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] PUBLISHING open source 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

BIRMINGHAM - MUMBAI

Table of Contents

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...

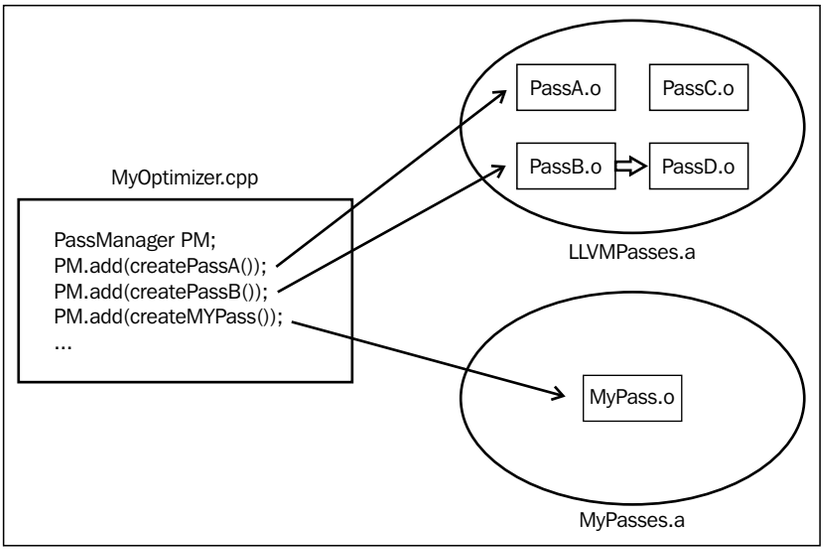
In the preceding example, we can see that, for the first command, the `instcombine` pass is run, which combines the instructions and hence optimizes `%B = add i32 %A, 0; ret i32 %B` to `ret i32 %A` without affecting the code.

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-gnueabihf`
- `gcc-4.x-multilib-arm-linux-gnueabihf`
- `binutils-arm-linux-gnueabihf`
- `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-gnueabihf
-DLLVM_TARGET_ARCH=ARM
-DLLVM_TARGETS_TO_BUILD=ARM
-DCMAKE_CXX_FLAGS='-target armv7a-linux-
gnueabihf -mcpu=cortex-a9 -I/usr/arm-linux-gnueabihf/include/
c++/4.x.x/arm-linux-gnueabihf/ -I/usr/arm-linux-gnueabihf/
include/ -mfloat-abi=hard -ccc-gcc-name arm-linux-gnueabihf-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
}
```

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

   `llvm-as test.ll –o test.bc`

3. 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:

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`.{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 in 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
```

2. The output is generated in the `test.s` file, which is the assembly code. To have a look at that, use the following command lines:

```
$ cat test.s
.text
.file "test.bc"
.globl mult
.align 16, 0x90
.type mult,@function
mult:                                   # @mult
.cfi_startproc
# BB#0:
Pushq  %rbp
.Ltmp0:
.cfi_def_cfa_offset 16
.Ltmp1:
.cfi_offset %rbp, -16
movq %rsp, %rbp
.Ltmp2:
.cfi_def_cfa_register %rbp
imull %esi, %edi
movl %edi, %eax
popq %rbp
retq
.Ltmp3:
.size mult, .Ltmp3-mult
.cfi_endproc
```

3. You can also use Clang to dump assembly code from the bitcode file format. By passing the `-S` option to Clang, we get `test.s` in assembly format when the `test.bc` file is in bitstream file format:

```
$ clang -S test.bc -o test.s –fomit-frame-pointer # using the
clang front end
```

The `test.s` file output is the same as that of the preceding example. We use the additional option `fomit-frame-pointer`, as Clang by default does not eliminate the frame pointer whereas `llc` eliminates it by default.

How it works...

The `llc` command compiles LLVM input into assembly language for a specified architecture. If we do not mention any architecture as in the preceding command, the assembly will be generated for the host machine where the `llc` command is being used. To generate executable from this assembly file, you can use assembler and linker.

There's more...

By specifying `-march=architecture flag` in the preceding command, you can specify the target architecture for which the assembly needs to be generated. Using the `-mcpu=cpu flag` setting, you can specify a CPU within the architecture to generate code. Also by specifying `-regalloc=basic/greedy/fast/pbqp`, you can specify the type of register allocation to be used.

Converting LLVM bitcode back to LLVM assembly

In this recipe, you will convert LLVM bitcode back to LLVM IR. Well, this is actually possible using the LLVM disassembler tool called `llvm-dis`.

Getting ready

To do this, you need the `llvm-dis` tool installed.

How to do it...

To see how the bitcode file is getting converted to IR, use the `test.bc` file generated in the recipe *Converting IR to LLVM Bitcode*. The `test.bc` file is provided as the input to the `llvm-dis` tool. Now proceed with the following steps:

1. Using the following command shows how to convert a bitcode file to an the one we had created in the IR file:

```
$ llvm-dis test.bc –o test.ll
```

2. Have a look at the generated LLVM IR by the following:

```
| $ cat test.ll
; ModuleID = 'test.bc'

define i32 @mult(i32 %a, i32 %b) #0 {
  %1 = mul nsw i32 %a, %b
```

```
  ret i32 %1
}
```

The output `test.ll` file is the same as the one we created in the recipe *Converting IR to LLVM Bitcode*.

How it works...

The `llvm-dis` command is the LLVM disassembler. It takes an LLVM bitcode file and converts it into LLVM assembly language.

Here, the input file is `test.bc`, which is transformed to `test.ll` by `llvm-dis`.

If the filename is omitted, `llvm-dis` reads its input from standard input.

Transforming LLVM IR

In this recipe, we will see how we can transform the IR from one form to another using the opt tool. We will see different optimizations being applied to IR code.

Getting ready

You need to have the opt tool installed.

How to do it...

The `opt` tool runs the transformation pass as in the following command:

```
$opt -passname input.ll -o output.ll
```

1. Let's take an actual example now. We create the LLVM IR equivalent to the C code used in the recipe *Converting a C source code to LLVM assembly*:

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

2. Converting and outputting it, we get the unoptimized output:

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

3. Now use the opt tool to transform it to a form where memory is promoted to register:

```
$ opt -mem2reg -S multiply.ll -o multiply1.ll
$ cat multiply1.ll
; ModuleID = 'multiply.ll'
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(i32 %a, i32 %b) #0 {
  %1 = mul nsw i32 %a, %b
  ret i32 %1
}
```

How it works...

The `opt`, LLVM optimizer, and analyzer tools take the `input.ll` file as the input and run the pass `passname` on it. The output after running the pass is obtained in the `output.ll` file that contains the IR code after the transformation. There can be more than one pass passed to the opt tool.

There's more...

When the `-analyze` option is passed to opt, it performs various analyses of the input source and prints results usually on the standard output or standard error. Also, the output can be redirected to a file when it is meant to be fed to another program.

When the –analyze option is not passed to opt, it runs the transformation passes meant to optimize the input file.

Some of the important transformations are listed as follows, which can be passed as a flag to the opt tool:

- `adce`: Aggressive Dead Code Elimination
- `bb-vectorize`: Basic-Block Vectorization
- `constprop`: Simple constant propagation
- `dce`: Dead Code Elimination
- `deadargelim`: Dead Argument Elimination
- `globaldce`: Dead Global Elimination
- `globalopt`: Global Variable Optimizer
- `gvn`: Global Value Numbering
- `inline`: Function Integration/Inlining
- `instcombine`: Combine redundant instructions
- `licm`: Loop Invariant Code Motion
- `loop`: unswitch: Unswitch loops
- `loweratomic`: Lower atomic intrinsics to non-atomic form
- `lowerinvoke`: Lower invokes to calls, for unwindless code generators
- `lowerswitch`: Lower SwitchInsts to branches
- `mem2reg`: Promote Memory to Register
- `memcpyopt`: MemCpy Optimization
- `simplifycfg`: Simplify the CFG
- `sink`: Code sinking
- `tailcallelim`: Tail Call Elimination

Run at least some of the preceding passes to get an understanding of how they work. To get to the appropriate source code on which these passes might be applicable, go to the `llvm/test/Transforms` directory. For each of the above mentioned passes, you can see the test codes. Apply the relevant pass and see how the test code is getting modified.

To see the mapping of how C code is converted to IR, after converting the C code to IR, as discussed in an earlier recipe *Converting a C source code to LLVM assembly*, run the `mem2reg` pass. It will then help you understand how a C instruction is getting mapped into IR instructions.

Linking LLVM bitcode

In this section, you will link previously generated `.bc` files to get one single bitcode file containing all the needed references.

Getting ready

To link the `.bc` files, you need the `llvm-link` tool.

How to do it...

Do the following steps:

1. To show the working of `llvm-link`, first write two codes in different files, where one makes a reference to the other:

```
$ cat test1.c
int func(int a) {
a = a*2;
return a;
}
$ cat test2.c
#include<stdio.h>
extern int func(int a);
int main() {
int num = 5;
num = func(num);
printf("number is %d\n", num);
return num;
}
```

2. Using the following formats to convert this C code to bitstream file format, first convert to `.ll` files, then from `.ll` files to `.bc` files:

```
$ clang -emit-llvm -S test1.c -o test1.ll
$ clang -emit-llvm -S test2.c -o test2.ll
$ llvm-as test1.ll -o test1.bc
$ llvm-as test2.ll -o test2.bc
```

   We get `test1.bc` and `test2.bc` with `test2.bc` making a reference to `func` syntax in the `test1.bc` file.

3. Invoke the `llvm-link` command in the following way to link the two LLVM bitcode files:

```
$ llvm-link test1.bc test2.bc –o output.bc
```

We provide multiple bitcode files to the `llvm-link` tool, which links them together to generate a single bitcode file. Here, `output.bc` is the generated output file. We will execute this bitcode file in the next recipe *Executing LLVM bitcode*.

How it works...

The `llvm-link` works using the basic functionality of a linker—that is, if a function or variable referenced in one file is defined in the other file, it is the job of linker to resolve all the references made in a file and defined in the other file. But note that this is not the traditional linker that links various object files to generate a binary. The `llvm-link` tool links bitcode files only.

In the preceding scenario, it is linking `test1.bc` and `test2.bc` files to generate the `output.bc` file, which has references resolved.

After linking the bitcode files, we can generate the output as an IR file by giving `–S` option to the `llvm-link` tool.

Executing LLVM bitcode

In this recipe, you will execute the LLVM bitcode that was generated in previous recipes.

Getting ready

To execute the LLVM bitcode, you need the `lli` tool.

How to do it...

We saw in the previous recipe how to create a single bitstream file after linking the two `.bc` files with one referencing the other to define `func`. By invoking the `lli` command in the following way, we can execute the `output.bc` file generated. It will display the output on the standard output:

```
$ lli output.bc
   number is 10
```

`The output.bc` file is the input to `lli`, which will execute the bitcode file and display the output, if any, on the standard output. Here the output is generated as number is `10`, which is a result of the execution of the `output.bc` file formed by linking `test1.c` and `test2.c` in the previous recipe. The main function in the `test2.c` file calls the function `func` in the `test1.c` file with integer 5 as the argument to the function. The `func` function doubles the input argument and returns the result to main the function that outputs it on the standard output.

How it works...

The `lli` tool command executes the program present in LLVM bitcode format. It takes the input in LLVM bitcode format and executes it using a just-in-time compiler, if there is one available for the architecture, or an interpreter.

If `lli` is making use of a just-in-time compiler, then it effectively takes all the code generator options as that of `llc`.

See also

- The *Adding JIT support for a language* recipe in *Chapter 3*, *Extending the Frontend and Adding JIT support*.

Using the C frontend Clang

In this recipe, you will get to know how the Clang frontend can be used for different purposes.

Getting ready

You will need Clang tool.

How to do it...

Clang can be used as the high-level compiler driver. Let us show it using an example:

1. Create a `hello world` C code, `test.c`:

```
$ cat test.c
#include<stdio.h>
int main() {
printf("hello world\n");
return 0; }
```

2. Use Clang as a compiler driver to generate the executable `a.out` file, which on execution gives the output as expected:

```
$ clang test.c
$ ./a.out
hello world
```

   Here the `test.c` file containing C code is created. Using Clang we compile it and produce an executable that on execution gives the desired result.

3. Clang can be used in preprocessor only mode by providing the `-E` flag. In the following example, create a C code having a #define directive defining the value of MAX and use this MAX as the size of the array you are going to create:

```
$ cat test.c
#define MAX 100
void func() {
int a[MAX];
}
```

4. Run the preprocessor using the following command, which gives the output on standard output:

```
$ clang test.c -E
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 308 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
```

```
void func() {
int a[100];
}
```

In the `test.c` file, which will be used in all the subsequent sections of this recipe, MAX is defined to be `100`, which on preprocessing is substituted to MAX in `a[MAX]`, which becomes `a[100]`.

5. You can print the AST for the `test.c` file from the preceding example using the following command, which displays the output on standard output:

```
| $ clang -cc1 test.c -ast-dump
TranslationUnitDecl 0x3f72c50 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x3f73148 <<invalid sloc>> <invalid sloc> implicit
__int128_t '__int128'
|-TypedefDecl 0x3f731a8 <<invalid sloc>> <invalid sloc> implicit
__uint128_t 'unsigned __int128'
|-TypedefDecl 0x3f73518 <<invalid sloc>> <invalid sloc> implicit
__builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x3f735b8 <test.c:3:1, line:5:1> line:3:6 func
'void ()'
  `-CompoundStmt 0x3f73790 <col:13, line:5:1>
    `-DeclStmt 0x3f73778 <line:4:1, col:11>
      `-VarDecl 0x3f73718 <col:1, col:10> col:5 a 'int [100]'
```

Here, the `-cc1` option ensures that only the compiler front-end should be run, not the driver, and it prints the AST corresponding to the `test.c` file code.

6. You can generate the LLVM assembly for the `test.c` file in previous examples, using the following command:

```
|$ clang test.c -S -emit-llvm -o -
|; ModuleID = 'test.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 void @func() #0 {
|%a = alloca [100 x i32], align 16
|ret void
|}
```

The `-S` and `-emit-llvm` flag ensure the LLVM assembly is generated for the `test.c` code.

7. To get machine code use for the same `test.c` testcode, pass the `-S` flag to Clang. It generates the output on standard output because of the option `-o -`:

```
$ clang -S test.c -o -
      .text
      .file  "test.c"
      .globl func
      .align 16, 0x90
      .type  func,@function
func:                                   # @func
      .cfi_startproc
# BB#0:
      pushq  %rbp
.Ltmp0:
      .cfi_def_cfa_offset 16
.Ltmp1:
      .cfi_offset %rbp, -16
      movq   %rsp, %rbp
.Ltmp2:
      .cfi_def_cfa_register %rbp
      popq   %rbp
      retq
.Ltmp3:
      .size  func, .Ltmp3-func
      .cfi_endproc
```

When the `-S` flag is used alone, machine code is generated by the code generation process of the compiler. Here, on running the command, machine code is output on the standard output as we use `-o -` options.

How it works...

Clang works as a preprocessor, compiler driver, frontend, and code generator in the preceding examples, thus giving the desired output as per the input flag given to it.

See also

- This was a basic introduction to how Clang can be used. There are also many other flags that can be passed to Clang, which makes it perform different operation. To see the list, use Clang `-help`.

Using the GO frontend

The `llgo` compiler is the LLVM-based frontend for Go written in Go language only. Using this frontend, we can generate the LLVM assembly code from a program written in Go.

Getting ready

You need to download the `llgo` binaries or build `llgo` from the source code and add the binaries in the `PATH` file location as configured.

How to do it...

Do the following steps:

1. Create a Go source file, for example, that will be used for generating the LLVM assembly using `llgo`. Create `test.go`:

```
$ cat test.go
package main
import "fmt"
func main() {
 fmt.Println("Test Message")
}
```

2. Now, use `llgo` to get the LLVM assembly:

```
$llgo -dump test.go
; ModuleID = 'main'
target datalayout = "e-p:64:64:64..."
target triple = "x86_64-unknown-linux"
%0 = type { i8*, i8* }
....
```

How it works...

The `llgo` compiler is the frontend for the Go language; it takes the `test.go` program as its input and emits the LLVM IR.

See also

- For information about how to get and install `llgo`, refer to `https://github.com/go-llvm/llgo`

Using DragonEgg

Dragonegg is a gcc plugin that allows gcc to make use of the LLVM optimizer and code generator instead of gcc's own optimizer and code generator.

Getting ready

You need to have gcc 4.5 or above, with the target machine being `x86-32/x86-64` and an ARM processor. Also, you need to download the dragonegg source code and build the `dragonegg.so` file.

How to do It...

Do the following steps:

1. Create a simple `hello world` program:

```
$ cat testprog.c
#include<stdio.h>
int main() {
printf("hello world");
}
```

2. Compile this program with your gcc; here we use gcc-4.5:

```
$ gcc testprog.c -S -O1 -o -
  .file  " testprog.c"
  .section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
  .string  "Hello world!"
  .text
.globl main
  .type  main, @function
main:
  subq  $8, %rsp
  movl  $.LC0, %edi
  call  puts
  movl  $0, %eax
  addq  $8, %rsp
  ret
  .size  main, .-main
```

3. Using the `-fplugin=path/dragonegg.so` flag in the command line of gcc makes gcc use LLVM's optimizer and LLVM codegen:

```
$ gcc testprog.c -S -O1 -o - -fplugin=./dragonegg.so
  .file  " testprog.c"
# Start of file scope inline assembly
  .ident  "GCC: (GNU) 4.5.0 20090928 (experimental) LLVM:
82450:82981"
# End of file scope inline assembly



  .text
  .align  16
  .globl  main
  .type  main,@function
main:
  subq  $8, %rsp
  movl  $.L.str, %edi
  call  puts
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .size  main, .-main
  .type  .L.str,@object
  .section  .rodata.str1.1,"aMS",@progbits,1
.L.str:
  .asciz  "Hello world!"
  .size  .L.str, 13

  .section  .note.GNU-stack,"",@progbits
```

See also

- To know about how to get the source code and installation procedure, refer to `http://dragonegg.llvm.org/`

2
Steps in Writing a Frontend

In this chapter, we will cover the following recipes:

- Defining a TOY language
- Implementing a lexer
- Defining Abstract Syntax Tree
- Implementing a parser
- Parsing simple expressions
- Parsing binary expressions
- Invoking a driver for parsing
- Running lexer and parser on our TOY language
- Defining IR code generation methods for each AST class
- Generating IR code for expressions
- Generating IR code for functions
- Adding IR optimization support

Introduction

In this chapter, you will get to know about how to write a frontend for a language. By making use of a custom-defined TOY language, you will have recipes on how to write a lexer and a parser, and how to generate IR code from the **Abstract Syntax Tree** (**AST**) generated by the frontend.

