



Quick answers to common problems

LLVM Cookbook

Over 80 engaging recipes that will help you build a compiler frontend, optimizer, and code generator using LLVM

Mayur Pandey

Suyog Sarda

[PACKT] open source*
PUBLISHING community experience distilled

LLVM Cookbook

Over 80 engaging recipes that will help you build a compiler frontend, optimizer, and code generator using LLVM

Mayur Pandey

Suyog Sarda

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

LLVM Cookbook

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2015

Production reference: 1270515

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-598-1

www.packtpub.com

Credits

Authors

Mayur Pandey
Suyog Sarda

Reviewers

Logan Chien
Michael Haidl
Dave (Jing) Tian

Commissioning Editor

Nadeem N. Bagban

Acquisition Editor

Vivek Anantharaman

Content Development Editor

Shweta Pant

Technical Editors

Prajakta Mhatre
Rohith Rajan
Rupali Shrawane

Copy Editors

Vikrant Phadke
Sameen Siddiqui

Project Coordinator

Shipra Chawhan

Proofreader

Stephen Copestake
Safis Editing

Indexer

Tejal Soni

Graphics

Disha Haria

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Authors

Mayur Pandey is a professional software engineer and an open source enthusiast. He focuses on compiler development and compiler tools. He is an active contributor to the LLVM open source community. He has been part of the compiler team for the Tizen project, and has hands-on experience with other proprietary compilers.

Mayur earned a bachelor's degree in information technology from Motilal Nehru National Institute of Technology Allahabad, India. Currently, he lives in Bengaluru, India.

I would like to thank my family and friends. They made it possible for me to complete the book by taking care of my other commitments and always encouraging me.

Suyog Sarda is a professional software engineer and an open source enthusiast. He focuses on compiler development and compiler tools. He is an active contributor to the LLVM open source community. He has been part of the compiler team for the Tizen project. Suyog was also involved in code performance improvements for the ARM and the x86 architecture. He has hands-on experience in other proprietary compilers. His interest in compiler development lies more in code optimization and vectorization.

Apart from compilers, Suyog is also interested in Linux kernel development. He has published a technical paper titled *Secure Co-resident Virtualization in Multicore Systems by VM Pinning and Page Coloring* at the IEEE Proceedings of the 2012 International Conference on Cloud Computing, Technologies, Applications, and Management at Birla Institute of Technology, Dubai. He earned a bachelor's degree in computer technology from College of Engineering, Pune, India. Currently, he lives in Bengaluru, India.

I would like to thank my family and friends. I would also like to thank the LLVM open-source community for always being helpful.

About the Reviewers

Logan Chien received his master's degree in computer science from National Taiwan University. His research interests include compiler design, compiler optimization, and virtual machines. He is a full-time software engineer. In his free time, he works on several open source projects, such as LLVM and Android. Logan has participated in the LLVM project since 2012.

Michael Haidl is a high performance computing engineer with focus on many core architectures that consist of Graphics Processing Units (GPUs) and Intel Xeon Phi accelerators. He has been a C++ developer for more than 14 years, and has gained many skills in parallel programming, exploiting various programming models (CUDA) over the years. He has a diploma in computer science and physics. Currently, Michael is employed as a research associate at the University of Münster, Germany, and is writing his PhD thesis with focus on compilation techniques for GPUs utilizing the LLVM infrastructure.

I would like to thank my wife for supporting me every day with her smiles and love. I would also like to thank the entire LLVM community for all the hard work they have put into LLVM/Clang and other LLVM projects. It is amazing to see how fast LLVM evolves.

Dave (Jing) Tian is a graduate research fellow and PhD student in the Department of Computer & Information Science & Engineering (CISE) at the University of Florida. He is a founding member of the SENSEI center. His research direction involves system security, embedded system security, trusted computing, static code analysis for security, and virtualization. He is interested in Linux kernel hacking and compiler hacking.

Dave spent a year on AI and machine learning, and taught Python and operating systems at the University of Oregon. Before that, he worked as a software developer in the LCP (Linux control platform) group in research and development at Alcatel-Lucent (formerly Lucent Technologies), for approximately 4 years. He holds a bachelor's degree in science and a master's degree in electronics engineering in China. You can reach him at `root@davejingtian.org` and visit his website <http://davejingtian.org>.

