli instrumentDemo.ll
Hello, you are running an instrumented binary.
Performance may vary while running an instrumented binary.
Bonjour!
```

Grand Finale!

Run our linked .ll file (using lli or compile to source)

```
mike:examples$ ./../lli instrumentDemo.ll
Hello, you are running an instrumented binary.
Performance may vary while running an instrumented binary.
Bonjour!
```

It works, we see our message before the “Bonjour” from hello.cpp!!

Going Further (Challenges/Project Ideas)

Time permitting:

- Easy
  - Print out function arguments
  - Recover and print metadata and/or Profile Guided Optimization Data with functions
  - Write a python script that ‘llvm-links’ all of your .ll files together.
- Medium
  - Build both a control flow graph and call graph and output to .dot
  - Find Program attributes
    - Add an attribute for any function < 10 instructions, and force it to inline
- Hard/Interesting?
  - Autovectorizing (Find patterns and Insert SIMD instructions)
  - Investigate the “sanitzer” projects. See if you can add interesting printouts.

Resources

Resources

- Online Resources
    - The Documentation: http://llvm.org/docs/
    - Developer Meetings: http://llvm.org/devmtg/
    - Downloading and setting up LLVM: http://llvm.org/docs/GettingStarted.html#checkout
    - An introductory guide: http://adriansampson.net/blog/llvm.html
    - Weekly LLVM Newsletter: http://llvmweekly.org/
        - Developers Mailing List: http://lists.llvm.org/mailman/listinfo/llvm-dev
    - IR Web interface: http://ellcc.org/demo/index.cgi
    - LLVM Blog: http://blog.llvm.org/
    - David Chisnall's course: https://www.cl.cam.ac.uk/teaching/1718/L25/materials.html
- Useful Tools to Try
    - Hexdump (hexdump -c some_bitcode.bc)
    - Meld - Tool for diff'ing and comparing files
    - xdot or graphviz - View .dot files
- Other homework
    - https://cseweb.ucsd.edu/classes/sp14/cse231-a/proj1.html

More Guidance - Your LLVM Syllabus

- June, 3 -- Day 1 (or Today?): https://www.youtube.com/watch?v=a5-WaD8VV38
- June, 4 -- Day 2: Official LLVM Youtube channel
- Extend Program Analysis Knowledge:
  - Youtube series on Program Analysis (Some LLVM Lectures!)
    - https://www.youtube.com/playlist?list=PLNC6lmsIySCOPjY8IwKBtD2cqe-MMgIGM

Contributing to LLVM

https://llvm.org/devmtg/2014-02/slides/ledru-how-to-contribute-to-llvm.pdf

How to contribute to LLVM, Clang, etc

Conclusion

- LLVM is an exciting project with a lot of power
- LLVM or its related projects are likely the ‘right’ tool if you are working on programming languages, performance, or tool building
- If you are still not convinced, your takeaway can still be to look at the codebase, and see some great engineering with the C++ language.
- It’s big, but should not be scary
  - The difficulty that arises is that it is a lot of ‘new’ things
  - You can do it!

Thank You!

@MichaelShah | www.mshah.io

Make sure we save output of opt

- Something new we are doing with this pass, is that it actually is modifying code.
- Occasionally you may see this message

```
mike:examples$ ./../opt -load ./../../lib/LLVMHello.so -hello4 < instrumentDemoText
.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.
```

- In our case, yes we do want to output the modified bitcode file, but this time to a new bitcode file.

Some Gotcha’s

- Having trouble with `llvm-config`?
  - Make sure your PATH variable is updated
  - `export PATH=/home/mike/Desktop/llvm/llvm_build/bin/:$PATH`

Courses Using LLVM

https://www.cs.utexas.edu/users/lin/cs380c/prog1.pdf

Tour of LLVM Project

https://blog.regehr.org/archives/1453 |
http://www.linux.org/threads/llvm-toolset.6644/

Useful debugging things

`dump()` command.

Build your own LLVM language

http://dev.stephendiehl.com/numpile/

LLVM Backend information

https://jonathan2251.github.io/lbd/funccall.html