Defining a TOY language

Before implementing a lexer and parser, the syntax and grammar of the language need to be determined first. In this chapter, a TOY language is used to demonstrate how a lexer and a parser can be implemented. The purpose of this recipe is to show how a language is skimmed through. For this purpose, the TOY language to be used is simple but meaningful.

A language typically has some variables, some function calls, some constants, and so on. To keep things simple, our TOY language in consideration has only numeric constants of 32-bit Integer type A, a variable that need not declare its type (like Python, in contrast to C/C++/Java, which require a type declaration) in the TOY language.

How to do it...

The grammar can be defined as follows (the production rules are defined below, with non-terminals on **Left Hand Side** (**LHS**) and a combination of terminals and non-terminals on **Right Hand Side** (**RHS**); when LHS is encountered, it yields appropriate RHS defined in the production rule):

1. A numeric expression will give a constant number:

```
numeric_expr := number
```

2. A parenthesis expression will have an expression in between an opening and a closing bracket:

```
paran_expr := '(' expression ')'
```

3. An identifier expression will either yield an identifier or a function call:

```
identifier_expr
:= identifier
:= identifier '('expr_list ')'
```

4. If identifier `_expr` is a function call, it will either have no arguments or list of arguments separated by a comma:

```
expr_list
:= (empty)
:= expression (',' expression)*
```

5. There will be some primary expression, the starting point of the grammar, which may yield an identifier expression, a numeric expression, or a parenthesis expression:

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

6. An expression can lead to a binary expression:

```
expression := primary binoprhs
```

7. A binary operation with RHS can yield combinations of binary operators and expressions:

```
binoprhs := ( binoperator primary )*
binoperators := '+'/'-'/'*'/'/'
```

8. A function declaration can have grammar as follows:

```
func_decl := identifier '(' identifier_list ')'
identifier_list := (empty)
                        := (identifier)*
```

9. A function definition is distinguished by a `def` keyword followed by a function declaration and an expression that defines its body:

```
function_defn := 'def' func_decl expression
```

10. Finally, there will be a top level expression that will yield an expression:

```
toplevel_expr := expression
```

An example of the TOY language based on the previously defined grammar can be written as follows:

```
def foo (x , y)
x +y * 16
```

Since we have defined the grammar, the next step is to write a lexer and parser for it.

Implementing a lexer

Lexer is a part of the first phase in compiling a program. Lexer tokenizes a stream of input in a program. Then parser consumes these tokens to construct an AST. The language to tokenize is generally a context-free language. A token is a string of one or more characters that are significant as a group. The process of forming tokens from an input stream of characters is called tokenization. Certain delimiters are used to identify groups of words as tokens. There are lexer tools to automate lexical analysis, such as **LEX**. In the TOY lexer demonstrated in the following procedure is a handwritten lexer using C++.

Getting ready

We must have a basic understanding of the TOY language defined in the recipe. Create a file named `toy.cpp` as follows:

```
$ vim toy.cpp
```

All the code that follows will contain all the lexer, parser, and code generation logic.

How to do it...

While implementing a lexer, types of tokens are defined to categorize streams of input strings (similar to states of an automata). This can be done using the **enumeration** (**enum**) type:

1. Open the `toy.cpp` file as follows:

```
$ vim toy.cpp
```

2. Write the `enum` in the `toy.cpp` file as follows:

```
enum Token_Type {
EOF_TOKEN = 0,
NUMERIC_TOKEN,
IDENTIFIER_TOKEN,
PARAN_TOKEN,
DEF_TOKEN
};
```

   Following is the term list for the preceding example:

   - `EOF_TOKEN`: It states the end of file
   - `NUMERIC_TOKEN`: The current token is of numeric type
   - `IDENTIFIER_TOKEN`: The current token is identifier
   - `PARAN_TOKEN`: The current token is parenthesis
   - `DEF_TOKEN`: The current token `def` states that whatever follows is a function definition

3. To hold numeric values, a static variable is defined in the `toy.cpp` file as follows:

```
static int Numeric_Val;
```

4. To hold the `Identifier` string name, a static variable can be defined in the `toy.cpp` file as follows:

```
    static std::string Identifier_string;
```

5. Now the lexer function can be defined by using library functions such as `isspace()`, `isalpha()`, and `fgetc()` in the `toy.cpp` file, as shown in the following:

```
static int get_token() {
  static int LastChar = ' ';

  while(isspace(LastChar))
  LastChar = fgetc(file);

  if(isalpha(LastChar)) {
    Identifier_string = LastChar;
```

```
    while(isalnum((LastChar = fgetc(file))))
    Identifier_string += LastChar;

    if(Identifier_string == "def")
    return DEF_TOKEN;
    return IDENTIFIER_TOKEN;
  }

  if(isdigit(LastChar)) {
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = fgetc(file);
    } while(isdigit(LastChar));

    Numeric_Val = strtod(NumStr.c_str(), 0);
    return NUMERIC_TOKEN;
  }

  if(LastChar == '#') {
    do LastChar = fgetc(file);
    while(LastChar != EOF && LastChar != '\n'
    && LastChar != '\r');

    if(LastChar != EOF) return get_token();
  }

  if(LastChar == EOF) return EOF_TOKEN;

  int ThisChar = LastChar;
  LastChar = fgetc(file);
  return ThisChar;
}
```

How it works...

The example TOY language defined earlier was as follows:

```
def foo (x , y)
x + y * 16
```

The lexer will get the preceding program as input. It will come across the `def` keyword and determine that whatever follows is a definition token, and hence returns the enum value `DEF_TOKEN`. After this, it will come across the function definition and its arguments. Then, there is an expression that involves two binary operators, two variables, and a numeric constant. How these are stored in data structures is demonstrated in the following recipes.

See also

- See more sophisticated and detailed handwritten lexer for the C++ language is written in Clang, at `http://clang.llvm.org/doxygen/Lexer_8cpp_source.html`

Defining Abstract Syntax Tree

AST is a tree representation of the abstract syntactic structure of the source code of a programming language. The ASTs of programming constructs, such as expressions, flow control statements, and so on, are grouped into operators and operands. ASTs represent relationships between programming constructs, and not the ways they are generated by grammar. ASTs ignore unimportant programming elements such as punctuations and delimiters. ASTs generally contain additional properties of every element in it, which are useful in further compilation phases. Location of source code is one such property, which can be used to throw an error line number if an error is encountered in determining the correctness of the source code in accordance with the grammar (location, line number, column number, and so on, and other related properties are stored in an object of the `SourceManager` class in Clang frontend for C++).

The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree allows verification of the correctness of the program. After verifying correctness, the AST serves as the base for code generation.

Getting ready

We must have run the lexer by now to obtain the tokens that will be used in generating the AST. The languages we intend to parse consist of expressions, function definitions, and function declarations. Again we have various types of expressions—variables, binary operators, numeric expressions, and so on.

How to do it...

To define AST structure, proceed with the following steps:

1. Open the `toy.cpp` file as follows:

   ```
   $ vi toy.cpp
   ```

   Below the lexer code, define ASTs.

2. A `base` class is defined for parsing an expression as follows:

```
class BaseAST {
  public :
  virtual ~BaseAST();
};
```

Then, several derived classes are defined for every type of expression to be parsed.

3. An AST class for variable expressions is defined as follows:

```
class VariableAST  : public BaseAST{
  std::string Var_Name;
// string object to store name of
// the variable.
  public:
  VariableAST (std::string &name) : Var_Name(name) {}  // ..//
parameterized constructor of variable AST class to be initialized
with the string passed to the constructor.
};
```

4. The language has some numeric expressions. The `AST` class for such numeric expressions can be defined as follows:

```
class NumericAST : public BaseAST {
  int numeric_val;
  public :
  NumericAST (intval) :numeric_val(val)  {}
};
```

5. For expressions involving binary operation, the `AST` class can be defined as follows:

```
Class BinaryAST : public BaseAST {
  std::string Bin_Operator;  // string object to store
  // binary operator
  BaseAST  *LHS, *RHS;  // Objects used to store LHS and
// RHS of a binary Expression. The LHS and RHS binary
// operation can be of any type, hence a BaseAST object
// is used to store them.
  public:
  BinaryAST (std::string op, BaseAST *lhs, BaseAST *rhs ) :
  Bin_Operator(op), LHS(lhs), RHS(rhs) {}  // Constructor
  //to initialize binary operator, lhs and rhs of the binary
  //expression.
};
```

6. The `AST` class for function declaration can be defined as follows:

```
class FunctionDeclAST {
  std::string Func_Name;
  std::vector<std::string> Arguments;
  public:
  FunctionDeclAST(const std::string &name, const
std::vector<std::string> &args) :
  Func_Name(name), Arguments(args) {};
};
```

7. The `AST` class for function definition can be defined as follows:

```
class FunctionDefnAST {
  FunctionDeclAST *Func_Decl;
  BaseAST* Body;
  public:
  FunctionDefnAST(FunctionDeclAST *proto, BaseAST *body) :
  Func_Decl(proto), Body(body) {}
};
```

8. The `AST` class for function call can be defined as follows:

```
class FunctionCallAST : public BaseAST {
  std::string Function_Callee;
  std::vector<BaseAST*> Function_Arguments;
  public:
  FunctionCallAST(const std::string &callee, std::vector<BaseAST*>
&args) :
  Function_Callee(callee), Function_Arguments(args) {}
};
```

The basic skeleton of the AST is now ready to use.

How it works...

The AST acts as a data structure for storing various information about the tokens given by the lexer. This information is generated in the parser logic and ASTs are filled up according to the type of token being parsed.

See also

- Having generated the AST, we will implement the parser, and only after that will we see an example where both lexer and parser will be invoked. For a more detailed AST structure of C++ in Clang, refer to: `http://clang.llvm.org/docs/IntroductionToTheClangAST.html`.

Implementing a parser

Parser analyzes a code syntactically according to the rules of the language's grammar. The parsing phase determines if the input code can be used to form a string of tokens according to the defined grammar. A parse tree is constructed in this phase. Parser defines functions to organize language into a data structure called AST. The parser defined in this recipe uses a recursive decent parser technique which is a top-down parser, and uses mutually recursive functions to build the AST.

Getting ready

We must have the custom-defined language, that is the TOY language in this case, and also a stream of tokens generated by the lexer.

How to do it...

Define some basic value holders in our TOY parser as shown in the following:

1. Open the `toy.cpp` file as follows:

```
$ vi toy.cpp
```

2. Define a global static variable to hold the current token from the lexer as follows:

```
static int Current_token;
```

3. Define a function to get the next token from the input stream from the lexer as follows:

```
static void next_token() {
  Current_token =  get_token();
}
```

4. The next step is to define functions for expression parsing by using the AST data structure defined in the previous section.
5. Define a generic function to call specific parsing functions according to the types of tokens determined by the lexer, as shown in the following:

```
static BaseAST* Base_Parser() {
  switch (Current_token) {
    default: return 0;
    case IDENTIFIER_TOKEN : return identifier_parser();
    case NUMERIC_TOKEN : return numeric_parser();
    case '(' : return paran_parser();
  }
}
```

How it works...

The stream of input is tokenized and fed to the parser. `Current_token` holds the token to be processed. The type of token is known at this stage and the corresponding parser functions are called to initialize ASTs.

See also

- In next few recipes, you will learn how to parse different expressions. For more detailed parsing of the C++ language implemented in Clang, refer to it works: `http://clang.llvm.org/doxygen/classclang_1_1Parser.html`.

Parsing simple expressions

In this recipe, you will learn how to parse a simple expression. A simple expression may consist of numeric values, identifiers, function calls, a function declaration, and function definitions. For each type of expression, individual parser logic needs to be defined.

Getting ready

We must have the custom-defined language—that is, the TOY language in this case—and also stream of tokens generated by lexer. We already defined ASTs above. Further, we are going to parse the expression and invoke AST constructors for every type of expression.

How to do it...

To parse simple expressions, proceed with the following code flow:

1. Open the `toy.cpp` file as follows:

   ```
   $ vi toy.cpp
   ```

   We already have lexer logic present in the `toy.cpp` file. Whatever code follows needs to be appended after the lexer code in the `toy.cpp` file.

2. Define the `parser` function for numeric expression as follows:

   ```
   static BaseAST *numeric_parser() {
     BaseAST *Result = new NumericAST(Numeric_Val);
     next_token();
     return Result;
   }
   ```

3. Define the `parser` function for an identifier expression. Note that identifier can be a variable reference or a function call. They are distinguished by checking if the next token is `(`. This is implemented as follows:

```
static BaseAST* identifier_parser() {
  std::string IdName = Identifier_string;

  next_token();

  if(Current_token != '(')
  return new VariableAST(IdName);

  next_token();

  std::vector<BaseAST*> Args;
  if(Current_token != ')') {
    while(1) {
      BaseAST* Arg = expression_parser();
      if(!Arg) return 0;
      Args.push_back(Arg);

      if(Current_token == ')') break;

      if(Current_token != ',')
      return 0;
      next_token();
    }
  }
  next_token();

  return new FunctionCallAST(IdName, Args);
}
```

4. Define the `parser` function for the function declaration as follows:

```
static FunctionDeclAST *func_decl_parser() {
  if(Current_token != IDENTIFIER_TOKEN)
  return 0;

  std::string FnName = Identifier_string;
  next_token();

  if(Current_token != '(')
  return 0;

  std::vector<std::string> Function_Argument_Names;
```

```
  while(next_token() == IDENTIFIER_TOKEN)
  Function_Argument_Names.push_back(Identifier_string);
  if(Current_token != ')')
  return 0;

  next_token();

  return new FunctionDeclAST(FnName, Function_Argument_Names);
}
```

5. Define the `parser` function for the function definition as follows:

```
static FunctionDefnAST *func_defn_parser() {
  next_token();
  FunctionDeclAST *Decl = func_decl_parser();
  if(Decl == 0) return 0;

  if(BaseAST* Body = expression_parser())
  return new FunctionDefnAST(Decl, Body);
  return 0;
}
```

Note that the function called `expression_parser` used in the preceding code, parses the expression. The function can be defined as follows:

```
static BaseAST* expression_parser() {
  BaseAST *LHS = Base_Parser();
  if(!LHS) return 0;
  return binary_op_parser(0, LHS);
}
```

How it works...

If a numeric token is encountered, the constructor for the numeric expression is invoked and the AST object for the numeric value is returned by the parser, filling up the AST for numeric values with the numeric data.

Similarly, for identifier expressions, the parsed data will either be a variable or a function call. For function declaration and definitions, the name of the function and function arguments is parsed and the corresponding AST class constructors are invoked.

Parsing binary expressions

In this recipe, you will learn how to parse a binary expression.

Getting ready

We must have the custom-defined language—that is, the toy language in this case—and also stream of tokens generated by lexer. The binary expression parser requires precedence of binary operators for determining LHS and RHS in order. An STL map can be used to define precedence of binary operators.

How to do it...

To parse a binary expression, proceed with the following code flow:

1. Open the `toy.cpp` file as follows:

```
$ vi toy.cpp
```

2. Declare a `map` for operator precedence to store the precedence at global scope in the `toy.cpp` file as follows:

```
static std::map<char, int>Operator_Precedence;
```

   The TOY language for demonstration has 4 operators where precedence of operators is defined as `-` < `+` < `/` < `*`.

3. A function to initialize precedence—that is, to store precedence value in `map`—can be defined in global scope in the `toy.cpp` file as follows:

```
static void init_precedence() {
  Operator_Precedence['-'] = 1;
  Operator_Precedence['+'] = 2;
  Operator_Precedence['/'] = 3;
  Operator_Precedence['*'] = 4;
}
```

4. A helper function to return precedence of binary operator can be defined as follows:

```
static int getBinOpPrecedence() {
  if(!isascii(Current_token))
return -1;

  int TokPrec = Operator_Precedence[Current_token];
  if(TokPrec <= 0) return -1;
  return TokPrec;
}
```

5. Now, the `binary` operator parser can be defined as follows:

```
static BaseAST* binary_op_parser(int Old_Prec, BaseAST *LHS) {
  while(1) {
    int Operator_Prec = getBinOpPrecedence();
```

```
    if(Operator_Prec < Old_Prec)
    return LHS;

    int BinOp = Current_token;
    next_token();

    BaseAST* RHS = Base_Parser();
    if(!RHS) return 0;

    int Next_Prec = getBinOpPrecedence();
    if(Operator_Prec < Next_Prec) {
      RHS = binary_op_parser(Operator_Prec+1, RHS);
      if(RHS == 0) return 0;
    }

    LHS = new BinaryAST(std::to_string(BinOp), LHS, RHS);
  }
}
```

Here, precedence of current operator is checked with the precedence of old operator, and the outcome is decided according to LHS and RHS of binary operators. Note that the binary operator parser is recursively called since the RHS can be an expression and not just a single identifier.

6. A `parser` function for parenthesis can be defined as follows:

```
static BaseAST* paran_parser() {
  next_token();
  BaseAST* V = expression_parser();
  if (!V) return 0;

  if(Current_token != ')')
    return 0;
  return V;
}
```

7. Some top-level functions acting as wrappers around these `parser` functions can be defined as follows:

```
static void HandleDefn() {
  if (FunctionDefnAST *F = func_defn_parser()) {
    if(Function* LF = F->Codegen()) {
  }
  }
  else {
    next_token();
```

```
  }
}

static void HandleTopExpression() {
  if(FunctionDefnAST *F = top_level_parser()) {
    if(Function *LF = F->Codegen()) {
  }
  }
  else {
    next_token();
  }
}
```

See also

- All of the remaining recipes in this chapter pertain to user objects. For detailed parsing of expressions, and for C++ parsing, please refer to: `http://clang.llvm.org/doxygen/classclang_1_1Parser.html`.

Invoking a driver for parsing

In this recipe, you will learn how to call the parser function from the main function of our TOY parser.

How to do it...