I would like to thank the author of this book, who has done a good job.
Thanks to the editors of the book at Packt Publishing, who made this book perfect and offered me the opportunity to review such a nice book.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: LLVM Design and Use	1
Introduction	1
Understanding modular design	2
Cross-compiling Clang/LLVM	5
Converting a C source code to LLVM assembly	7
Converting IR to LLVM bitcode	9
Converting LLVM bitcode to target machine assembly	11
Converting LLVM bitcode back to LLVM assembly	13
Transforming LLVM IR	14
Linking LLVM bitcode	17
Executing LLVM bitcode	18
Using the C frontend Clang	19
Using the GO frontend	23
Using DragonEgg	24
Chapter 2: Steps in Writing a Frontend	27
Introduction	27
Defining a TOY language	28
Implementing a lexer	29
Defining Abstract Syntax Tree	32
Implementing a parser	35
Parsing simple expressions	36
Parsing binary expressions	38
Invoking a driver for parsing	41
Running lexer and parser on our TOY language	42
Defining IR code generation methods for each AST class	43
Generating IR code for expressions	45

Generating IR code for functions	46
Adding IR optimization support	49
Chapter 3: Extending the Frontend and Adding JIT Support	51
Introduction	51
Handling decision making paradigms – if/then/else constructs	52
Generating code for loops	58
Handling user-defined operators – binary operators	63
Handling user-defined operators – unary operators	68
Adding JIT support	73
Chapter 4: Preparing Optimizations	77
Introduction	77
Various levels of optimization	78
Writing your own LLVM pass	79
Running your own pass with the opt tool	82
Using another pass in a new pass	83
Registering a pass with pass manager	85
Writing an analysis pass	87
Writing an alias analysis pass	90
Using other analysis passes	93
Chapter 5: Implementing Optimizations	97
Introduction	97
Writing a dead code elimination pass	98
Writing an inlining transformation pass	102
Writing a pass for memory optimization	106
Combining LLVM IR	108
Transforming and optimizing loops	110
Reassociating expressions	112
Vectorizing IR	114
Other optimization passes	121
Chapter 6: Target-independent Code Generator	125
Introduction	125
The life of an LLVM IR instruction	126
Visualizing LLVM IR CFG using GraphViz	129
Describing targets using TableGen	136
Defining an instruction set	137
Adding a machine code descriptor	138
Implementing the MachineInstrBuilder class	142
Implementing the MachineBasicBlock class	142
Implementing the MachineFunction class	144

Writing an instruction selector	145
Legalizing SelectionDAG	151
Optimizing SelectionDAG	158
Selecting instruction from the DAG	163
Scheduling instructions in SelectionDAG	170
Chapter 7: Optimizing the Machine Code	173
Introduction	173
Eliminating common subexpression from machine code	174
Analyzing live intervals	184
Allocating registers	190
Inserting the prologue-epilogue code	196
Code emission	200
Tail call optimization	202
Sibling call optimisation	205
Chapter 8: Writing an LLVM Backend	207
Introduction	207
Defining registers and registers sets	208
Defining the calling convention	210
Defining the instruction set	211
Implementing frame lowering	212
Printing an instruction	215
Selecting an instruction	219
Adding instruction encoding	222
Supporting a subtarget	224
Lowering to multiple instructions	226
Registering a target	228
Chapter 9: Using LLVM for Various Useful Projects	241
Introduction	241
Exception handling in LLVM	242
Using sanitizers	247
Writing the garbage collector with LLVM	249
Converting LLVM IR to JavaScript	256
Using the Clang Static Analyzer	257
Using bugpoint	259
Using LLDB	262
Using LLVM utility passes	267
Index	271

Preface

A programmer might have come across compilers at some or the other point when programming. Simply speaking, a compiler converts a human-readable, high-level language into machine-executable code. But have you ever wondered what goes on under the hood? A compiler does lot of processing before emitting optimized machine code. Lots of complex algorithms are involved in writing a good compiler.

This book travels through all the phases of compilation: frontend processing, code optimization, code emission, and so on. And to make this journey easy, LLVM is the simplest compiler infrastructure to study. It's a modular, layered compiler infrastructure where every phase is dished out as a separate recipe. Written in object-oriented C++, LLVM gives programmers a simple interface and lots of APIs to write their own compiler.

As authors, we maintain that simple solutions frequently work better than complex solutions; throughout this book, we'll look at a variety of recipes that will help develop your skills, make you consider all the compiling options, and understand that there is more to simply compiling code than meets the eye.

We also believe that programmers who are not involved in compiler development will benefit from this book, as knowledge of compiler implementation will help them code optimally next time they write code.

We hope you will find the recipes in this book delicious, and after tasting all the recipes, you will be able to prepare your own dish of compilers. Feeling hungry? Let's jump into the recipes!

What this book covers

Chapter 1, LLVM Design and Use, introduces the modular world of LLVM infrastructure, where you learn how to download and install LLVM and Clang. In this chapter, we play with some examples to get accustomed to the workings of LLVM. We also see some examples of various frontends.

Chapter 2, Steps in Writing a Frontend, explains the steps to write a frontend for a language. We will write a bare-metal toy compiler frontend for a basic toy language. We will also see how a frontend language can be converted into the LLVM intermediate representation (IR).

Chapter 3, Extending the Frontend and Adding JIT Support, explores the more advanced features of the toy language and the addition of JIT support to the frontend. We implement some powerful features of a language that are found in most modern programming languages.

Chapter 4, Preparing Optimizations, takes a look at the pass infrastructure of the LLVM IR. We explore various optimization levels, and the optimization techniques kicking at each level. We also see a step-by-step approach to writing our own LLVM pass.

Chapter 5, Implementing Optimizations, demonstrates how we can implement various common optimization passes on LLVM IR. We also explore some vectorization techniques that are not yet present in the LLVM open source code.

Chapter 6, Target-independent Code Generator, takes us on a journey through the abstract infrastructure of a target-independent code generator. We explore how LLVM IR is converted to Selection DAGs, which are further processed to emit target machine code.

Chapter 7, Optimizing the Machine Code, examines how Selection DAGs are optimized and how target registers are allocated to variables. This chapter also describes various optimization techniques on Selection DAGs as well as various register allocation techniques.

Chapter 8, Writing an LLVM Backend, takes us on a journey of describing a target architecture. This chapter covers how to describe registers, instruction sets, calling conventions, encoding, subtarget features, and so on.

Chapter 9, Using LLVM for Various Useful Projects, explores various other projects where LLVM IR infrastructure can be used. Remember that LLVM is not just a compiler; it is a compiler infrastructure. This chapter explores various projects that can be applied to a code snippet to get useful information from it.

What you need for this book

All you need to work through most of the examples covered in this book is a Linux machine, preferably Ubuntu. You will also need a simple text or code editor, Internet access, and a browser. We recommend installing the `meld` tool for comparison of two files; it works well on the Linux platform.

Who this book is for

The book is for compiler programmers who are familiar with concepts of compilers and want to indulge in understanding, exploring, and using LLVM infrastructure in a meaningful way in their work.

This book is also for programmers who are not directly involved in compiler projects but are often involved in development phases where they write thousands of lines of code. With knowledge of how compilers work, they will be able to code in an optimal way and improve performance with clean code.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections.

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
primary := identifier_expr
:=numeric_expr
:=paran_expr
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
primary := identifier_expr
:=numeric_expr
:=paran_expr
```

Any command-line input or output is written as follows:

```
$ cat testfile.ll
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/59810S_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

LLVM Design and Use

In this chapter, we will cover the following topics:

- ▶ Understanding modular design
- ▶ Cross-compiling Clang/LLVM
- ▶ Converting a C source code to LLVM assembly
- ▶ Converting IR to LLVM bitcode
- ▶ Converting LLVM bitcode to target machine assembly
- ▶ Converting LLVM bitcode back to LLVM assembly
- ▶ Transforming LLVM IR
- ▶ Linking LLVM bitcode
- ▶ Executing LLVM bitcode
- ▶ Using C frontend Clang
- ▶ Using the GO frontend
- ▶ Using DragonEgg

Introduction

In this recipe, you get to know about **LLVM**, its design, and how we can make multiple uses out of the various tools it provides. You will also look into how you can transform a simple C code to the LLVM intermediate representation and how you can transform it into various forms. You will also learn how the code is organized within the LLVM source tree and how can you use it to write a compiler on your own later.