To invoke a driver program to start parsing, define the driver function as shown in the following:

1. Open the `toy.cpp` file:

```
$ vi toy.cpp
```

2. A `Driver` function called from the main function, and a parser can now be defined as follows:

```
static void Driver() {
  while(1) {
    switch(Current_token) {
    case EOF_TOKEN : return;
    case ';' : next_token(); break;
    case DEF_TOKEN : HandleDefn(); break;
    default : HandleTopExpression(); break;
  }
  }
}
```

3. The `main()` function to run the whole program can be defined as follows:

```
int main(int argc, char* argv[]) {
  LLVMContext &Context = getGlobalContext();
  init_precedence();
  file = fopen(argv[1], "r");
  if(file == 0) {
    printf("Could not open file\n");
  }
  next_token();
  Module_Ob = new Module("my compiler", Context);
  Driver();
  Module_Ob->dump();
      return 0;
}
```

How it works...

The main function is responsible for calling the lexer and parser so that both can act over a piece of code that is being input to the compiler frontend. From the main function, driver function is invoked to start the process of parsing.

See also

- For details on how the main function and driver function work for c++ parsing in Clang, refer to `http://llvm.org/viewvc/llvm-project/cfe/trunk/tools/driver/cc1_main.cpp`

Running lexer and parser on our TOY language

Now that a full-fledged lexer and parser for our TOY language grammar are defined, it's time to run it on example TOY language.

Getting ready

To do this, you should have understanding of TOY language grammar and all the previous recipes of this chapter.

How to do it...

Run and test the Lexer and Parser on TOY Language, as shown in the following:

1. First step is to compile the `toy.cpp` program into an executable:

```
$ clang++ toy.cpp  -O3 -o toy
```

2. The `toy` executable is our TOY compiler frontend. The `toy` language to be parsed is in a file called `example`:

```
$ cat example
def foo(x , y)
x + y * 16
```

3. This file is passed as argument to be processed by the `toy` compiler:

```
$ ./toy example
```

How it works...

The TOY compiler will open the `example` file in read mode. Then, it will tokenize the stream of words. It will come across the def keyword and return `DEF_TOKEN`. Then, the `HandleDefn()` function will be called, which will store the function name and the argument. It will recursively check for the type of token and then call the specific token handler functions to store them into respective ASTs.

See also

- The aforementioned lexer and parser do not handle errors in syntax except a few trivial ones. To implement Error handling, refer to `http://llvm.org/docs/tutorial/LangImpl2.html#parser-basics`.

Defining IR code generation methods for each AST class

Now, since the AST is ready with all the necessary information in its data structure, the next phase is to generate LLVM IR. LLVM APIs are used in this code generation. LLVM IR has a predefined format that is generated by the inbuilt APIs of LLVM.

Getting ready

You must have created the AST from any input code of the TOY language.

How to do it...

In order to generate LLVM IR, a virtual `CodeGen` function is defined in each AST class (the AST classes were defined earlier in the AST section; these functions are additional to those classes) as follows:

1. Open the `toy.cpp` file as follows:

```
$ vi toy.cpp
```

2. In the `BaseAST` class defined earlier, append the `Codegen()` functions as follows:

```
class BaseAST {
  …
  …
  virtual Value* Codegen() = 0;
};
class NumericAST : public BaseAST {
  …
  …
  virtual Value* Codegen();
};
class VariableAST : public BaseAST {
  …
  …
  virtual Value* Codegen();
};
```

   This virtual `Codegen()` function is included in every AST class we defined.

   This function returns an LLVM Value object, which represents **Static Single Assignment** (**SSA**) value in LLVM. A few more static variables are defined that will be used during Codegen.

3. Declare the following static variables in global scope as follows:

```
static Module *Module_Ob;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*>Named_Values;
```

How it works...

The `Module_Ob` module contains all the functions and variables in the code.

The `Builder` object helps to generate LLVM IR and keeps track of the current point in the program to insert LLVM instructions. The `Builder` object has functions to create new instructions.

The `Named_Values` map keeps track of all the values defined in the current scope like a symbol table. For our language, this map will contain function parameters.

Generating IR code for expressions

In this recipe, you will see how IR code gets generated for an expression using the compiler frontend.

How to do it...

To implement LLVM IR code generation for our TOY language, proceed with the following code flow:

1. Open the `toy.cpp` file as follows:

```
$ vi toy.cpp
```

2. The function to generate code for numeric values can be defined as follows:

```
Value *NumericAST::Codegen() {
  return ConstantInt::get(Type::getInt32Ty(getGlobalContext()),
numeric_val);
}
```

   In LLVM IR, integer constants are represented by the `ConstantInt` class whose numeric value is held by the `APInt` class.

3. The function for generating code for variable expressions can be defined as follows:

```
Value *VariableAST::Codegen() {
  Value *V = Named_Values[Var_Name];
  return V ? V : 0;
}
```

4. The `Codegen()` function for binary expression can be defined as follows:

```
Value *BinaryAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if(L == 0 || R == 0) return 0;

  switch(atoi(Bin_Operator.c_str())) {
    case '+' : return Builder.CreateAdd(L, R, "addtmp");
    case '-' : return Builder.CreateSub(L, R, "subtmp");
    case '*' : return Builder.CreateMul(L, R, "multmp");
    case '/' : return Builder.CreateUDiv(L, R, "divtmp");
    default : return 0;
  }
}
```

   If the code above emits multiple `addtmp` variables, LLVM will automatically provide each one with an increasing, unique numeric suffix.

See also

- The next recipe shows how to generate IR code for function; we will learn how the code generation actually works.

Generating IR code for functions

In this recipe you, will learn how to generate IR code for a function.

How to do it...

Do the following steps:

1. The `Codegen()` function for the function call can be defined as follows:

```
Value *FunctionCallAST::Codegen() {
  Function *CalleeF =
  Module_Ob->getFunction(Function_Callee);
  std::vector<Value*>ArgsV;
  for(unsigned i = 0, e = Function_Arguments.size();
  i != e; ++i) {
    ArgsV.push_back(Function_Arguments[i]->Codegen());
    if(ArgsV.back() == 0) return 0;
  }
  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

   Once we have the function to call, we recursively call the `Codegen()` function for each argument that is to be passed in and create an LLVM call instruction.

2. Now that the `Codegen()` function for a function call has been defined, it's time to define the `Codegen()` functions for declarations and function definitions.

   The `Codegen()` function for function declarations can be defined as follows:

```
Function *FunctionDeclAST::Codegen() {
  std::vector<Type*>Integers(Arguments.size(), Type::getInt32Ty(ge
tGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getInt32Ty(getGlobalC
ontext()), Integers, false);
  Function *F = Function::Create(FT,  Function::ExternalLinkage,
Func_Name, Module_Ob);

  if(F->getName() != Func_Name) {
    F->eraseFromParent();
    F = Module_Ob->getFunction(Func_Name);

    if(!F->empty()) return 0;
```

```
    if(F->arg_size() != Arguments.size()) return 0;

  }

  unsigned Idx = 0;
  for(Function::arg_iterator Arg_It = F->arg_begin(); Idx !=
Arguments.size(); ++Arg_It, ++Idx) {
    Arg_It->setName(Arguments[Idx]);
    Named_Values[Arguments[Idx]] = Arg_It;
  }

  return F;
}
```

The `Codegen()` function for function definition can be defined as follows:

```
Function *FunctionDefnAST::Codegen() {
  Named_Values.clear();

  Function *TheFunction = Func_Decl->Codegen();
  if(TheFunction == 0) return 0;

  BasicBlock *BB = BasicBlock::Create(getGlobalContext(),"entry",
TheFunction);
  Builder.SetInsertPoint(BB);

  if(Value *RetVal = Body->Codegen()) {
    Builder.CreateRet(RetVal);
    verifyFunction(*TheFunction);
    return TheFunction;
  }

  TheFunction->eraseFromParent();
  return 0;
}
```

3. That's it! LLVMIR is now ready. These `Codegen()` functions can be called in the wrappers written to parse top-level expressions as follows:

```
static void HandleDefn() {
  if (FunctionDefnAST *F = func_defn_parser()) {
    if(Function* LF = F->Codegen()) {
    }
  }
  else {
```

```
    next_token();
  }
}
static void HandleTopExpression() {
  if(FunctionDefnAST *F = top_level_parser()) {
    if(Function *LF = F->Codegen()) {
    }
  }
  else {
    next_token();
  }
}
```

So, after parsing successfully, the respective `Codegen()` functions are called to generate the LLVM IR. The `dump()` function is called to print the generated IR.

How it works...

The `Codegen()` functions use LLVM inbuilt function calls to generate IR. The header files to include for this purpose are `llvm/IR/Verifier.h`, `llvm/IR/DerivedTypes.h`, `llvm/IR/IRBuilder.h`, and `llvm/IR/LLVMContext.h`, `llvm/IR/Module.h`.

1. While compiling, this code needs to be linked with LLVM libraries. For this purpose, the `llvm-config` tool can be used as follows:

   ```
   llvm-config  --cxxflags  --ldflags  --system-libs  --libs core.
   ```

2. For this purpose, the `toy` program is recompiled with additional flags as follows:

   ```
   $ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --
   system-libs --libs core` -o toy
   ```

3. When the `toy` compiler is now run on `example` code, it will generate LLVM IR as follows:

   ```
   $ ./toy example

   define i32 @foo (i32 %x, i32 %y) {
     entry:
     %multmp = muli32 %y, 16
     %addtmp = add i32 %x, %multmp
     reti32 %addtmp
   }
   ```

   Another `example2` file has a function `call.$ cat example2`:

   ```
   foo(5, 6);
   ```

Its LLVM IR will be dumped as follows:

```
$ ./toy example2
define i32 @1 () {
  entry:
  %calltmp = call i32@foo(i32 5, i32 6)
  reti32 %calltmp
}
```

See also

- For details on how `Codegen()` functions for C++ in Clang, refer to `http://llvm.org/viewvc/llvm-project/cfe/trunk/lib/CodeGen/`

Adding IR optimization support

LLVM provides a wide variety of optimization passes. LLVM allows a compiler implementation to decide which optimizations to use, their order, and so on. In this recipe, you will learn how to add IR optimization support.

How to do it...

Do the following steps:

1. To start with the addition of IR optimization support, first of all a static variable for function manager has to be defined as follows:

   ```
   static FunctionPassManager *Global_FP;
   ```

2. Then, a function pass manager needs to be defined for the `Module` object used previously. This can be done in the `main()` function as follows:

   ```
   FunctionPassManager My_FP(TheModule);
   ```

3. Now a pipeline of various optimizer passes can be added in the `main()` function as follows:

   ```
   My_FP.add(createBasicAliasAnalysisPass());
   My_FP.add(createInstructionCombiningPass());
   My_FP.add(createReassociatePass());
   My_FP.add(createGVNPass());
   My_FP.doInitialization();
   ```

4. Now the static global function Pass Manager is assigned to this pipeline as follows:

```
Global_FP = &My_FP;
Driver();
```

   This PassManager has a run method, which we can run on the function IR generated before returning from `Codegen()` of the function definition. This is demonstrated as follows:

```
Function* FunctionDefnAST::Codegen() {
  Named_Values.clear();
  Function *TheFunction = Func_Decl->Codegen();
  if (!TheFunction) return 0;
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry",
TheFunction);
  Builder.SetInsertPoint(BB);
  if (Value* Return_Value = Body->Codegen()) {
    Builder.CreateRet(Return_Value);
    verifyFunction(*TheFunction);
    Global_FP->run(*TheFunction);
    returnTheFunction;
  }
  TheFunction->eraseFromParent();
  return 0;
}
```

This is a lot more beneficial as it optimizes the function in place, improving the code generated for the function body.

See also

- How to add our own optimization pass and its run method will be demonstrated in the later chapters

3

Extending the Frontend and Adding JIT Support

In this chapter, we will cover the following recipes:

- Handling decision making paradigms – if/then/else constructs
- Generating code for loops
- Handling user-defined operators – binary operators
- Handling user-defined operators – unary operators
- Adding JIT support

Introduction

In the last chapter, the basics of the frontend component for a language were defined. This included defining tokens for different types of expressions, writing a lexer to tokenize a stream of input, chalking out a skeleton for the abstract syntax tree of various expressions, writing a parser, and generating code for the language. Also, how various optimizations can be hooked to the frontend was explained.

A language is more powerful and expressive when it has control flow and loops to decide the flow of a program. JIT support explores the possibility of compiling code on-the-fly. In this chapter, implementation of these more sophisticated programming paradigms will be discussed. This chapter deals with enhancements of a programming language that make it more meaningful and powerful to use. The recipes in this chapter demonstrate how to include those enhancements for a given language.

Handling decision making paradigms – if/then/else constructs

In any programming language, executing a statement based on certain conditions gives a very powerful advantage to the language. The `if/then/else` constructs provide the capability to alter the control flow of a program, based on certain conditions. The condition is present in an `if` construct. If the condition is true, the expression following the `then` construct is executed. If it is `false`, the expression following the `else` construct is executed. This recipe demonstrates a basic infrastructure to parse and generate code for the `if/then/else` construct.

Getting ready

The TOY language for `if/then/else` can be defined as:

```
if x < 2 then
x + y
else
x - y
```

For checking a condition, a comparison operator is required. A simple *less than* operator, `<`, will serve the purpose. To handle `<`, precedence needs to be defined in the `init_precedence()` function, as shown here:

```
static void init_precedence() {
  Operator_Precedence['<'] = 0;
  …
  …
}
```

Also, the `codegen()` function for binary expressions needs to be included for `<`:

```
Value* BinaryAST::Codegen() {
…
…
…
case '<' :
L = Builder.CreateICmpULT(L, R, "cmptmp");
return Builder.CreateZExt(L, Type::getInt32Ty(getGlobalContext()),
                                "booltmp");…
…
}
```

Now, the LLVM IR will generate a comparison instruction and a Boolean instruction as a result of the comparison, which will be used to determine where the control of the program will flow. It's time to handle the `if/then/else` paradigm now.

How to do it...

Do the following steps:

1. The lexer in the `toy.cpp` file has to be extended to handle the `if/then/else` constructs. This can be done by appending a token for this in the `enum` of tokens:

```
enum Token_Type{
…
…
IF_TOKEN,
THEN_TOKEN,
ELSE_TOKEN
}
```

2. The next step is to append the entries for these tokens in the `get_token()` function, where we match strings and return the appropriate tokens:

```
static int get_token() {
…
…
…
if (Identifier_string == "def")  return DEF_TOKEN;
if(Identifier_string == "if") return IF_TOKEN;
if(Identifier_string == "then") return THEN_TOKEN;
if(Identifier_string == "else") return ELSE_TOKEN;
…
…
}
```

3. Then we define an AST node in the `toy.cpp` file:

```
class ExprIfAST : public BaseAST {
  BaseAST *Cond, *Then, *Else;

public:
  ExprIfAST(BaseAST *cond, BaseAST *then, BaseAST * else_st)
      : Cond(cond), Then(then), Else(else_st) {}
  Value *Codegen() override;
};
```

4. The next step is to define the parsing logic for the `if/then/else` constructs:

```
static BaseAST *If_parser() {
  next_token();

  BaseAST *Cond = expression_parser();
  if (!Cond)
```

```
    return 0;

  if (Current_token != THEN_TOKEN)
    return 0;
  next_token();

  BaseAST *Then = expression_parser();
  if (Then == 0)
    return 0;

  if (Current_token != ELSE_TOKEN)
    return 0;

  next_token();

  BaseAST *Else = expression_parser();
  if (!Else)
    return 0;

  return new ExprIfAST(Cond, Then, Else);
}
```

The parser logic is simple: first, the `if` token is searched for and the expression following it is parsed for the condition. After that, the `then` token is identified and the true condition expression is parsed. Then the `else` token is searched for and the false condition expression is parsed.

5. Next we hook up the previously defined function with `Base_Parser()`:

```
static BaseAST* Base_Parser() {
switch(Current_token) {
…
…
…
case IF_TOKEN : return If_parser();
…
}
```

6. Now that the AST of `if/then/else` is filled with the expression by the parser, it's time to generate the LLVM IR for the conditional paradigm. Let's define the `Codegen()` function:

```
Value *ExprIfAST::Codegen() {
  Value *Condtn = Cond->Codegen();
  if (Condtn == 0)
    return 0;
```

```
  Condtn = Builder.CreateICmpNE(
      Condtn, Builder.getInt32(0), "ifcond");

  Function *TheFunc = Builder.GetInsertBlock()->getParent();

  BasicBlock *ThenBB =
      BasicBlock::Create(getGlobalContext(), "then", TheFunc);
  BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(),
"else");
  BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(),
"ifcont");

  Builder.CreateCondBr(Condtn, ThenBB, ElseBB);

  Builder.SetInsertPoint(ThenBB);

  Value *ThenVal = Then->Codegen();
  if (ThenVal == 0)
    return 0;

  Builder.CreateBr(MergeBB);
  ThenBB = Builder.GetInsertBlock();

  TheFunc->getBasicBlockList().push_back(ElseBB);
  Builder.SetInsertPoint(ElseBB);

  Value *ElseVal = Else->Codegen();
  if (ElseVal == 0)
    return 0;

  Builder.CreateBr(MergeBB);
  ElseBB = Builder.GetInsertBlock();

  TheFunc->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *Phi = Builder.CreatePHI(Type::getInt32Ty(getGlobalConte
xt()), 2, "iftmp");

  Phi->addIncoming(ThenVal, ThenBB);
  Phi->addIncoming(ElseVal, ElseBB);
  return Phi;
}
```

Now that we are ready with the code, let's compile and run it on a sample program containing the `if/then/else` constructs.

How it works...

Do the following steps:

1. Compile the `toy.cpp` file:

```
$ g++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-
libs --libs core ` -O3 -o toy
```

2. Open an example file:

```
$ vi example
```

3. Write the following `if/then/else` code in the example file:

```
def fib(x)
  if x < 3 then
    1
  Else
    fib(x-1)+fib(x-2);
```

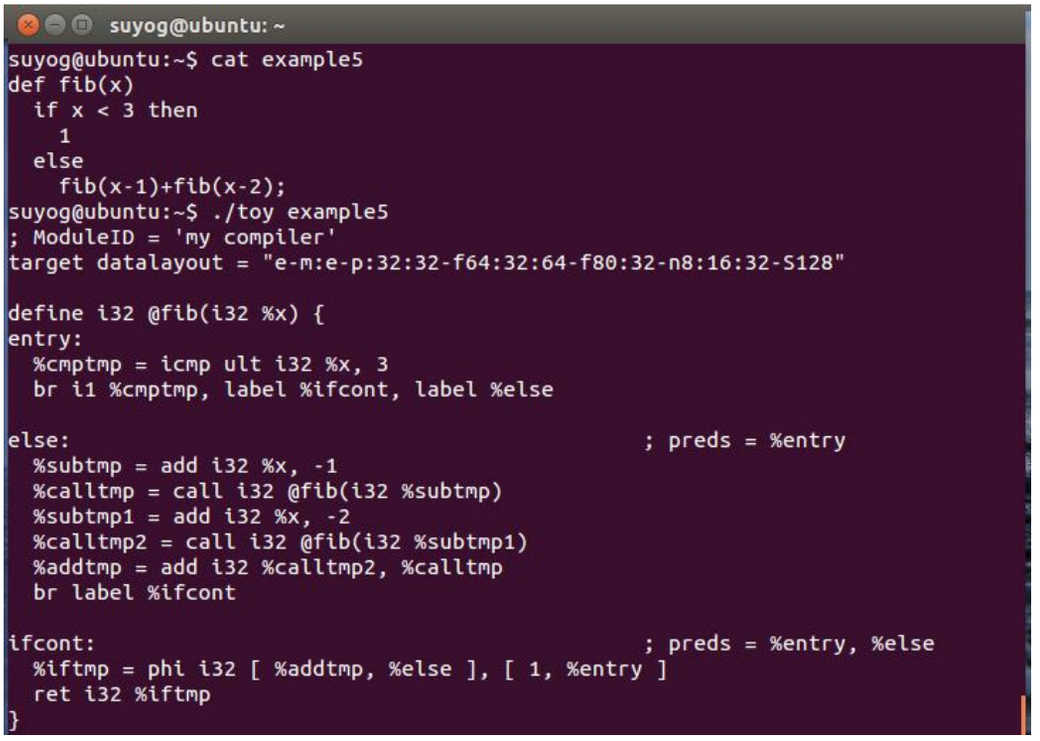
4. Compile the example file with the TOY compiler:

```
$ ./toy example
```

The LLVM IR generated for the `if/then/else` code will look like this:

```
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @fib(i32 %x) {
entry:
  %cmptmp = icmp ult i32 %x, 3
  br i1 %cmptmp, label %ifcont, label %else

else:                                             ; preds = %entry
  %subtmp = add i32 %x, -1
  %calltmp = call i32 @fib(i32 %subtmp)
  %subtmp1 = add i32 %x, -2
  %calltmp2 = call i32 @fib(i32 %subtmp1)
  %addtmp = add i32 %calltmp2, %calltmp
  br label %ifcont

ifcont:                                           ; preds = %entry,
%else
  %iftmp = phi i32 [ %addtmp, %else ], [ 1, %entry ]
  ret i32 %iftmp
}
```

Here's what the output looks like:



The parser identifies the `if/then/else` constructs and the statements that are to be executed in true and false conditions, and stores them in the AST. The code generator then converts the AST into LLVM IR, where the condition statement is generated. IR is generated for true as well as false conditions. Depending on the state of the condition variable, the appropriate statement is executed at runtime.

See also

- For a detailed example on how an `if else` statement is handled in C++ by Clang, refer to `http://clang.llvm.org/doxygen/classclang_1_1IfStmt.html`.

Generating code for loops

Loops make a language powerful enough to perform the same operation several times, with limited lines of code. Loops are present in almost every language. This recipe demonstrates how loops are handled in the TOY language.

Getting ready

A loop typically has a start that initializes the induction variable, a step that indicates an increment or decrement in the induction variable, and an end condition for termination of the loop. The loop in our TOY language can be defined as follows:

```
for i = 1, i < n, 1 in
     x + y;
```

The start expression is the initialization of `i = 1`. The end condition for the loop is `i<n`. The first line of the code indicates `i` be incremented by `1`.

As long as the end condition is true, the loop iterates and, after each iteration, the induction variable, `i`, is incremented by 1. An interesting thing called **PHI** node comes into the picture to decide which value the induction variable, `i`, will take. Remember that our IR is in the **single static assignment** (**SSA**) form. In a control flow graph, for a given variable, the values can come from two different blocks. To represent SSA in LLVM IR, the `phi` instruction is defined. Here is an example of `phi`:

```
%i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
```

The preceding IR indicates that the value for `i` can come from two basic blocks: `%entry` and `%loop`. The value from the `%entry` block will be `1`, while the `%nextvar` variable will be from `%loop`. We will see the details after implementing the loop for our toy compiler.

How to do it...

Like any other expression, loops are also handled by including states in lexer, defining the AST data structure to hold loop values, and defining the parser and the `Codegen()` function to generate the LLVM IR:

1. The first step is to define tokens in the lexer in `toy.cpp` file:

```
enum Token_Type {
  …
  …
  FOR_TOKEN,
  IN_TOKEN
  …
  …
};
```

2. Then we include the logic in the lexer:

```
static int get_token() {
  …
  …
if (Identifier_string == "else")
      return ELSE_TOKEN;
    if (Identifier_string == "for")
      return FOR_TOKEN;
    if (Identifier_string == "in")
      return IN_TOKEN;
  …
  …
}
```

3. The next step is to define the AST for the `for` loop:

```
class ExprForAST  : public BaseAST {
  std::string Var_Name;
  BaseAST *Start, *End, *Step, *Body;

public:
  ExprForAST (const std::string &varname, BaseAST *start, BaseAST
*end,
              BaseAST *step, BaseAST *body)
      : Var_Name(varname), Start(start), End(end), Step(step),
Body(body) {}
  Value *Codegen() override;
};
```

4. Then we define the parser logic for the loop:

```
static BaseAST *For_parser() {
  next_token();

  if (Current_token != IDENTIFIER_TOKEN)
    return 0;

  std::string IdName = Identifier_string;
  next_token();

  if (Current_token != '=')
    return 0;
  next_token();

  BaseAST *Start = expression_parser();
  if (Start == 0)
```

```
    return 0;
  if (Current_token != ',')
    return 0;
  next_token();

  BaseAST *End = expression_parser();
  if (End == 0)
    return 0;

  BaseAST *Step = 0;
  if (Current_token == ',') {
    next_token();
    Step = expression_parser();
    if (Step == 0)
      return 0;
  }

  if (Current_token != IN_TOKEN)
    return 0;
  next_token();

  BaseAST *Body = expression_parser();
  if (Body == 0)
    return 0;

  return new ExprForAST (IdName, Start, End, Step, Body);
}
```

5. Next we define the `Codegen()` function to generate the LLVM IR:

```
Value *ExprForAST::Codegen() {

  Value *StartVal = Start->Codegen();
  if (StartVal == 0)
    return 0;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder.GetInsertBlock();
  BasicBlock *LoopBB =
      BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

  Builder.CreateBr(LoopBB);

  Builder.SetInsertPoint(LoopBB);
```

```
  PHINode *Variable = Builder.CreatePHI(Type::getInt32Ty(getGlobal
Context()), 2, Var_Name.c_str());
  Variable->addIncoming(StartVal, PreheaderBB);

  Value *OldVal = Named_Values[Var_Name];
  Named_Values[Var_Name] = Variable;

  if (Body->Codegen() == 0)
    return 0;

  Value *StepVal;
  if (Step) {
    StepVal = Step->Codegen();
    if (StepVal == 0)
      return 0;
  } else {
    StepVal = ConstantInt::get(Type::getInt32Ty(getGlobalConte
xt()), 1);
  }

  Value *NextVar = Builder.CreateAdd(Variable, StepVal,
"nextvar");

  Value *EndCond = End->Codegen();
  if (EndCond == 0)
    return EndCond;

  EndCond = Builder.CreateICmpNE(
      EndCond, ConstantInt::get(Type::getInt32Ty(getGlobalConte
xt()), 0), "loopcond");

  BasicBlock *LoopEndBB = Builder.GetInsertBlock();
  BasicBlock *AfterBB =
      BasicBlock::Create(getGlobalContext(), "afterloop",
TheFunction);

  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  Builder.SetInsertPoint(AfterBB);

  Variable->addIncoming(NextVar, LoopEndBB);

  if (OldVal)
    Named_Values[Var_Name] = OldVal;
```

```
  else
    Named_Values.erase(Var_Name);

  return Constant::getNullValue(Type::getInt32Ty(getGlobalConte
xt()));
}
```

How it works...

Do the following steps:

1. Compile the `toy.cpp` file:

   ```
   $ g++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-
   libs --libs core ` -O3 -o toy
   ```

2. Open an example file:

   ```
   $ vi example
   ```

3. Write the following code for a `for` loop in the example file:

   ```
   def printstar(n x)
     for i = 1, i < n, 1.0 in
       x + 1
   ```

4. Compile the example file with the TOY compiler:

   ```
   $ ./toy example
   ```

5. The LLVM IR for the preceding `for` loop code will be generated, as follows:

   ```
   ; ModuleID = 'my compiler'
   target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

   define i32 @printstar(i32 %n, i32 %x) {
   entry:
     br label %loop

   loop:                                             ; preds = %loop,
   %entry
     %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
     %nextvar = add i32 %i, 1
     %cmptmp = icmp ult i32 %i, %n
     br i1 %cmptmp, label %loop, label %afterloop

   afterloop:                                        ; preds = %loop
     ret i32 0
   }
   ```

The parser you just saw identifies the loop, initialization of the induction variable, the termination condition, the step value for the induction variable, and the body of the loop. It then converts each of the blocks in LLVM IR, as seen previously.

As seen previously, a `phi` instruction gets two values for the variable `i` from two basic blocks: `%entry` and `%loop`. In the preceding case, the `%entry` block represents the value assigned to the induction variable at the start of the loop (this is `1`). The next updated value of `i` comes from the `%loop` block, which completes one iteration of the loop.

See also

- To get a detailed overview of how loops are handled for C++ in Clang, visit `http://llvm.org/viewvc/llvm-project/cfe/trunk/lib/Parse/ParseExprCXX.cpp`

Handling user-defined operators – binary operators

User-defined operators are similar to the C++ concept of operator overloading, where a default definition of an operator is altered to operate on a wide variety of objects. Typically, operators are unary or binary operators. Implementing binary operator overloading is easier with the existing infrastructure. Unary operators need some additional code to handle. First, binary operator overloading will be defined, and then unary operator overloading will be looked into.

Getting ready

The first part is to define a binary operator for overloading. The logical OR operator (`|`) is a good example to start with. The `|` operator in our TOY language can be used as follows:

```
def binary | (LHS RHS)
if LHS then
1
else if RHS then
1
else
0;
```

As seen in the preceding code, if any of the values of the LHS or RHS are not equal to 0, then we return `1`. If both the LHS and RHS are null, then we return `0`.

How to do it...

Do the following steps:

1. The first step, as usual, is to append the `enum` states for the binary operator and return the enum states on encountering the `binary` keyword:

```
 enum Token_Type {
…
…
BINARY_TOKEN
}
static int get_token() {
…
…
if (Identifier_string == "in") return IN_TOKEN;
if (Identifier_string == "binary") return BINARY_TOKEN;
…
…
}
```

2. The next step is to add an AST for the same. Note that it doesn't need a new AST to be defined. It can be handled with the function declaration AST. We just need to modify it by adding a flag to represent whether it's a binary operator. If it is, then determine its precedence:

```
class FunctionDeclAST {
  std::string Func_Name;
  std::vector<std::string> Arguments;
  bool isOperator;
  unsigned Precedence;
public:
  FunctionDeclAST(const std::string &name, const
std::vector<std::string> &args,
                bool isoperator = false, unsigned prec = 0)
      : Func_Name(name), Arguments(args), isOperator(isoperator),
Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Arguments.size()
== 1; }
  bool isBinaryOp() const { return isOperator && Arguments.size()
== 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Func_Name[Func_Name.size() - 1];
  }
```

```
  unsigned getBinaryPrecedence() const { return Precedence;
}

  Function *Codegen();
};
```

3. Once the modified AST is ready, the next step is to modify the parser of the function declaration:

```
static FunctionDeclAST *func_decl_parser() {
  std::string FnName;

  unsigned Kind = 0;
  unsigned BinaryPrecedence = 30;

  switch (Current_token) {
  default:
    return 0;
  case IDENTIFIER_TOKEN:
    FnName = Identifier_string;
    Kind = 0;
    next_token();
    break;
  case UNARY_TOKEN:
    next_token();
    if (!isascii(Current_token))
      return 0;
    FnName = "unary";
    FnName += (char)Current_token;
    Kind = 1;
    next_token();
    break;
  case BINARY_TOKEN:
    next_token();
    if (!isascii(Current_token))
      return 0;
    FnName = "binary";
    FnName += (char)Current_token;
    Kind = 2;
    next_token();

    if (Current_token == NUMERIC_TOKEN) {
      if (Numeric_Val < 1 || Numeric_Val > 100)
        return 0;
      BinaryPrecedence = (unsigned)Numeric_Val;
```

```
      next_token();
    }
    break;
  }

  if (Current_token != '(')
    return 0;

  std::vector<std::string> Function_Argument_Names;
  while (next_token() == IDENTIFIER_TOKEN)
    Function_Argument_Names.push_back(Identifier_string);
  if (Current_token != ')')
    return 0;

  next_token();

  if (Kind && Function_Argument_Names.size() != Kind)
    return 0;

  return new FunctionDeclAST(FnName,
Function_Argument_Names, Kind != 0, BinaryPrecedence);
}
```

4. Then we modify the `Codegen()` function for the binary AST:

```
Value* BinaryAST::Codegen() {
 Value* L = LHS->Codegen();
Value* R = RHS->Codegen();
switch(Bin_Operator) {
case '+' : return Builder.CreateAdd(L, R, "addtmp");
case '-' : return Builder.CreateSub(L, R, "subtmp");
case '*': return Builder.CreateMul(L, R, "multmp");
case '/': return Builder.CreateUDiv(L, R, "divtmp");
case '<' :
L = Builder.CreateICmpULT(L, R, "cmptmp");
return Builder.CreateUIToFP(L, Type::getIntTy(getGlobalContext()),
"booltmp");
default :
break;
}
Function *F = TheModule->getFunction(std::string("binary")+Op);
  Value *Ops[2] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}
```

5. Next we modify the function definition; it can be defined as:

```
Function* FunctionDefnAST::Codegen() {
Named_Values.clear();
Function *TheFunction = Func_Decl->Codegen();
if (!TheFunction) return 0;
if (Func_Decl->isBinaryOp())
    Operator_Precedence [Func_Decl->getOperatorName()] = Func_
Decl->getBinaryPrecedence();
BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry",
TheFunction);
Builder.SetInsertPoint(BB);
if (Value* Return_Value = Body->Codegen()) {
    Builder.CreateRet(Return_Value);
…
…
```

How it works...

Do the following steps:

1. Compile the `toy.cpp` file:

```
$ g++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-
libs --libs core ` -O3 -o toy
```

2. Open an example file:

```
$ vi example
```

3. Write the following binary operator overloading code in the example file:

```
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;
```

4. Compile the example file with the TOY compiler:

```
$ ./toy example


output :


; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"
```

```
define i32 @"binary|"(i32 %LHS, i32 %RHS) {
entry:
  %ifcond = icmp eq i32 %LHS, 0
  %ifcond1 = icmp eq i32 %RHS, 0
  %. = select i1 %ifcond1, i32 0, i32 1
  %iftmp5 = select i1 %ifcond, i32 %., i32 1
  ret i32 %iftmp5
}
```

The binary operator we just defined will be parsed. Its definition is also parsed. Whenever the `|` binary operator is encountered, the LHS and RHS are initialized and the definition body is executed, giving the appropriate result as per the definition. In the preceding example, if either the LHS or RHS is nonzero, then the result is `1`. If both the LHS and RHS are zero, then the result is `0`.

See also

- For detailed examples on handling other binary operators, refer to `http://llvm.org/docs/tutorial/LangImpl6.html`

Handling user-defined operators – unary operators

We saw in the previous recipe how binary operators can be handled. A language may also have some unary operator, operating on 1 operand. In this recipe, we will see how to handle unary operators.

Getting ready

The first step is to define a unary operator in the TOY language. A simple unary NOT operator (`!`) can serve as a good example; let's see one definition:

```
def unary!(v)
  if v then
    0
  else
    1;
```

If the value `v` is equal to `1`, then `0` is returned. If the value is `0`, `1` is returned as the output.

How to do it...

Do the following steps:

1. The first step is to define the `enum` token for the unary operator in the `toy.cpp` file:

```
enum Token_Type {
…
…
BINARY_TOKEN,
UNARY_TOKEN
}
```

2. Then we identify the unary string and return a unary token:

```
static int get_token() {
…
…
if (Identifier_string == "in") return IN_TOKEN;
if (Identifier_string == "binary") return BINARY_TOKEN;
if (Identifier_string == "unary") return UNARY_TOKEN;

…
…
}
```

3. Next, we define an AST for the unary operator:

```
class ExprUnaryAST : public BaseAST {
  char Opcode;
  BaseAST *Operand;
public:
  ExprUnaryAST(char opcode, BaseAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};
```

4. The AST is now ready. Let's define a parser for the unary operator:

```
static BaseAST *unary_parser() {

  if (!isascii(Current_token) || Current_token == '(' || Current_
token == ',')
    return Base_Parser();

    int Op = Current_token;

  next_token();

  if (ExprAST *Operand = unary_parser())
```

```
    return new ExprUnaryAST(Opc, Operand);

return 0;
}
```

5. The next step is to call the `unary_parser()` function from the binary operator parser:

```
static BaseAST *binary_op_parser(int Old_Prec, BaseAST *LHS) {

  while (1) {
    int Operator_Prec = getBinOpPrecedence();

    if (Operator_Prec < Old_Prec)
      return LHS;

    int BinOp = Current_token;
    next_token();

    BaseAST *RHS = unary_parser();
    if (!RHS)
      return 0;

    int Next_Prec = getBinOpPrecedence();
    if (Operator_Prec < Next_Prec) {
      RHS = binary_op_parser(Operator_Prec + 1, RHS);
      if (RHS == 0)
        return 0;
    }

    LHS = new BinaryAST(std::to_string(BinOp), LHS, RHS);
  }
}
```

6. Now let's call the `unary_parser()` function from the expression parser:

```
static BaseAST *expression_parser() {
  BaseAST *LHS = unary_parser();
  if (!LHS)
    return 0;

  return binary_op_parser(0, LHS);
}
```

7. Then we modify the function declaration parser:

```
static FunctionDeclAST* func_decl_parser() {
std::string Function_Name = Identifier_string;
unsigned Kind = 0;
unsigned BinaryPrecedence = 30;
switch (Current_token) {
  default:
    return 0;
  case IDENTIFIER_TOKEN:
    Function_Name = Identifier_string;
    Kind = 0;
    next_token();
    break;
  case UNARY_TOKEN:
  next_token();
if (!isascii(Current_token))
      return0;
    Function_Name = "unary";
    Function_Name += (char)Current_token;
    Kind = 1;
    next_token();
    break;
  case BINARY_TOKEN:
    next_token();
    if (!isascii(Current_token))
      return 0;
    Function_Name = "binary";
    Function_Name += (char)Current_token;
    Kind = 2;
   next_token();
   if (Current_token == NUMERIC_TOKEN) {
      if (Numeric_Val < 1 || Numeric_Val > 100)
        return 0;
      BinaryPrecedence = (unsigned)Numeric_Val;
      next_token();
    }
    break;
  }
if (Current_token ! = '(') {
printf("error in function declaration");
return 0;
}
std::vector<std::string> Function_Argument_Names;
```

```
while(next_token() == IDENTIFIER_TOKEN)
Function_Argument_Names.push_back(Identifier_string);
if(Current_token != ')')  {                    printf("Expected
')' ");                      return 0;
}
next_token();
if (Kind && Function_Argument_Names.size() != Kind)
    return 0;
return new FunctionDeclAST(Function_Name, Function_Arguments_
Names, Kind !=0, BinaryPrecedence);
}
```

8. The final step is to define the `Codegen()` function for the unary operator:

```
Value *ExprUnaryAST::Codegen() {

  Value *OperandV = Operand->Codegen();

  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opco
de);

  if (F == 0)
    return 0;

  return Builder.CreateCall(F, OperandV, "unop");
}
```

How it works...

Do the following steps:

1. Compile the `toy.cpp` file:

```
$ g++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-
libs --libs core ` -O3 -o toy
```

2. Open an example file:

```
$ vi example
```

3. Write the following unary operator overloading code in the example file:

```
def unary!(v)
  if v then
    0
  else
    1;
```

4. Compile the example file with the TOY compiler:

```
$ ./toy example
```

The output should be as shown:

```
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @"unary!"(i32 %v) {
entry:
  %ifcond = icmp eq i32 %v, 0
  %. = select i1 %ifcond, i32 1, i32 0
  ret i32 %.
}
```

The unary operator defined by the user will be parsed, and IR will be generated for it. In the case you just saw, if the unary operand is not zero then the result is `0`. If the operand is zero, then the result is `1`.

See also

- To learn more detailed implementations of unary operators, visit `http://llvm.org/docs/tutorial/LangImpl6.html`

Adding JIT support

A wide variety of tools can be applied to LLVM IR. For example, as demonstrated in *Chapter 1*, *LLVM Design and Use*, the IR can be dumped into bitcode or into an assembly. An optimization tool called opt can be run on IR. IR acts as the common platform—an abstract layer for all of these tools.

JIT support can also be added. It immediately evaluates the top-level expressions typed in. For example, `1 + 2;`, as soon as it is typed in, evaluates the code and prints the value as `3`.

How to do it...

Do the following steps:

1. Define a static global variable for the execution engine in the `toy.cpp` file:

```
static ExecutionEngine *TheExecutionEngine;
```

2. In the `toy.cpp` file's `main()` function, write the code for JIT:

```
int main() {
…
…
init_precedence();
TheExecutionEngine = EngineBuilder(TheModule).create();
…
…
}
```

3. Modify the top-level expression parser in the `toy.cpp` file:

```
static void HandleTopExpression() {

if (FunctionDefAST *F = expression_parser())
   if (Function *LF = F->Codegen()) {
        LF -> dump();
       void *FPtr = TheExecutionEngine-
>getPointerToFunction(LF);
      int (*Int)() = (int (*)())(intptr_t)FPtr;

    printf("Evaluated to %d\n", Int());
}
   else
next_token();
}
```

How it works...

Do the following steps:

1. Compile the `toy.cpp` program:

```
$ g++ -g toy.cpp `llvm-config --cxxflags --ldflags --system-
libs --libs core mcjit native` -O3 -o toy
```

2. Open an example file:

```
$ vi example
```

3. Write the following TOY code in the example file:

```
…
4+5;
```

4. Finally, run the TOY compiler on the example file:

```
$ ./toy example
The output will be
define i32 @0() {
entry:
  ret i32 9
}
```

The LLVM JIT compiler matches the native platform ABI, casts the result pointer into a function pointer of that type, and calls it directly. There is no difference between JIT-compiled code and native machine code that is statically linked to the application.

4
Preparing Optimizations

In this chapter, we will cover the following recipes:

- Various levels of optimization
- Writing your own LLVM pass
- Running your own pass with the opt tool
- Using another pass in a new pass
- Registering a pass with pass manager
- Writing an analysis pass
- Writing an alias analysis pass
- Using other analysis passes

Introduction

Once the source code transformation completes, the output is in the LLVM IR form. This IR serves as a common platform for converting into assembly code, depending on the backend. However, before converting into an assembly code, the IR can be optimized to produce more effective code. The IR is in the SSA form, where every new assignment to a variable is a new variable itself—a classic case of an SSA representation.

In the LLVM infrastructure, a pass serves the purpose of optimizing LLVM IR. A pass runs over the LLVM IR, processes the IR, analyzes it, identifies the optimization opportunities, and modifies the IR to produce optimized code. The command-line interface **opt** is used to run optimization passes on LLVM IR.

In the upcoming chapters, various optimization techniques will be discussed. Also, how to write and register a new optimization pass will be explored.

Various levels of optimization

There are various levels of optimization, starting at 0 and going up to 3 (there is also `s` for space optimization). The code gets more and more optimized as the optimization level increases. Let's try to explore the various optimization levels.

Getting ready...

Various optimization levels can be understood by running the opt command-line interface on LLVM IR. For this, an example C program can first be converted to IR using the **Clang** frontend.

1. Open an `example.c` file and write the following code in it:

```
$ vi  example.c
int main(int argc, char **argv) {
  int i, j, k, t = 0;
  for(i = 0; i < 10; i++) {
    for(j = 0; j < 10; j++) {
      for(k = 0; k < 10; k++) {
        t++;
      }
    }
    for(j = 0; j < 10; j++) {
      t++;
    }
  }
  for(i = 0; i < 20; i++) {
    for(j = 0; j < 20; j++) {
      t++;
    }
    for(j = 0; j < 20; j++) {
      t++;
    }
  }
  return t;
}
```

2. Now convert this into LLVM IR using the `clang` command, as shown here:

```
$ clang -S -O0 -emit-llvm example.c
```

   A new file, `example.ll`, will be generated, containing LLVM IR. This file will be used to demonstrate the various optimization levels available.

How to do it...

Do the following steps:

1. The opt command-line tool can be run on the IR-generated `example.ll` file:

   ```
   $ opt –O0 –S example.ll
   ```

   The `–O0` syntax specifies the least optimization level.

2. Similarly, you can run other optimization levels:

   ```
   $ opt –O1 –S example.ll
   $ opt –O2 –S example.ll
   $ opt –O3 –S example.ll
   ```

How it works...

The opt command-line interface takes the `example.ll` file as the input and runs the series of passes specified in each optimization level. It can repeat some passes in the same optimization level. To see which passes are being used in each optimization level, you have to add the `--debug-pass=Structure` command-line option with the previous opt commands.

See Also

- To know more on various other options that can be used with the opt tool, refer to `http://llvm.org/docs/CommandGuide/opt.html`

Writing your own LLVM pass

All LLVM passes are subclasses of the `pass` class, and they implement functionality by overriding the virtual methods inherited from `pass`. LLVM applies a chain of analyses and transformations on the target program. A pass is an instance of the Pass LLVM class.

Getting ready

Let's see how to write a pass. Let's name the pass `function block counter`; once done, it will simply display the name of the function and count the basic blocks in that function when run. First, a `Makefile` needs to be written for the pass. Follow the given steps to write a `Makefile`:

1. Open a `Makefile` in the `llvm lib/Transform` folder:

   ```
   $ vi Makefile
   ```

2. Specify the path to the LLVM root folder and the library name, and make this pass a loadable module by specifying it in `Makefile`, as follows:

```
LEVEL = ../../..
LIBRARYNAME = FuncBlockCount
LOADABLE_MODULE = 1
include $(LEVEL)/Makefile.common
```

This `Makefile` specifies that all the `.cpp` files in the current directory are to be compiled and linked together in a shared object.

How to do it...

Do the following steps:

1. Create a new `.cpp` file called `FuncBlockCount.cpp`:

```
$ vi FuncBlockCount.cpp
```

2. In this file, include some header files from LLVM:

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

3. Include the `llvm` namespace to enable access to LLVM functions:

```
using namespace llvm;
```

4. Then start with an anonymous namespace:

```
namespace {
```

5. Next declare the pass:

```
struct FuncBlockCount : public FunctionPass {
```

6. Then declare the pass identifier, which will be used by LLVM to identify the pass:

```
static char ID;
FuncBlockCount() : FunctionPass(ID) {}
```

7. This step is one of the most important steps in writing a pass—writing a `run` function. Since this pass inherits `FunctionPass` and runs on a function, a `runOnFunction` is defined to be run on a function:

```
bool runOnFunction(Function &F) override {
      errs() << "Function " << F.getName() << '\n';
      return false;
    }
  };
}
```

This function prints the name of the function that is being processed.

8. The next step is to initialize the pass ID:

```
char FuncBlockCount::ID = 0;
```

9. Finally, the pass needs to be registered, with a command-line argument and a name:

```
static RegisterPass<FuncBlockCount> X("funcblockcount", "Function
Block Count", false, false);
```

Putting everything together, the entire code looks like this:

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
namespace {
struct FuncBlockCount : public FunctionPass {
  static char ID;
  FuncBlockCount() : FunctionPass(ID) {}
  bool runOnFunction(Function &F) override {
    errs() << "Function " << F.getName() << '\n';
    return false;
  }
          };
       }
      char FuncBlockCount::ID = 0;
      static RegisterPass<FuncBlockCount> X("funcblockcount",
"Function Block Count", false, false);
```

How it works

A simple `gmake` command compiles the file, so a new file `FuncBlockCount.so` is generated at the LLVM root directory. This shared object file can be dynamically loaded to the opt tool to run it on a piece of LLVM IR code. How to load and run it will be demonstrated in the next section.

See also

- To know more on how a pass can be built from scratch, visit `http://llvm.org/docs/WritingAnLLVMPass.html`

Running your own pass with the opt tool

The pass written in the previous recipe, *Writing your own LLVM pass*, is ready to be run on the LLVM IR. This pass needs to be loaded dynamically for the opt tool to recognize and execute it.

How to do it...

Do the following steps:

1. Write the C test code in the `sample.c` file, which we will convert into an `.ll` file in the next step:

```
$ vi sample.c

int foo(int n, int m) {
  int sum = 0;
  int c0;
  for (c0 = n; c0 > 0; c0--) {
    int c1 = m;
    for (; c1 > 0; c1--) {
      sum += c0 > c1 ? 1 : 0;
    }
  }
  return sum;
}
```

2. Convert the C test code into LLVM IR using the following command:

```
$ clang -O0 -S -emit-llvm sample.c -o sample.ll
```

   This will generate a `sample.ll` file.

3. Run the new pass with the opt tool, as follows:

```
$ opt  -load (path_to_.so_file)/FuncBlockCount.so
-funcblockcount sample.ll
```

   The output will look something like this:

```
Function foo
```

How it works...

As seen in the preceding code, the shared object loads dynamically into the opt command-line tool and runs the pass. It goes over the function and displays its name. It does not modify the IR. Further enhancement in the new pass is demonstrated in the next recipe.

See also

- To know more about the various types of the Pass class, visit `http://llvm.org/docs/WritingAnLLVMPass.html#pass-classes-and-requirements`

Using another pass in a new pass

A pass may require another pass to get some analysis data, heuristics, or any such information to decide on a further course of action. The pass may just require some analysis such as memory dependencies, or it may require the altered IR as well. The new pass that you just saw simply prints the name of the function. Let's see how to enhance it to count the basic blocks in a loop, which also demonstrates how to use other pass results.

Getting ready

The code used in the previous recipe remains the same. Some modifications are required, however, to enhance it—as demonstrated in next section—so that it counts the number of basic blocks in the IR.

How to do it...

The `getAnalysis` function is used to specify which other pass will be used:

1. Since the new pass will be counting the number of basic blocks, it requires loop information. This is specified using the `getAnalysis` loop function:

```
LoopInfo *LI = &getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```

2. This will call the `LoopInfo` pass to get information on the loop. Iterating through this object gives the basic block information:

```
unsigned num_Blocks = 0;
  Loop::block_iterator bb;
  for(bb = L->block_begin(); bb != L->block_end();++bb)
    num_Blocks++;
  errs() << "Loop level " << nest << " has " << num_Blocks
<< " blocks\n";
```

3. This will go over the loop to count the basic blocks inside it. However, it counts only the basic blocks in the outermost loop. To get information on the innermost loop, recursive calling of the `getSubLoops` function will help. Putting the logic in a separate function and calling it recursively makes more sense:

```
void countBlocksInLoop(Loop *L, unsigned nest) {
  unsigned num_Blocks = 0;
  Loop::block_iterator bb;
  for(bb = L->block_begin(); bb != L->block_end();++bb)
    num_Blocks++;
  errs() << "Loop level " << nest << " has " << num_Blocks
<< " blocks\n";
  std::vector<Loop*> subLoops = L->getSubLoops();
  Loop::iterator j, f;
  for (j = subLoops.begin(), f = subLoops.end(); j != f;
++j)
    countBlocksInLoop(*j, nest + 1);
}

virtual bool runOnFunction(Function &F) {
  LoopInfo *LI = &getAnalysis<LoopInfoWrapperPass>().
getLoopInfo();
  errs() << "Function " << F.getName() + "\n";
  for (Loop *L : *LI)
    countBlocksInLoop(L, 0);
  return false;
}
```

How it works...

The newly modified pass now needs to run on a sample program. Follow the given steps to modify and run the sample program:

1. Open the `sample.c` file and replace its content with the following program:

```
int main(int argc, char **argv) {
  int i, j, k, t = 0;
  for(i = 0; i < 10; i++) {
    for(j = 0; j < 10; j++) {
      for(k = 0; k < 10; k++) {
        t++;
      }
    }
    for(j = 0; j < 10; j++) {
      t++;
    }
```

```
  }
  for(i = 0; i < 20; i++) {
    for(j = 0; j < 20; j++) {
      t++;
    }
    for(j = 0; j < 20; j++) {
      t++;
    }
  }
  return t;
}
```

2. Convert it into a `.ll` file using Clang:

```
$ clang -O0 -S -emit-llvm sample.c -o sample.ll
```

3. Run the new pass on the previous sample program:

```
$ opt  -load (path_to_.so_file)/FuncBlockCount.so  -
funcblockcount sample.ll
```

The output will look something like this:

```
Function main
Loop level 0 has 11 blocks
Loop level 1 has 3 blocks
Loop level 1 has 3 blocks
Loop level 0 has 15 blocks
Loop level 1 has 7 blocks
Loop level 2 has 3 blocks
Loop level 1 has 3 blocks
```

There's more...

The LLVM's pass manager provides a debug pass option that gives us the chance to see which passes interact with our analyses and optimizations, as follows:

```
$ opt  -load (path_to_.so_file)/FuncBlockCount.so  -
funcblockcount sample.ll -disable-output -debug-pass=Structure
```

Registering a pass with pass manager

Until now, a new pass was a dynamic object that was run independently. The opt tool consists of a pipeline of such passes that are registered with the pass manager, and a part of LLVM. Let's see how to register our pass with the Pass Manager.

Getting ready

The `PassManager` class takes a list of passes, ensures that their prerequisites are set up correctly, and then schedules the passes to run efficiently. The Pass Manager does two main tasks to try to reduce the execution time of a series of passes:

- Shares the analysis results to avoid recomputing analysis results as much as possible
- Pipelines the execution of passes to the program to get better cache and memory usage behavior out of a series of passes by pipelining the passes together

How to do it...

Follow the given steps to register a pass with Pass Manager:

1. Define a `DEBUG_TYPE` macro, specifying the debugging name in the `FuncBlockCount.cpp` file:

```
#define DEBUG_TYPE "func-block-count"
```

2. In the `FuncBlockCount` struct, specify the `getAnalysisUsage` syntax as follows:

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<LoopInfoWrapperPass>();
  }
```

3. Now initialize the macros for initialization of the new pass:

```
INITIALIZE_PASS_BEGIN(FuncBlockCount, " funcblockcount ",
                      "Function Block Count", false, false)
INITIALIZE_PASS_DEPENDENCY(LoopInfoWrapperPass)

INITIALIZE_PASS_END(FuncBlockCount, "funcblockcount",
                    "Function Block Count", false, false)

Pass *llvm::createFuncBlockCountPass() { return new
FuncBlockCount(); }
```

4. Add the `createFuncBlockCount` Pass function in the `LinkAllPasses.h` file, located at `include/llvm/`:

```
(void) llvm:: createFuncBlockCountPass ();
```

5. Add the declaration to the `Scalar.h` file, located at `include/llvm/Transforms`:

```
Pass * createFuncBlockCountPass ();
```

6. Also modify the constructor of the pass:

```
FuncBlockCount() : FunctionPass(ID) {initializeFuncBlockCount Pass
(*PassRegistry::getPassRegistry());}
```

7. In the `Scalar.cpp file`, located at `lib/Transforms/Scalar/`, add the initialization pass entry:

```
initializeFuncBlockCountPass (Registry);
```

8. Add this initialization declaration to the `InitializePasses.h` file, which is located at `include/llvm/`:

```
void initializeFuncBlockCountPass (Registry);
```

9. Finally, add the `FuncBlockCount.cpp` filename to the `CMakeLists.txt` file, located at `lib/Transforms/Scalar/`:

```
FuncBlockCount.cpp
```

How it works...

Compile the LLVM with the `cmake` command as specified in *Chapter 1*, *LLVM Design and Use*. The Pass Manager will include this pass in the pass pipeline of the opt command-line tool. Also, this pass can be run in isolation from the command line:

```
$ opt -funcblockcount sample.ll
```

See Also

- To know more about adding a pass in Pass Manager in simple steps, study the LoopInstSimplify pass at `http://llvm.org/viewvc/llvm-project/llvm/trunk/lib/Transforms/Scalar/LoopInstSimplify.cpp`

Writing an analysis pass

The analysis pass provides higher-level information about IR without actually changing the IR. The results that the analysis pass provides can be used by another analysis pass to compute its result. Also, once an analysis pass calculates the result, its result can be used several times by different passes until the IR on which this pass was run is changed. In this recipe, we will write an analysis pass that counts and outputs the number of opcodes used in a function.