Understanding modular design

LLVM is designed as a set of libraries unlike other compilers such as **GNU Compiler Collection (GCC)**. In this recipe, LLVM optimizer will be used to understand this design. As LLVM optimizer's design is library-based, it allows you to order the passes to be run in a specified order. Also, this design allows you to choose which optimization passes you can run—that is, there might be a few optimizations that might not be useful to the type of system you are designing, and only a few optimizations will be specific to the system. When looking at traditional compiler optimizers, they are built as a tightly interconnected mass of code, that is difficult to break down into small parts that you can understand and use easily. In LLVM, you need not know about how the whole system works to know about a specific optimizer. You can just pick one optimizer and use it without having to worry about other components attached to it.

Before we go ahead and look into this recipe, we must also know a little about LLVM assembly language. The LLVM code is represented in three forms: in memory compiler **Intermediate Representation (IR)**, on disk bitcode representation, and as human readable assembly. LLVM is a **Static Single Assignment (SSA)**-based representation that provides type safety, low level operations, flexibility, and the capability to represent all the high-level languages cleanly. This representation is used throughout all the phases of LLVM compilation strategy. The LLVM representation aims to be a universal IR by being at a low enough level that high-level ideas may be cleanly mapped to it. Also, LLVM assembly language is well formed. If you have any doubts about understanding the LLVM assembly mentioned in this recipe, refer to the link provided in the *See also* section at the end of this recipe.

Getting ready

We must have installed the LLVM toolchain on our host machine. Specifically, we need the `opt` tool.

How to do it...

We will run two different optimizations on the same code, one-by-one, and see how it modifies the code according to the optimization we choose.

1. First of all, let us write a code we can input for these optimizations. Here we will write it into a file named `testfile.ll`:

```
$ cat testfile.ll
define i32 @test1(i32 %A) {
    %B = add i32 %A, 0
    ret i32 %B
}
```

```
define internal i32 @test(i32 %X, i32 %dead) {  
    ret i32 %X  
}
```

```
define i32 @caller() {  
    %A = call i32 @test(i32 123, i32 456)  
    ret i32 %A  
}
```

2. Now, run the `opt` tool for one of the optimizations—that is, for combining the instruction:

```
$ opt -S -instcombine testfile.ll -o output1.ll
```

3. View the output to see how `instcombine` has worked:

```
$ cat output1.ll  
; ModuleID = 'testfile.ll'
```

```
define i32 @test1(i32 %A) {  
    ret i32 %A  
}
```

```
define internal i32 @test(i32 %X, i32 %dead) {  
    ret i32 %X  
}
```

```
define i32 @caller() {  
    %A = call i32 @test(i32 123, i32 456)  
    ret i32 %A  
}
```

4. Run the `opt` command for dead argument elimination optimization:

```
$ opt -S -deadargelim testfile.ll -o output2.ll
```


5. View the output, to see how `deadargelim` has worked:

```
$ cat output2.ll
; ModuleID = testfile.ll

define i32 @test1(i32 %A) {
    %B = add i32 %A, 0
    ret i32 %B
}

define internal i32 @test(i32 %X) {
    ret i32 %X
}

define i32 @caller() {
    %A = call i32 @test(i32 123)
    ret i32 %A
}
```

How it works...

In the preceding example, we can see that, for the first command, the `instcombine` pass is run, which combines the instructions and hence optimizes `%B = add i32 %A, 0; ret i32 %B` to `ret i32 %A` without affecting the code.

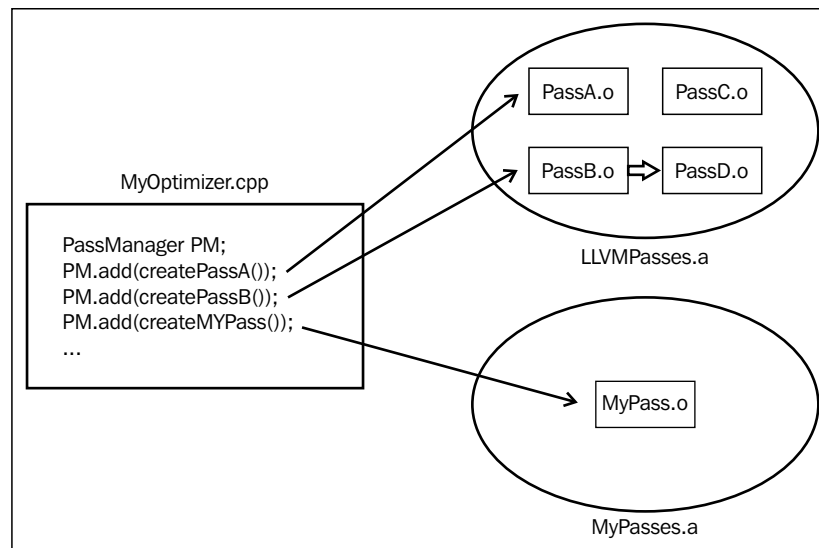
In the second case, when the `deadargelim` pass is run, we can see that there is no modification in the first function, but the part of code that was not modified last time gets modified with the function arguments that are not used getting eliminated.

LLVM optimizer is the tool that provided the user with all the different passes in LLVM. These passes are all written in a similar style. For each of these passes, there is a compiled object file. Object files of different passes are archived into a library. The passes within the library are not strongly connected, and it is the LLVM **PassManager** that has the information about dependencies among the passes, which it resolves when a pass is executed. The following image shows how each pass can be linked to a specific object file within a specific library. In the following figure, the **PassA** references **LLVMPasses.a** for **PassA.o**, whereas the custom pass refers to a different library **MyPasses.a** for the **MyPass.o** object file.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.



There's more...

Similar to the optimizer, the LLVM code generator also makes use of its modular design, splitting the code generation problem into individual passes: instruction selection, register allocation, scheduling, code layout optimization, and assembly emission. Also, there are many built-in passes that are run by default. It is up to the user to choose which passes to run.

See also

- ▶ In the upcoming chapters, we will see how to write our own custom pass, where we can choose which of the optimization passes we want to run and in which order. Also, for a more detailed understanding, refer to <http://www.aosabook.org/en/llvm.html>.
- ▶ To understand more about LLVM assembly language, refer to <http://llvm.org/docs/LangRef.html>.

Cross-compiling Clang/LLVM

By cross-compiling we mean building a binary on one platform (for example, x86) that will be run on another platform (for example, ARM). The machine on which we build the binary is called the **host**, and the machine on which the generated binary will run is called the **target**. The compiler that builds code for the same platform on which it is running (the host and target platforms are the same) is called a **native assembler**, whereas the compiler that builds code for a target platform different from the host platform is called a **cross-compiler**.

In this recipe, cross-compilation of LLVM for a platform different than the host platform will be shown, so that you can use the built binaries for the required target platform. Here, cross-compiling will be shown using an example where cross-compilation from host platform `x86_64` for target platform ARM will be done. The binaries thus generated can be used on a platform with ARM architecture.

Getting ready

The following packages need to be installed on your system (host platform):

- ▶ `cmake`
- ▶ `ninja-build` (from backports in Ubuntu)
- ▶ `gcc-4.x-arm-linux-gnueabi`
- ▶ `gcc-4.x-multilib-arm-linux-gnueabi`
- ▶ `binutils-arm-linux-gnueabi`
- ▶ `libgcc1-armhf-cross`
- ▶ `libsfgcc1-armhf-cross`
- ▶ `libstdc++6-armhf-cross`
- ▶ `libstdc++6-4.x-dev-armhf-cross`
- ▶ install llvm on your host platform

How to do it...

To compile for the ARM target from the host architecture, that is **X86_64** here, you need to perform the following steps:

1. Add the following `cmake` flags to the normal `cmake` build for LLVM:

```
-DCMAKE_CROSSCOMPILING=True  
-DCMAKE_INSTALL_PREFIX= path-where-you-want-the-  
toolchain(optional)  
-DLLVM_TABLEGEN=<path-to-host-installed-llvm-toolchain-bin>/llvm-  
tblgen  
-DCLANG_TABLEGEN=< path-to-host-installed-llvm-toolchain-bin >/  
clang-tblgen  
-DLLVM_DEFAULT_TARGET_TRIPLE=arm-linux-gnueabi  
-DLLVM_TARGET_ARCH=ARM  
-DLLVM_TARGETS_TO_BUILD=ARM  
-DCMAKE_CXX_FLAGS='-target armv7a-linux-  
gnueabi -mcpu=cortex-a9 -I/usr/arm-linux-gnueabi/include/  
c++/4.x.x/arm-linux-gnueabi/ -I/usr/arm-linux-gnueabi/  
include/ -mfloat-abi=hard -ccc-gcc-name arm-linux-gnueabi-gcc'
```