Getting ready

First of all, we write the test code on which we will be running our pass:

```
$ cat testcode.c
int func(int a, int b){
  int sum = 0;
  int iter;
  for (iter = 0; iter < a; iter++) {
```

```
      int iter1;
      for (iter1 = 0; iter1 < b; iter1++) {
        sum += iter > iter1 ? 1 : 0;
      }
    }
    return sum;
  }
```

Transform this into a `.bc` file, which we will use as the input to the analysis pass:

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

Now create the file containing the pass source code in `llvm_root_dir/lib/Transforms/opcodeCounter`. Here, `opcodeCounter` is the directory we have created, and it is where our pass's source code will reside.

Make the necessary `Makefile` changes so that this pass can be compiled.

How to do it...

Now let's start writing the source code for our analysis pass:

1. Include the necessary header files and use the `llvm` namespace:

```
#define DEBUG_TYPE "opcodeCounter"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;
```

2. Create the structure defining the pass:

```
namespace {
struct CountOpcode: public FunctionPass {
```

3. Within the structure, create the necessary data structures to count the number of opcodes and to denote the pass ID of the pass:

```
std::map< std::string, int> opcodeCounter;
static char ID;
CountOpcode () : FunctionPass(ID) {}
```

4. Within the preceding structure, write the code for the actual implementation of the pass, overloading the `runOnFunction` function:

```
virtual bool runOnFunction (Function &F) {
 llvm::outs() << "Function " << F.getName () << '\n';
for ( Function::iterator bb = F.begin(), e = F.end(); bb !=
e; ++bb) {
```

```
  for ( BasicBlock::iterator i = bb->begin(), e = bb->end();
  i!= e; ++i) {
    if(opcodeCounter.find(i->getOpcodeName()) ==
opcodeCounter.end()) {
    opcodeCounter[i->getOpcodeName()] = 1;
    } else {
    opcodeCounter[i->getOpcodeName()] += 1;
    }
  }
}

std::map< std::string, int>::iterator i =
opcodeCounter.begin();
std::map< std::string, int>::iterator e =
opcodeCounter.end();
while (i != e) {
  llvm::outs()  << i->first << ": " << i->second << "\n";
  i++;
}
llvm::outs()  << "\n";
opcodeCounter.clear();
return false;
}
};
}
```

5. Write the code for registering the pass:

```
char CountOpcode::ID = 0;
static RegisterPass<CountOpcode> X("opcodeCounter", "Count
number of opcode in a functions");
```

6. Compile the pass using the `make` or `cmake` command.
7. Run the pass on the test code using the opt tool to get the information on the number of opcodes present in the function:

```
$ opt -load path-to-build-folder/lib/LLVMCountopcodes.so
-opcodeCounter -disable-output testcode.bc
Function func
add: 3
alloca: 5
br: 8
icmp: 3
load: 10
ret: 1
select: 1
store: 8
```

How it works...

This analysis pass works on a function level, running once for each function in the program. Hence, we have inherited the `FunctionPass` function when declaring the `CountOpcodes : public FunctionPass` struct.

The `opcodeCounter` function keeps a count of every opcode that has been used in the function. In the following for loops, we collect the opcodes from all the functions:

```
for (Function::iterator bb = F.begin(), e = F.end(); bb != e;
++bb) {
for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e;
++i) {
```

The first `for` loop iterates over all the basic blocks present in the function, and the second for loop iterates over all the instructions present in the basic block.

The code in the first `for` loop is the actual code that collects the opcodes and their numbers. The code below the `for` loops is meant for printing the results. As we have used a map to store the result, we iterate over it to print the pair of the opcode name and its number in the function.

We return `false` because we are not modifying anything in the test code. The last two lines of the code are meant for registering this pass with the given name so that the opt tool can use this pass.

Finally, on execution of the test code, we get the output as different opcodes used in the function and their numbers.

Writing an alias analysis pass

Alias analysis is a technique by which we get to know whether two pointers point to the same location—that is, whether the same location can be accessed in more ways than one. By getting the results of this analysis, you can decide about further optimizations, such as common subexpression elimination. There are different ways and algorithms to perform alias analysis. In this recipe, we will not deal with these algorithms, but we will see how LLVM provides the infrastructure to write your own alias analysis pass. In this recipe, we will write an alias analysis pass to see how to get started with writing such a pass. We will not make use of any specific algorithm, but will return the `MustAlias` response in every case of the analysis.

Getting ready

Write the test code that will be the input for alias analysis. Here, we will take the `testcode.c` file used in the previous recipe as the test code.

Make the necessary `Makefile` changes, make changes to register the pass by adding entries for the pass in `llvm/lib/Analysis/Analysis.cpp llvm/include/llvm/InitializePasses.h`, `llvm/include/llvm/LinkAllPasses.h`, `llvm/include/llvm/Analysis/Passes.h` and create a file in `llvm_source_dir/lib/Analysis/` named `EverythingMustAlias.cpp` that will contain the source code for our pass.

How to do it...

Do the following steps:

1. Include the necessary header files and use the `llvm` namespace:

```
#include "llvm/Pass.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
using namespace llvm;
```

2. Create a structure for our pass by inheriting the `ImmutablePass` and `AliasAnalysis` classes:

```
namespace {
struct EverythingMustAlias : public ImmutablePass, public
AliasAnalysis {
```

3. Declare the data structures and constructor:

```
static char ID;
EverythingMustAlias() : ImmutablePass(ID) {}
initializeEverythingMustAliasPass(*PassRegistry::getPassRegist
ry());}
```

4. Implement the `getAdjustedAnalysisPointer` function:

```
    void *getAdjustedAnalysisPointer(const void *ID) override {
      if (ID == &AliasAnalysis::ID)
        return (AliasAnalysis*)this;
      return this;
    }
```

5. Implement the `initializePass` function to initialize the pass:

```
bool doInitialization(Module &M) override {
     DL = &M.getDataLayout();
      return true;
    }
```

6. Implement the `alias` function:

```
void *getAdjustedAnalysisPointer(const void *ID) override {
      if (ID == &AliasAnalysis::ID)
        return (AliasAnalysis*)this;
      return this;
    }
};
}
```

7. Register the pass:

```
char EverythingMustAlias::ID = 0;
INITIALIZE_AG_PASS(EverythingMustAlias, AliasAnalysis, "must-aa",
"Everything Alias (always returns 'must' alias)", true, true,
 true)

ImmutablePass *llvm::createEverythingMustAliasPass() { return new
EverythingMustAlias(); }
```

8. Compile the pass using the `cmake` or `make` command.
9. Execute the test code using the `.so` file that is formed after compiling the pass:

```
$ opt  -must-aa -aa-eval -disable-output testcode.bc
===== Alias Analysis Evaluator Report =====
 10 Total Alias Queries Performed
 0 no alias responses (0.0%)
 0 may alias responses (0.0%)
 0 partial alias responses (0.0%)
 10 must alias responses (100.0%)
 Alias Analysis Evaluator Pointer Alias Summary:
0%/0%/0%/100%
 Alias Analysis Mod/Ref Evaluator Summary: no mod/ref!
```

How it works...

The `AliasAnalysis` class gives the interface that the various alias analysis implementations should support. It exports the `AliasResult` and `ModRefResult` enums, representing the results of the `alias` and `modref` query respectively.

The `alias` method is used to check whether two memory objects are pointing to the same location or not. It takes two memory objects as the input and returns `MustAlias`, `PartialAlias`, `MayAlias`, or `NoAlias` as appropriate.

The `getModRefInfo` method returns the information on whether the execution of an instruction can read or modify a memory location. The pass in the preceding example works by returning the value `MustAlias` for every set of two pointers, as we have implemented it that way. Here, we have inherited the `ImmutablePasses` class, which suits our pass, as it is a very basic pass. We have inherited the `AliasAnalysis` pass, which provides the interface for our implementation.

The `getAdjustedAnalysisPointer` function is used when a pass implements an analysis interface through multiple inheritance. If needed, it should override this to adjust the pointer as required for the specified pass information.

The `initializePass` function is used to initialize the pass that contains the `InitializeAliasAnalysis` method, which should contain the actual implementation of the alias analysis.

The `getAnalysisUsage` method is used to declare any dependency on other passes by explicitly calling the `AliasAnalysis::getAnalysisUsage` method.

The `alias` method is used to determine whether two memory objects alias each other or not. It takes two memory objects as the input and returns the `MustAlias`, `PartialAlias`, `MayAlias`, or `NoAlias` responses as appropriate.

The code following the `alias` method is meant for registering the pass. Finally, when we use this pass over the test code, we get 10 `MustAlias` responses (`100.0%`) as the result, as implemented in our pass.

See also

For a more detailed insight into LLVM alias analysis, refer to `http://llvm.org/docs/AliasAnalysis.html`.

Using other analysis passes

In this recipe, we will take a brief look into the other analysis passes that are provided by LLVM and can be used to get analysis information about a basic block, function, module, and so on. We will look into passes that have already been implemented in LLVM, and how we can use them for our purpose. We will not go through all the passes but take a look at only some of them.

Getting ready...

Write the test code in the `testcode1.c` file, which will be used for analysis purposes:

```
$ cat testcode1.c
void func() {
int i;
char C[2];
char A[10];
for(i = 0; i != 10; ++i) {
  ((short*)C)[0] = A[i];
  C[1] = A[9-i];
}
}
```

Convert the C code to bitcode format, using the following command line:

```
$ clang -c -emit-llvm testcode1.c -o testcode1.bc
```

How to do it...

Follow the steps given to use other analysis passes:

1. Use the alias analysis evaluator pass by passing `-aa-eval` as a command-line option to the opt tool:

```
$ opt -aa-eval -disable-output testcode1.bc
===== Alias Analysis Evaluator Report =====
36 Total Alias Queries Performed
0 no alias responses (0.0%)
36 may alias responses (100.0%)
0 partial alias responses (0.0%)
0 must alias responses (0.0%)
Alias Analysis Evaluator Pointer Alias Summary: 0%/100%/0%/0%
Alias Analysis Mod/Ref Evaluator Summary: no mod/ref!
```

2. Print the dominator tree information using the `-print-dom-info` command-line option along with opt:

```
$ opt  -print-dom-info -disable-output testcode1.bc
=================================-------------------------------------
Inorder Dominator Tree:
  [1] %0 {0,9}
    [2] %1 {1,8}
      [3] %4 {2,5}
        [4] %19 {3,4}
      [3] %22 {6,7}
```

3. Count the number of queries made by one pass to another using the `-count-aa` command-line option along with opt:

```
$ opt -count-aa -basicaa -licm -disable-output testcode1.bc
No alias:    [4B] i32* %i, [1B] i8* %7
No alias:    [4B] i32* %i, [2B] i16* %12
No alias:    [1B] i8* %7, [2B] i16* %12
No alias:    [4B] i32* %i, [1B] i8* %16
Partial alias:    [1B] i8* %7, [1B] i8* %16
No alias:    [2B] i16* %12, [1B] i8* %16
Partial alias:    [1B] i8* %7, [1B] i8* %16
No alias:    [4B] i32* %i, [1B] i8* %18
No alias:    [1B] i8* %18, [1B] i8* %7
No alias:    [1B] i8* %18, [1B] i8* %16
Partial alias:    [2B] i16* %12, [1B] i8* %18
Partial alias:    [2B] i16* %12, [1B] i8* %18

===== Alias Analysis Counter Report =====
 Analysis counted:
  12 Total Alias Queries Performed
  8 no alias responses (66%)
  0 may alias responses (0%)
  4 partial alias responses (33%)
  0 must alias responses (0%)
  Alias Analysis Counter Summary: 66%/0%/33%/0%

  0 Total Mod/Ref Queries Performed
```

4. Print the alias sets in a program using the `-print-alias-sets` command-line option with opt:

```
$ opt  -basicaa -print-alias-sets -disable-output testcode1.bc
Alias Set Tracker: 3 alias sets for 5 pointer values.
  AliasSet[0x336b120, 1] must alias, Mod/Ref   Pointers: (i32*
%i, 4)
  AliasSet[0x336b1c0, 2] may alias, Ref       Pointers: (i8*
%7, 1), (i8* %16, 1)
  AliasSet[0x338b670, 2] may alias, Mod       Pointers: (i16*
%12, 2), (i8* %18, 1)
```

How it works...

In the first case, where we use the `-aa-eval` option, the opt tool runs the alias analysis evaluator pass, which outputs the analysis on the screen. It iterates through all pairs of pointers in the function and queries whether the two are aliases of each other or not.

Using the `-print-dom-info` option, the pass for printing the dominator tree is run, through which information about the dominator tree can be obtained.

In the third case, we execute the `opt -count-aa -basicaa –licm` command. The `count-aa` command option counts the number of queries made by the `licm` pass to the `basicaa` pass. This information is obtained by the count alias analysis pass using the opt tool.

To print all the alias sets within a program, we use the `- print-alias-sets` command-line option. In this case, it prints the alias sets obtained after analyzing with the `basicaa` pass.

See also

Refer to `http://llvm.org/docs/Passes.html#analysis-passes` to know about more passes not mentioned here.

5
Implementing Optimizations

In this chapter, we will cover the following recipes:

- Writing a dead code elimination pass
- Writing an inlining transformation pass
- Writing a pass for memory optimization
- Combining LLVM IR
- Transforming and optimizing loops
- Reassociating expressions
- Vectorizing IR
- Other optimization passes

Introduction

In the previous chapter, we saw how to write a pass in LLVM. We also demonstrated writing a few analysis passes with an example of alias analysis. Those passes just read the source code and gave us information about it. In this chapter, we will go further and write transformation passes that will actually change the source code, trying to optimize it for the faster execution of code. In the first two recipes, we will show you how a transformation pass is written and how it changes the code. After that, we will see how we can make changes in the code of passes to tinker with the behavior of the passes.

Writing a dead code elimination pass

In this recipe, you will learn how to eliminate dead code from the program. By dead code elimination, we mean removing the code that has no effect whatsoever on the results that the source program outputs on executing. The main reasons to do so are reduction of the program size, which makes the code quality good and makes it easier to debug the code later on; and improving the run time of the program, as the unnecessary code is prevented from being executed. In this recipe, we will show you a variant of dead code elimination, called aggressive dead code elimination, that assumes every piece of code to be dead until proven otherwise. We will see how to implement this pass ourselves, and what modifications we need to make so that the pass can run just like other passes in the `lib/Transforms/Scalar` folder of the LLVM trunk.

Getting ready

To show the implementation of dead code elimination, we will need a piece of test code, on which we will run the aggressive dead code elimination pass:

```
$ cat testcode.ll
declare i32 @strlen(i8*) readonly nounwind
define void @test() {
  call i32 @strlen( i8* null )
  ret void
}
```

In this test code, we can see that a call to the `strlen` function is made in the `test` function, but the return value is not used. So, this should be treated as dead code by our pass.

In the file, include the `InitializePasses.h` file, located at `/llvm/`; and in the `llvm` namespace, add an entry for the pass that we are going to write:

```
namespace llvm {
…
…
void initializeMYADCEPass(PassRegistry&);    // Add this line
```

In the `scalar.h` file, located at `include/llvm-c/scalar.h/Transform/`, add the entry for the pass:

```
void LLVMAddMYAggressiveDCEPass(LLVMPassManagerRef PM);
```

In the `include/llvm/Transform/scalar.h` file, add the entry for the pass in the `llvm` namespace:

```
FunctionPass *createMYAggressiveDCEPass();
```

In the `lib/Transforms/Scalar/scalar.cpp` file, add the entry for the pass in two places. In the `void llvm::initializeScalarOpts(PassRegistry &Registry)` function, add the following code:

```
initializeMergedLoadStoreMotionPass(Registry);  // already present
in the file
initializeMYADCEPass(Registry);    // add this line
initializeNaryReassociatePass(Registry);  // already present in
the file
…
…
void LLVMAddMemCpyOptPass(LLVMPassManagerRef PM) {
  unwrap(PM)->add(createMemCpyOptPass());
}

// add the following three lines
void LLVMAddMYAggressiveDCEPass(LLVMPassManagerRef PM) {
  unwrap(PM)->add(createMYAggressiveDCEPass());
}

void LLVMAddPartiallyInlineLibCallsPass(LLVMPassManagerRef PM) {
  unwrap(PM)->add(createPartiallyInlineLibCallsPass());
}
…
```

How to do it...