2. If using your platform compiler, run:

```
$ cmake -G Ninja <llvm-source-dir> <options above>
```

If using Clang as the cross-compiler, run:

```
$ CC='clang' CXX='clang++' cmake -G Ninja <source-dir> <options above>
```

If you have clang/Clang++ on the path, it should work fine.

3. To build LLVM, simply type:

```
$ ninja
```

4. After the LLVM/Clang has built successfully, install it with the following command:

```
$ ninja install
```

This will create a `sysroot` on the `install-dir` location if you have specified the `DCMAKE_INSTALL_PREFIX` options

How it works...

The `cmake` package builds the toolchain for the required platform by making the use of option flags passed to `cmake`, and the `tblgen` tools are used to translate the target description files into C++ code. Thus, by using it, the information about targets is obtained, for example—what instructions are available on the target, the number of registers, and so on.



If Clang is used as the cross-compiler, there is a problem in the LLVM ARM backend that produces absolute relocations on **position-independent code (PIC)**, so as a workaround, disable PIC at the moment.

The ARM libraries will not be available on the host system. So, either download a copy of them or build them on your system.

Converting a C source code to LLVM assembly

Here we will convert a C code to intermediate representation in LLVM using the C frontend Clang.

Getting ready

Clang must be installed in the PATH.

How to do it...

1. Lets create a C code in the `multiply.c` file, which will look something like the following:

```
$ cat multiply.c
int mult() {
    int a =5;
    int b = 3;
    int c = a * b;
    return c;
}
```

2. Use the following command to generate LLVM IR from the C code:

```
$ clang -emit-llvm -S multiply.c -o multiply.ll
```

3. Have a look at the generated IR:

```
$ cat multiply.ll
; ModuleID = 'multiply.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @mult() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 5, i32* %a, align 4
    store i32 3, i32* %b, align 4
    %1 = load i32* %a, align 4
    %2 = load i32* %b, align 4
    %3 = mul nsw i32 %1, %2
    store i32 %3, i32* %c, align 4
    %4 = load i32* %c, align 4
    ret i32 %4
}
```

We can also use the `cc1` for generating IR:

```
$ clang -cc1 -emit-llvm testfile.c -o testfile.ll
```

How it works...

The process of C code getting converted to IR starts with the process of lexing, wherein the C code is broken into a token stream, with each token representing an Identifier, Literal, Operator, and so on. This stream of tokens is fed to the parser, which builds up an abstract syntax tree with the help of **Context free grammar (CFG)** for the language. Semantic analysis is done afterwards to check whether the code is semantically correct, and then we generate code to IR.

Here we use the Clang frontend to generate the IR file from C code.

See also

- In the next chapter, we will see how the lexer and parser work and how code generation is done. To understand the basics of LLVM IR, you can refer to <http://llvm.org/docs/LangRef.html>.

Converting IR to LLVM bitcode

In this recipe, you will learn to generate LLVM bit code from IR. The LLVM bit code file format (also known as bytecode) is actually two things: a bitstream container format and an encoding of LLVM IR into the container format.

Getting Ready

The `llvm-as` tool must be installed in the PATH.

How to do it...

Do the following steps:

1. First create an IR code that will be used as input to `llvm-as`:

```
$ cat test.ll
define i32 @mult(i32 %a, i32 %b) #0 {
    %1 = mul nsw i32 %a, %b
    ret i32 %1
}
```

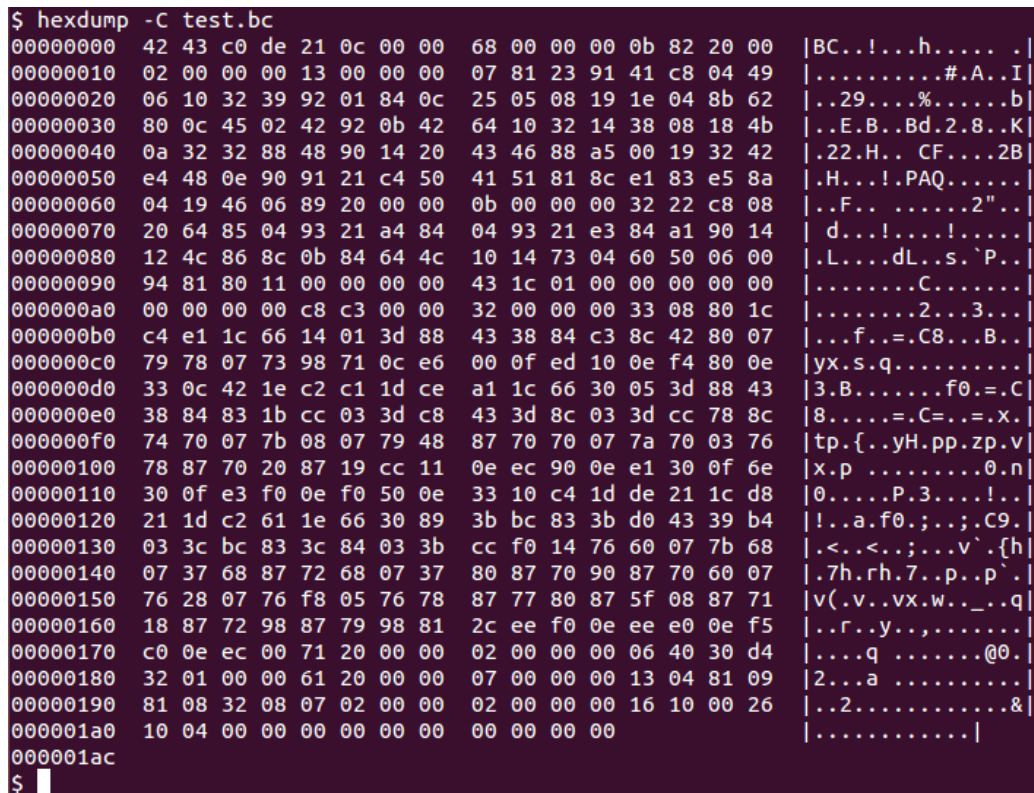
- To convert LLVM IR in `test.ll` to bitcode format, you need to use the following command:

```
llvm-as test.ll -o test.bc
```
- The output is generated in the `test.bc` file, which is in bit stream format; so, when we want to have a look at output in text format, we get it as shown in the following screenshot:



```
$ cat test.bc
BC...!...h.... .|
.....#.A..I|
..29....%....b|
..E.B..Bd.2.8..K|
..22.H.. CF....2B|
.H...!.PAQ.....|
..F.. ....2"..|
d...!.....!....|
.L....dL..s.`P..|
.....C.....|
.....2...3...|
...f...=.C8...B..|
yx.s.q.....|
3.B.....f0.=.C|
8.....=.C=...X.|
tp.{.yH.pp.zp.v|
x.p .....0.n|
0....P.3....!..|
!...a.f0.;...C9.|
|<...<...v`.f|h|
|.7h.rh.7..p..p`.|
|v(.v..vx.w.._...q|
|..r..y...|
|...q .....@0.|
|2...a .....|
|..2.....&|
|.....|
000001ac
$
```

Since this is a bitcode file, the best way to view its content would be by using the `hexdump` tool. The following screenshot shows the output of `hexdump`:



```
$ hexdump -C test.bc
00000000  42 43 c0 de 21 0c 00 00  68 00 00 00 0b 82 20 00  |BC...!...h.... .|
00000010  02 00 00 00 13 00 00 00  07 81 23 91 41 c8 04 49  |.....#.A..I|
00000020  06 10 32 39 92 01 84 0c  25 05 08 19 1e 04 8b 62  |..29....%....b|
00000030  80 0c 45 02 42 92 0b 42  64 10 32 14 38 08 18 4b  |..E.B..Bd.2.8..K|
00000040  0a 32 32 88 48 90 14 20  43 46 88 a5 00 19 32 42  |..22.H.. CF....2B|
00000050  e4 48 0e 90 91 21 c4 50  41 51 81 8c e1 83 e5 8a  |.H...!.PAQ.....|
00000060  04 19 46 06 89 20 00 00  0b 00 00 00 32 22 c8 08  |..F.. ....2"..|
00000070  20 64 85 04 93 21 a4 84  04 93 21 e3 84 a1 90 14  |d...!.....!....|
00000080  12 4c 86 8c 0b 84 64 4c  10 14 73 04 60 50 06 00  |.L....dL..s.`P..|
00000090  94 81 80 11 00 00 00 00  43 1c 01 00 00 00 00 00  |.....C.....|
000000a0  00 00 00 00 c8 c3 00 00  32 00 00 00 33 08 80 1c  |.....2...3...|
000000b0  c4 e1 1c 66 14 01 3d 88  43 38 84 c3 8c 42 80 07  |...f...=.C8...B..|
000000c0  79 78 07 73 98 71 0c e6  00 0f ed 10 0e f4 80 0e  |yx.s.q.....|
000000d0  33 0c 42 1e c2 c1 1d ce  a1 1c 66 30 05 3d 88 43  |3.B.....f0.=.C|
000000e0  38 84 83 1b cc 03 3d c8  43 3d 8c 03 3d cc 78 8c  |8.....=.C=...X.|
000000f0  74 70 07 7b 08 07 79 48  87 70 70 07 7a 70 03 76  |tp.{.yH.pp.zp.v|
00000100  78 87 70 20 87 19 cc 11  0e ec 90 0e e1 30 0f 6e  |x.p .....0.n|
00000110  30 0f e3 f0 0e f0 50 0e  33 10 c4 1d de 21 1c d8  |0....P.3....!..|
00000120  21 1d c2 61 1e 66 30 89  3b bc 83 3b d0 43 39 b4  |!...a.f0.;...C9.|
00000130  03 3c bc 83 3c 84 03 3b  cc f0 14 76 60 07 7b 68  ||<...<...v`.f|h|
00000140  07 37 68 87 72 68 07 37  80 87 70 90 87 70 60 07  ||.7h.rh.7..p..p`.|
00000150  76 28 07 76 f8 05 76 78  87 77 80 87 5f 08 87 71  ||v(.v..vx.w.._...q|
00000160  18 87 72 98 87 79 98 81  2c ee f0 0e ee e0 0e f5  ||..r..y...|
00000170  c0 0e ec 00 71 20 00 00  02 00 00 00 06 40 30 d4  ||...q .....@0.|
00000180  32 01 00 00 61 20 00 00  07 00 00 00 13 04 81 09  ||2...a .....|
00000190  81 08 32 08 07 02 00 00  02 00 00 00 16 10 00 26  ||..2.....&|
000001a0  10 04 00 00 00 00 00 00  00 00 00 00                |.....|
000001ac
$
```


How it works...

The `llvm-as` is the LLVM assembler. It converts the LLVM assembly file that is the LLVM IR into LLVM bitcode. In the preceding command, it takes the `test.ll` file as the input and outputs, and `test.bc` as the bitcode file.

There's more...

To encode LLVM IR into bitcode, the concept of blocks and records is used. Blocks represent regions of bitstream, for example—a function body, symbol table, and so on. Each block has an ID specific to its content (for example, function bodies in LLVM IR are represented by ID 12). Records consist of a record code and an integer value, and they describe the entities within the file such as instructions, global variable descriptors, type descriptions, and so on.

Bitcode files for LLVM IR might be wrapped in a simple wrapper structure. This structure contains a simple header that indicates the offset and size of the embedded BC file.

See also

- ▶ To get a detailed understanding of the LLVM the bitstream file format, refer to <http://llvm.org/docs/BitCodeFormat.html#abstract>

Converting LLVM bitcode to target machine assembly

In this recipe, you will learn how to convert the LLVM bitcode file to target specific assembly code.

Getting ready

The LLVM static compiler `llc` should be installed from the LLVM toolchain.

How to do it...

Do the following steps:

1. The bitcode file created in the previous recipe, `test.bc`, can be used as input to `llc` here. Using the following command, we can convert LLVM bitcode to assembly code:

```
$ llc test.bc -o test.s
```