We will now write the code for the pass:

1. Include the necessary header files:

```
#include "llvm/Transforms/Scalar.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/CFG.h"
#include "llvm/IR/InstIterator.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/Pass.h"
using namespace llvm;
```

2. Declare the structure of our pass:

```
namespace {
struct MYADCE : public FunctionPass {
  static char ID; // Pass identification, replacement for
typeid
  MYADCE() : FunctionPass(ID) {
    initializeMYADCEPass(*PassRegistry::getPassRegistry());
  }

  bool runOnFunction(Function& F) override;

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

3. Initialize the pass and its ID:

```
char MYADCE::ID = 0;
INITIALIZE_PASS(MYADCE, "myadce", "My Aggressive Dead Code
Elimination", false, false)
```

4. Implement the actual pass in the `runOnFunction` function:

```
bool MYADCE::runOnFunction(Function& F) {
  if (skipOptnoneFunction(F))
    return false;

  SmallPtrSet<Instruction*, 128> Alive;
  SmallVector<Instruction*, 128> Worklist;

  // Collect the set of "root" instructions that are known
live.
  for (Instruction &I : inst_range(F)) {
    if (isa<TerminatorInst>(I) || isa<DbgInfoIntrinsic>(I)
     || isa<LandingPadInst>(I) || I.mayHaveSideEffects()) {
      Alive.insert(&I);
      Worklist.push_back(&I);
    }
  }

  // Propagate liveness backwards to operands.
  while (!Worklist.empty()) {
    Instruction *Curr = Worklist.pop_back_val();
    for (Use &OI : Curr->operands()) {
```

```
      if (Instruction *Inst = dyn_cast<Instruction>(OI))
        if (Alive.insert(Inst).second)
          Worklist.push_back(Inst);
    }
  }

// the instructions which are not in live set are
considered dead in this pass. The instructions which do not
effect the control flow, return value and do not have any
side effects are hence deleted.
  for (Instruction &I : inst_range(F)) {
    if (!Alive.count(&I)) {
      Worklist.push_back(&I);
      I.dropAllReferences();
    }
  }

  for (Instruction *&I : Worklist) {
    I->eraseFromParent();
  }

  return !Worklist.empty();
}
}


FunctionPass *llvm::createMYAggressiveDCEPass() {
  return new MYADCE();
}
```

5. Run the preceding pass after compiling the `testcode.ll` file, which can be found in the *Getting ready* section of this recipe:

```
$ opt -myadce -S testcode.ll

; ModuleID = 'testcode.ll'

; Function Attrs: nounwind readonly
declare i32 @strlen(i8*) #0

define void @test() {
  ret void
}
```

How it works...

The pass works by first collecting a list of all the root instructions that are live in the first `for` loop of the `runOnFunction` function.

Using this information, we move backwards, propagating liveness to the operands in the `while (!Worklist.empty())` loop.

In the next `for` loop, we remove the instructions that are not live, that is, dead. Also, we check whether any reference was made to these values. If so, we drop all such references, which are also dead.

On running the the pass on the test code, we see the dead code; the call to the `strlen` function is removed.

Note that the code has been added to the LLVM trunk revision number 234045. So, when you are actually trying to implement it, some definitions might be updated. In this case, modify the code accordingly.

See also

For various other kinds of dead code elimination method, you can refer to the `llvm/lib/Transfroms/Scalar` folder, where the code for other kinds of DCEs is present.

Writing an inlining transformation pass

As we know, by inlining we mean expanding the function body of the function called at the call site, as it may prove useful through faster execution of code. The compiler takes the decision whether to inline a function or not. In this recipe, you will learn to how to write a simple function-inlining pass that makes use of the implementation in LLVM for inlining. We will write a pass that will handle the functions marked with the `alwaysinline` attribute.

Getting ready

Let's write a test code that we will run our pass on. Make the necessary changes in the `lib/Transforms/IPO/IPO.cpp` and `include/llvm/InitializePasses.h` files, the `include/llvm/Transforms/IPO.h` file, and the `/include/llvm-c/Transforms/IPO.h` file to include the following pass. Also make the necessary `makefile` changes to include his pass:

```
$ cat testcode.c
define i32 @inner1() alwaysinline {
  ret i32 1
}
```

```
define i32 @outer1() {
  %r = call i32 @inner1()
  ret i32 %r
}
```

How to do it...

We will now write the code for the pass:

1. Include the necessary header files:

```
#include "llvm/Transforms/IPO.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/AssumptionCache.h"
#include "llvm/Analysis/CallGraph.h"
#include "llvm/Analysis/InlineCost.h"
#include "llvm/IR/CallSite.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/Transforms/IPO/InlinerPass.h"

using namespace llvm;
```

2. Describe the class for our pass:

```
namespace {

class MyInliner : public Inliner {
  InlineCostAnalysis *ICA;

public:
    MyInliner() : Inliner(ID, -2000000000,
/*InsertLifetime*/ true),
                  ICA(nullptr) {
    initializeMyInlinerPass(*PassRegistry::getPassRegistry());
```

```
  }

  MyInliner(bool InsertLifetime)
      : Inliner(ID, -2000000000, InsertLifetime), ICA(nullptr) {
    initializeMyInlinerPass(*PassRegistry::getPassRegistry());
  }

  static char ID;

  InlineCost getInlineCost(CallSite CS) override;

  void getAnalysisUsage(AnalysisUsage &AU) const override;
  bool runOnSCC(CallGraphSCC &SCC) override;

  using llvm::Pass::doFinalization;
  bool doFinalization(CallGraph &CG) override {
    return removeDeadFunctions(CG, /*AlwaysInlineOnly=*/
true);
  }
};
}
```

3. Initialize the pass and add the dependencies:

```
char MyInliner::ID = 0;
INITIALIZE_PASS_BEGIN(MyInliner, "my-inline",
                "Inliner for always_inline functions", false,
false)
INITIALIZE_AG_DEPENDENCY(AliasAnalysis)
INITIALIZE_PASS_DEPENDENCY(AssumptionTracker)
INITIALIZE_PASS_DEPENDENCY(CallGraphWrapperPass)
INITIALIZE_PASS_DEPENDENCY(InlineCostAnalysis)
INITIALIZE_PASS_END(MyInliner, "my-inline",
                "Inliner for always_inline functions", false,
false)

Pass *llvm::createMyInlinerPass() { return new
MyInliner(); }
```

```
Pass *llvm::createMynlinerPass(bool InsertLifetime) {
  return new MyInliner(InsertLifetime);
}
```

4. Implement the function to get the inlining cost:

```
InlineCost MyInliner::getInlineCost(CallSite CS) {
  Function *Callee = CS.getCalledFunction();
if (Callee && !Callee->isDeclaration() &&
      CS.hasFnAttr(Attribute::AlwaysInline) &&
      ICA->isInlineViable(*Callee))
    return InlineCost::getAlways();

  return InlineCost::getNever();
}
```

5. Write the other helper methods:

```
bool MyInliner::runOnSCC(CallGraphSCC &SCC) {
  ICA = &getAnalysis<InlineCostAnalysis>();
  return Inliner::runOnSCC(SCC);
}

void MyInliner::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.addRequired<InlineCostAnalysis>();
  Inliner::getAnalysisUsage(AU);
}
```

6. Compile the pass. After compiling, run it on the preceding test case:

```
$ opt -inline-threshold=0 -always-inline -S test.ll

; ModuleID = 'test.ll'

; Function Attrs: alwaysinline
define i32 @inner1() #0 {
  ret i32 1
}
define i32 @outer1() {
  ret i32 1
}
```

How it works...

This pass that we have written will work for the functions with the `alwaysinline` attribute. The pass will always inline such functions.

The main function at work here is `InlineCost getInlineCost(CallSite CS)`. This is a function in the `inliner.cpp` file, which needs to be overridden here. So, on the basis of the inlining cost calculated here, we decide whether or not to inline a function. The actual implementation, on how the inlining process works, is in the `inliner.cpp` file.

In this case, we return `InlineCost::getAlways()`; for the functions marked with the `alwaysinline` attribute. For the others, we return `InlineCost::getNever()`. In this way, we can implement inlining for this simple case. If you want to dig deeper and try other variations of inlining—and learn how to make decisions about inlining—you can check out the `inlining.cpp` file.

When this pass is run over the test code, we see that the call of the `inner1` function is replaced by its actual function body.

Writing a pass for memory optimization

In this recipe, we will briefly discuss a transformation pass that deals with memory optimization.

Getting ready

For this recipe, you will need the opt tool installed.

How to do it...

1. Write the test code on which we will run the `memcpy` optimization pass:

```
$ cat memcopytest.ll
@cst = internal constant [3 x i32] [i32 -1, i32 -1, i32 -1],
align 4


declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8*
nocapture, i64, i32, i1) nounwind
declare void @foo(i32*) nounwind


define void @test1() nounwind {
  %arr = alloca [3 x i32], align 4
  %arr_i8 = bitcast [3 x i32]* %arr to i8*
```

```
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %arr_i8, i8* bitcast
([3 x i32]* @cst to i8*), i64 12, i32 4, i1 false)
  %arraydecay = getelementptr inbounds [3 x i32], [3 x i32]*
%arr, i64 0, i64 0
  call void @foo(i32* %arraydecay) nounwind
  ret void
}
```

2. Run the `memcpyopt` pass on the preceding test case:

```
$ opt -memcpyopt -S memcopytest.ll
; ModuleID = ' memcopytest.ll'

@cst = internal constant [3 x i32] [i32 -1, i32 -1, i32 -1],
align 4

; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8*
nocapture readonly, i64, i32, i1) #0

; Function Attrs: nounwind
declare void @foo(i32*) #0

; Function Attrs: nounwind
define void @test1() #0 {
  %arr = alloca [3 x i32], align 4
  %arr_i8 = bitcast [3 x i32]* %arr to i8*
  call void @llvm.memset.p0i8.i64(i8* %arr_i8, i8 -1, i64 12,
i32 4, i1 false)
  %arraydecay = getelementptr inbounds [3 x i32]* %arr, i64 0,
i64 0
  call void @foo(i32* %arraydecay) #0
  ret void
}

; Function Attrs: nounwind
declare void @llvm.memset.p0i8.i64(i8* nocapture, i8, i64,
i32, i1) #0

attributes #0 = { nounwind }
```

How it works...

The `Memcpyopt` pass deals with eliminating the `memcpy` calls wherever possible, or transforms them into other calls.

Consider this `memcpy` call:

```
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %arr_i8, i8* bitcast ([3 x
i32]* @cst to i8*), i64 12, i32 4, i1 false).
```

In the preceding test case, this pass converts it into a `memset` call:

```
call void @llvm.memset.p0i8.i64(i8* %arr_i8, i8 -1, i64 12, i32 4, i1
false)
```

If we look into the source code of the pass, we realize that this transformation is brought about by the `tryMergingIntoMemset` function in the `MemCpyOptimizer.cpp` file in `llvm/lib/Transforms/Scalar`.

The `tryMergingIntoMemset` function looks for some other pattern to fold away when scanning forward over instructions. It looks for stores in the neighboring memory and, on seeing consecutive ones, it attempts to merge them together into `memset`.

The `processMemSet` function looks out for any other neighboring `memset` to this `memset`, which helps us widen out the `memset` call to create a single larger store.

See also

To see the details of the various types of memory optimization passes, go to `http://llvm.org/docs/Passes.html#memcpyopt-memcpy-optimization`.

Combining LLVM IR

In this recipe, you will learn about instruction combining in LLVM. By instruction combining, we mean replacing a sequence of instructions with more efficient instructions that produce the same result in fewer machine cycles. In this recipe, we will see how we can make modifications in the LLVM code to combine certain instructions.

Getting started

To test our implementation, we will write test code that we will use to verify that our implementation is working properly to combine instructions:

```
define i32 @test19(i32 %x, i32 %y, i32 %z) {
 %xor1 = xor i32 %y, %z
```

```
 %or = or i32 %x, %xor1
 %xor2 = xor i32 %x, %z
 %xor3 = xor i32 %xor2, %y
 %res = xor i32 %or, %xor3
 ret i32 %res
}
```

How to do it...

1. Open the `lib/Transforms/InstCombine/InstCombineAndOrXor.cpp` file.
2. In the `InstCombiner::visitXor(BinaryOperator &I)` function, go to the `if` condition—`if (Op0I && Op1I)`—and add this:

```
if (match(Op0I, m_Or(m_Xor(m_Value(B), m_Value(C)), m_Value(A)))
&& match(Op1I, m_Xor( m_Xor(m_Specific(A), m_Specific(C)),
m_Specific(B)))) {

      return BinaryOperator::CreateAnd(A, Builder-
>CreateXor(B,C)); }
```

3. Now build LLVM again so that the Opt tool can use the new functionality and run the test case in this way:

```
Opt -instcombine -S testcode.ll
define i32 @test19(i32 %x, i32 %y, i32 %z) {
%1 = xor i32 %y, %z
  %res = and i32 %1, %x
  ret i32 %res
}
```

How it works...

In this recipe, we added code to the instruction combining file, which handles transformations involving the AND, OR, and XOR operators.

We added code for matching the pattern of the `(A | (B ^ C)) ^ ((A ^ C) ^ B)` form, and reduced it to `A & (B ^ C)`. The `if (match(Op0I, m_Or(m_Xor(m_Value(B), m_Value(C)), m_Value(A))) && match(Op1I, m_Xor( m_Xor(m_Specific(A), m_Specific(C)), m_Specific(B))))` line looks out for the pattern similar to the one shown at the start of this paragraph.

The `return BinaryOperator::CreateAnd(A, Builder->CreateXor(B,C));` line returns the reduced value after building a new instruction, replacing the previous matched code.

When we run the `instcombine` pass over the test code, we get the reduced result. You can see the number of operations is reduced from five to two.

See also

- The topic of instruction combining is very wide, and there are loads and loads of possibilities. Similar to the instruction combining function is the instruction simplify function, where we simplify complicated instructions but don't necessarily reduce the number of instructions, as is the case with instruction combining. To look more deeply into this, go through the code in the `lib/Transforms/InstCombine` folder

Transforming and optimizing loops

In this recipe, we will see how we can transform and optimize loops to get shorter execution times. We will mainly be looking into the **Loop-Invariant Code Motion** (**LICM**) optimization technique, and see how it works and how it transforms the code. We will also look at a relatively simpler technique called **loop deletion**, where we eliminate loops with non-infinite, computable trip counts that have no side effects on a function's return value.

Getting ready

You must have the opt tool built for this recipe.

How to do it...

1. Write the test cases for the LICM pass:

```
$ cat testlicm.ll
define void @testfunc(i32 %i) {
; <label>:0
  br label %Loop
Loop:        ; preds = %Loop, %0
  %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]        ; <i32>
[#uses=1]
  %i2 = mul i32 %i, 17        ; <i32> [#uses=1]
  %Next = add i32 %j, %i2        ; <i32> [#uses=2]
  %cond = icmp eq i32 %Next, 0        ; <i1> [#uses=1]
  br i1 %cond, label %Out, label %Loop
Out:        ; preds = %Loop
  ret void
}
```

2. Execute the LICM pass on this test code:

```
$ opt licmtest.ll -licm -S
; ModuleID = 'licmtest.ll'

define void @testfunc(i32 %i) {
  %i2 = mul i32 %i, 17
  br label %Loop

Loop:                                             ; preds =
%Loop, %0
  %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]
  %Next = add i32 %j, %i2
  %cond = icmp eq i32 %Next, 0
  br i1 %cond, label %Out, label %Loop

Out:                                              ; preds =
%Loop
  ret void
}
```

3. Write the test code for the loop deletion pass:

```
$ cat deletetest.ll
define void @foo(i64 %n, i64 %m) nounwind {
entry:
  br label %bb

bb:
  %x.0 = phi i64 [ 0, %entry ], [ %t0, %bb2 ]
  %t0 = add i64 %x.0, 1
  %t1 = icmp slt i64 %x.0, %n
  br i1 %t1, label %bb2, label %return
bb2:
  %t2 = icmp slt i64 %x.0, %m
  br i1 %t1, label %bb, label %return

return:
  ret void
}
```

4. Finally, run the loop deletion pass over the test code:

```
$ opt deletetest.ll -loop-deletion -S
; ModuleID = "deletetest.ll'

; Function Attrs: nounwind
define void @foo(i64 %n, i64 %m) #0 {
entry:
  br label %return

return:                                           ; preds =
%entry
  ret void
}

attributes #0 = { nounwind }
```

How it works...

The LICM pass performs loop-invariant code motion; it tries to move the code that is not modified in the loop out of the loop. It can go either above the loop in the pre-header block, or after the loop exits from the exit block.

In the example shown earlier, we saw the `%i2 = mul i32 %i, 17` part of the code being moved above the loop, as it is not getting modified within the loop block shown in that example.

The loop deletion pass looks out for loops with non-infinite trip counts that have no effect on the return value of the function.

In the test code, we saw how both the basic blocks `bb:` and `bb2:`, which have the loop part, get deleted. We also saw how the `foo` function directly branches to the return statement.

There are many other techniques for optimizing loops, such as `loop-rotate`, `loop-unswitch`, and `loop-unroll`, which you can try yourself. You will then see how they affect the code.

Reassociating expressions

In this recipe, you will learn about reassociating expressions and how it helps in optimization.

Getting Ready

The opt tool should be installed for this recipe to work.

How to do it...

1. Write the test case for a simple reassociate transformation:

```
$ cat testreassociate.ll
define i32 @test(i32 %b, i32 %a) {
  %tmp.1 = add i32 %a, 1234
  %tmp.2 = add i32 %b, %tmp.1
  %tmp.4 = xor i32 %a, -1
  ; (b+(a+1234))+~a -> b+1233
  %tmp.5 = add i32 %tmp.2, %tmp.4
  ret i32 %tmp.5
}
```

2. Run the reassociate pass on this test case to see how the code is modified:

```
$ opt testreassociate.ll  -reassociate -die -S
define i32 @test(i32 %b, i32 %a) {
%tmp.5 = add i32 %b, 1233
ret i32 %tmp.5
}
```

How it works ...

By reassociation, we mean applying algebraic properties such as associativity, commutativity, and distributivity to rearrange an expression to enable other optimizations, such as constant folding, LICM, and so on.

In the preceding example, we used the inverse property to eliminate patterns such as `"X + ~X" -> "-1"` using reassociation.

The first three lines of the test case give us the expression of the form `(b+(a+1234))+~a`. In this expression, using the reassociate pass, we transform `a+~a` to `-1`. Hence, in the result, we get the final return value as `b+1234-1 = b+1233`.

The code that handles this transformation is in the `Reassociate.cpp` file, located under `lib/Transforms/Scalar`.

If you look into this file, specifically the code segment, you can see that it checks whether there are `a` and `~a` in the operand list:

```
if (!BinaryOperator::isNeg(TheOp) && !BinaryOperator::isNot(TheOp))
      continue;

    Value *X = nullptr;
    …
    …
    else if (BinaryOperator::isNot(TheOp))
      X = BinaryOperator::getNotArgument(TheOp);

unsigned FoundX = FindInOperandList(Ops, i, X);
```

The following code is responsible for handling and inserting the `-1` value when it gets such values in the expression:

```
if (BinaryOperator::isNot(TheOp)) {
      Value *V = Constant::getAllOnesValue(X->getType());
      Ops.insert(Ops.end(), ValueEntry(getRank(V), V));
      e += 1;
    }
```

Vectorizing IR

Vectorization is an important optimization for compilers where we can vectorize code to execute an instruction on multiple datasets in one go. If the backend architecture supports vector registers, a broad range of data can be loaded into those vector registers, and special vector instructions can be executed on the registers.

There are two types of vectorization in LLVM—**Superword-Level Parallelism** (**SLP**) and **loop vectorization**. Loop vectorization deals with vectorization opportunities in a loop, while SLP vectorization deals with vectorizing straight-line code in a basic block. In this recipe, we will see how straight-line code is vectorized.

Getting ready

SLP vectorization constructs a bottom-up tree of the IR expression, and broadly compares the nodes of the tree to see whether they are similar and hence can be combined to form vectors. The file to be modified is `lib/Transform/Vectorize/SLPVectorizer.cpp`.

We will try to vectorize a piece of straight-line code, such as `return a[0] + a[1] + a[2] + a[3]`.

The expression tree for the preceding type of code will be a somewhat one-sided tree. We will run a DFS to store the operands and the operators.

The IR for the preceding kind of expression will look like this:

```
define i32 @hadd(i32* %a) {
entry:
    %0 = load i32* %a, align 4
    %arrayidx1 = getelementptr inbounds i32* %a, i32 1
    %1 = load i32* %arrayidx1, align 4
    %add = add nsw i32 %0, %1
    %arrayidx2 = getelementptr inbounds i32* %a, i32 2
    %2 = load i32* %arrayidx2, align 4
    %add3 = add nsw i32 %add, %2
    %arrayidx4 = getelementptr inbounds i32* %a, i32 3
    %3 = load i32* %arrayidx4, align 4
    %add5 = add nsw i32 %add3, %3
    ret i32 %add5
}
```

The vectorization model follows three steps:

1. Checking whether it's legal to vectorize.
2. Calculating the profitability of the vectorized code over the scalarized code.
3. Vectorizing the code if these two conditions are satisfied.

How to do it...

1. Open the `SLPVectorizer.cpp` file. A new function needs to be implemented for DFS traversal of the expression tree for the IR shown in the *Getting ready* section:

```
bool matchFlatReduction(PHINode *Phi, BinaryOperator *B,
const DataLayout *DL) {

  if (!B)
    return false;

  if (B->getType()->isVectorTy() ||
    !B->getType()->isIntegerTy())
    return false;

ReductionOpcode = B->getOpcode();
ReducedValueOpcode = 0;
ReduxWidth = MinVecRegSize / DL->getTypeAllocSizeInBits(B-
  >getType());
```

```
ReductionRoot = B;
ReductionPHI = Phi;

if (ReduxWidth < 4)
  return false;
if (ReductionOpcode != Instruction::Add)
  return false;

SmallVector<BinaryOperator *, 32> Stack;
ReductionOps.push_back(B);
ReductionOpcode = B->getOpcode();
Stack.push_back(B);

// Traversal of the tree.
while (!Stack.empty()) {
  BinaryOperator *Bin = Stack.back();
  if (Bin->getParent() != B->getParent())
    return false;
  Value *Op0 = Bin->getOperand(0);

  Value *Op1 = Bin->getOperand(1);
  if (!Op0->hasOneUse() || !Op1->hasOneUse())
    return false;
  BinaryOperator *Op0Bin = dyn_cast<BinaryOperator>(Op0);
  BinaryOperator *Op1Bin = dyn_cast<BinaryOperator>(Op1);
  Stack.pop_back();

  // Do not handle case where both the operands are binary
//operators
  if (Op0Bin && Op1Bin)
    return false;
  // Both the operands are not binary operator.
  if (!Op0Bin && !Op1Bin) {
    ReducedVals.push_back(Op1);
    ReducedVals.push_back(Op0);

    ReductionOps.push_back(Bin);
    continue;
}

// One of the Operand is binary operand, push that into stack

// for further processing. Push the other non-binary operand //
into ReducedVals.
  if (Op0Bin) {
```

```
    if (Op0Bin->getOpcode() != ReductionOpcode)
      return false;
    Stack.push_back(Op0Bin);
    ReducedVals.push_back(Op1);

    ReductionOps.push_back(Op0Bin);
  }

  if (Op1Bin) {

    if (Op1Bin->getOpcode() != ReductionOpcode)
      return false;
    Stack.push_back(Op1Bin);
    ReducedVals.push_back(Op0);
    ReductionOps.push_back(Op1Bin);
  }
}
SmallVector<Value *, 16> Temp;
// Reverse the loads from a[3], a[2], a[1], a[0]

// to a[0], a[1], a[2], a[3] for checking incremental
// consecutiveness further ahead.
while (!ReducedVals.empty())
  Temp.push_back(ReducedVals.pop_back_val());
ReducedVals.clear();
for (unsigned i = 0, e = Temp.size(); i < e; ++i)
  ReducedVals.push_back(Temp[i]);
  return true;
}
```

2. Calculate the cost of the resultant vectorized IR and conclude whether it is profitable to vectorize. In the `SLPVectorizer.cpp` file, add the following lines to the `getReductionCost()` function:

```
int HAddCost = INT_MAX;
// If horizontal addition pattern is identified, calculate
cost.

// Such horizontal additions can be modeled into
combination of

// shuffle sub-vectors and vector adds and one single
extract element

// from last resultant vector.
```

```
// e.g. a[0]+a[1]+a[2]+a[3] can be modeled as // %1 = load
<4 x> %0
// %2 = shuffle %1 <2, 3, undef, undef>
// %3 = add <4 x> %1, %2

// %4 = shuffle %3 <1, undef, undef, undef>

// %5 = add <4 x> %3, %4

// %6 = extractelement %5 <0>
if (IsHAdd) {
  unsigned VecElem = VecTy->getVectorNumElements();
  unsigned NumRedxLevel = Log2_32(VecElem);
  HAddCost = NumRedxLevel *
   (TTI->getArithmeticInstrCost(ReductionOpcode, VecTy) +
    TTI->getShuffleCost(TargetTransformInfo::
    SK_ExtractSubvector, VecTy, VecElem / 2, VecTy)) +
    TTI->getVectorInstrCost(Instruction::ExtractElement,
    VecTy, 0);
  }
```

3. In the same function, after calculating `PairwiseRdxCost` and `SplittingRdxCost`, compare them with `HAddCost`:

```
VecReduxCost = HAddCost < VecReduxCost ? HAddCost :
VecReduxCost;
```

4. In the `vectorizeChainsInBlock()` function, call the `matchFlatReduction()` function you just defined:

```
// Try to vectorize horizontal reductions feeding into a
return.
if (ReturnInst *RI = dyn_cast<ReturnInst>(it))

if (RI->getNumOperands() != 0)
if (BinaryOperator *BinOp =
   dyn_cast<BinaryOperator>(RI->getOperand(0))) {

  DEBUG(dbgs() << "SLP: Found a return to vectorize.\n");

  HorizontalReduction HorRdx;
  IsReturn = true;

  if ((HorRdx.matchFlatReduction(nullptr, BinOp, DL) &&
HorRdx.tryToReduce(R, TTI)) || tryToVectorizePair(BinOp-
>getOperand(0), BinOp->getOperand(1), R)) {
  Changed = true;
```

```
  it = BB->begin();
  e = BB->end();
  continue;

}
}
```

5. Define two global flags to keep a track of horizontal reduction, which feeds into a return:

```
static bool IsReturn = false;
static bool IsHAdd = false;
```

6. Allow the vectorization of small trees if they feed into a return. Add the following line to the `isFullyVectorizableTinyTree()` function:

```
if (VectorizableTree.size() == 1 && IsReturn && IsHAdd)
return true;
```

How it works...

Compile the LLVM project after saving the file containing the preceding code, and run the opt tool on the example IR, as follows:

1. Open the `example.ll` file and paste the following IR in it:

```
define i32 @hadd(i32* %a) {
entry:
    %0 = load i32* %a, align 4
    %arrayidx1 = getelementptr inbounds i32* %a, i32 1
    %1 = load i32* %arrayidx1, align 4
    %add = add nsw i32 %0, %1
    %arrayidx2 = getelementptr inbounds i32* %a, i32 2
    %2 = load i32* %arrayidx2, align 4
    %add3 = add nsw i32 %add, %2
    %arrayidx4 = getelementptr inbounds i32* %a, i32 3
    %3 = load i32* %arrayidx4, align 4
    %add5 = add nsw i32 %add3, %3
    ret i32 %add5
}
```

2. Run the opt tool on `example.ll`:

```
$ opt -basicaa -slp-vectorizer -mtriple=aarch64-unknown-linux-
gnu -mcpu=cortex-a57
```

The output will be vectorized code, like the following:

```
define i32 @hadd(i32* %a) {


entry:


%0 = bitcast i32* %a to <4 x i32>*

%1 = load <4 x i32>* %0, align 4 %rdx.shuf = shufflevector <4
x i32> %1, <4 x i32> undef, <4 x i32> <i32 2, i32 3, i32
undef, i32 undef>


%bin.rdx = add <4 x i32> %1,


%rdx.shuf %rdx.shuf1 = shufflevector <4 x i32>


%bin.rdx, <4 x i32> undef, <4 x i32> <i32 1, i32 undef, i32
undef, i32 undef> %bin.rdx2 = add <4 x i32> %bin.rdx,
%rdx.shuf1


%2 = extractelement <4 x i32> %bin.rdx2, i32 0


ret i32 %2


}
```

As observed, the code gets vectorized. The `matchFlatReduction()` function performs a DFS traversal of the expression and stores all the loads in `ReducedVals`, while adds are stored in `ReductionOps`. After this, the cost of horizontal vectorization is calculated in `HAddCost` and compared with scalar cost. It turns out to be profitable. Hence, it vectorizes the expression. This is handled in the `tryToReduce()` function, which is already implemented.

See also...

- For detailed vectorization concepts, refer to the paper *Loop-Aware SLP in GCC* by Ira Rosen, Dorit Nuzman, and Ayal Zaks

Other optimization passes

In this recipe, we will look at some more transformational passes, which are more like of utility passes. We will look at the `strip-debug-symbols` pass and the `prune-eh` pass.

Getting ready...

The opt tool must be installed.

How to do it...

1. Write a test case for checking the strip-debug pass, which strips off the debug symbols from the test code:

```
$ cat teststripdebug.ll
@x = common global i32 0                          ; <i32*>
[#uses=0]

define void @foo() nounwind readnone optsize ssp {
entry:
  tail call void @llvm.dbg.value(metadata i32 0, i64 0,
metadata !5, metadata !{}), !dbg !10
  ret void, !dbg !11
}

declare void @llvm.dbg.value(metadata, i64, metadata,
metadata) nounwind readnone

!llvm.dbg.cu = !{!2}
!llvm.module.flags = !{!13}
!llvm.dbg.sp = !{!0}
!llvm.dbg.lv.foo = !{!5}
!llvm.dbg.gv = !{!8}

!0 = !MDSubprogram(name: "foo", linkageName: "foo", line: 2,
isLocal: false, isDefinition: true, virtualIndex: 6,
isOptimized: true, file: !12, scope: !1, type: !3, function:
void ()* @foo)
!1 = !MDFile(filename: "b.c", directory: "/tmp")
```

```
!2 = !MDCompileUnit(language: DW_LANG_C89, producer: "4.2.1
(Based on Apple Inc. build 5658) (LLVM build)", isOptimized:
true, emissionKind: 0, file: !12, enums: !4, retainedTypes:
!4)

!3 = !MDSubroutineType(types: !4)

!4 = !{null}

!5 = !MDLocalVariable(tag: DW_TAG_auto_variable, name: "y",
line: 3, scope: !6, file: !1, type: !7)

!6 = distinct !MDLexicalBlock(line: 2, column: 0, file: !12,
scope: !0)

!7 = !MDBasicType(tag: DW_TAG_base_type, name: "int", size:
32, align: 32, encoding: DW_ATE_signed)

!8 = !MDGlobalVariable(name: "x", line: 1, isLocal: false,
isDefinition: true, scope: !1, file: !1, type: !7, variable:
i32* @x)

!9 = !{i32 0}

!10 = !MDLocation(line: 3, scope: !6)

!11 = !MDLocation(line: 4, scope: !6)

!12 = !MDFile(filename: "b.c", directory: "/tmp")

!13 = !{i32 1, !"Debug Info Version", i32 3}
```

2. Run the `strip-debug-symbols` pass by passing the `-strip-debug` command-line option to the `opt` tool:

```
$ opt -strip-debug teststripdebug.ll  -S

; ModuleID = ' teststripdebug.ll'


@x = common global i32 0


; Function Attrs: nounwind optsize readnone ssp

define void @foo() #0 {

entry:

  ret void

}


attributes #0 = { nounwind optsize readnone ssp }


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


!0 = metadata !{i32 1, metadata !"Debug Info Version", i32 2}
```

3. Write a test case for checking the `prune-eh` pass:

```
$ cat simpletest.ll
declare void @nounwind() nounwind

define internal void @foo() {
  call void @nounwind()
  ret void
}

define i32 @caller() {
  invoke void @foo( )
        to label %Normal unwind label %Except

Normal:        ; preds = %0
  ret i32 0

Except:        ; preds = %0
    landingpad { i8*, i32 } personality i32 (...)*
@__gxx_personality_v0
                catch i8* null
  ret i32 1
}
declare i32 @__gxx_personality_v0(...)
```

4. Run the pass to remove unused exception information by passing the `-prune-eh` command-line option to the opt tool:

```
$ opt -prune-eh -S simpletest.ll
; ModuleID = 'simpletest.ll'

; Function Attrs: nounwind
declare void @nounwind() #0

; Function Attrs: nounwind
define internal void @foo() #0 {
  call void @nounwind()
  ret void
}
```

```
; Function Attrs: nounwind
define i32 @caller() #0 {
  call void @foo()
  br label %Normal

Normal:                                           ; preds = %0
  ret i32 0
}

declare i32 @__gxx_personality_v0(...)

attributes #0 = { nounwind }
```

How it works...

In the first case, where we are running the `strip-debug` pass, it removes the debug information from the code, and we can get compact code. This pass must be used only when we are looking for compact code, as it can delete the names of virtual registers and the symbols for internal global variables and functions, thus making the source code less readable and making it difficult to reverse engineer the code.

The part of code that handles this transformation is located in the `llvm/lib/Transforms/IPO/StripSymbols.cpp` file, where the `StripDeadDebugInfo::runOnModule` function is responsible for stripping the debug information.

The second test is for removing unused exception information using the `prune-eh` pass, which implements an interprocedural pass. This walks the call-graph, turning invoke instructions into call instructions only if the callee cannot throw an exception, and marking functions as `nounwind` if they cannot throw the exceptions.

See also

- Refer to `http://llvm.org/docs/Passes.html#transform-passes` for other transformation passes

6
Target-independent Code Generator

In this chapter, we will cover the following recipes:

- The life of an LLVM IR instruction
- Visualizing the LLVM IR CFG using GraphViz
- Describing the target using TableGen
- Defining an instruction set
- Adding a machine code descriptor
- Implementing the MachineInstrBuilder class
- Implementing the MachineBasicBlock class
- Implementing the MachineFunction class
- Writing an instruction selector
- Legalizing SelectionDAG
- Optimizing SelectionDAG
- Selecting instructions from the DAG
- Scheduling instructions in SelectionDAG

Introduction

After optimizing the LLVM IR, it needs to be converted into machine instructions for execution. The machine-independent code generator interface gives an abstract layer that helps convert IR into machine instructions. In this phase, the IR is converted into SelectionDAG (**DAG** stands for **Directed Acyclic Graph**). Various phases work on the nodes of SelectionDAG. This chapter describes the important phases in target-independent code generation.

The life of an LLVM IR instruction

In previous chapters, we saw how high-level language instructions, statements, logical blocks, function calls, loops, and so on get transformed into the LLVM IR. Various optimization passes then process the IR to make it more optimal. The IR generated is in the SSA form and, in abstract format, almost independent of any high- or low-level language constraints, which facilitates optimization passes running on it. There might be some optimizations that are target-specific and take place later, when the IR gets converted into machine instructions.

After we get an optimal LLVM IR, the next phase is to convert it into target-machine-specific instructions. LLVM uses the SelectionDAG approach to convert the IR into machine instructions. The Linear IR is converted into SelectionDAG, a DAG that represents instructions as nodes. The SDAG then goes through various phases:

- The SelectionDAG is created out of LLVM IR
- Legalizing SDAG nodes
- DAG combine optimization
- Instruction selection from the target instruction
- Scheduling and emitting a machine instruction
- Register allocation—SSA destruction, register assignment, and register spilling
- Emitting code

All the preceding stages are modularized in LLVM.

C Code to LLVM IR

The first step is to convert the front end language example to LLVM IR. Let's take an example:

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

Its LLVM IR will be as follows:

```
define i32 @test(i32 %a, i32 %b, i32 %c) {
      %add = add nsw i32 %a, %b
      %div = sdiv i32 %add, %c
      return i32 %div
    }
```

IR optimization

The IR then goes through various optimization passes, as described in previous chapters. The IR, in the transformation phase, goes through the `InstCombiner::visitSDiv()` function in the `InstCombine` pass. In that function, it also goes through the `SimplifySDivInst()` function and tries to check whether an opportunity exists to further simplify the instruction.

LLVM IR to SelectionDAG

After the IR transformations and optimizations are over, the LLVM IR instruction passes through a **Selection DAG node** incarnation. Selection DAG nodes are created by the `SelectionDAGBuilder` class. The `SelectionDAGBuilder::visit()` function call from the `SelectionDAGISel` class visits each IR instruction for creating an `SDAGNode` node. The method that handles an `SDiv` instruction is `SelectionDAGBuilder::visitSDiv`. It requests a new `SDNode` node from the DAG with the`ISD::SDIV` opcode, which then becomes a node in the DAG.

SelectionDAG legalization

The `SelectionDAG` node created may not be supported by the target architecture. In the initial phase of Selection DAG, these unsupported nodes are called *illegal*. Before the `SelectionDAG` machinery actually emits machine instructions from the DAG nodes, these undergo a few other transformations, legalization being one of the important phases.

The legalization of `SDNode` involves type and operation legalization. The target-specific information is conveyed to the target-independent algorithms via an interface called `TargetLowering`. This interface is implemented by the target and, describes how LLVM IR instructions should be lowered to legal `SelectionDAG` operations. For instance, x86 lowering is implemented in the `X86TargetLowering` interface. The `setOperationAction()` function specifies whether the ISD node needs to be expanded or customized by operation legalization. When `SelectionDAGLegalize::LegalizeOp` sees the expand flag, it replaces the `SDNode` node with the parameter specified in the `setOperationAction()` call.

Conversion from target-independent DAG to machine DAG

Now that we have legalized the instruction, `SDNode` should be converted to `MachineSDNode`. The machine instructions are described in a generic table-based fashion in the target description `.td` files. Using `tablegen`, these files are then converted into `.inc` files that have registers/instructions as enums to refer to in the C++ code. Instructions can be selected by an automated selector, `SelectCode`, or they can be handled specifically by writing a customized `Select` function in the `SelectionDAGISel` class. The DAG node created at this step is a `MachineSDNode` node, a subclass of `SDNode` that holds the information required to construct an actual machine instruction but is still in the DAG node form.

Scheduling instructions

A machine executes a linear set of instructions. So far, we have had machine instructions that are still in the DAG form. To convert a DAG into a linear set of instructions, a topological sort of the DAG can yield the instructions in linear order. However, the linear set of instructions generated might not result in the most optimized code, and may cause execution delays due to dependencies among instructions, register pressure, and pipeline stalling issues. Therein comes the concept of scheduling instructions. Since each target has its own set of registers and customized pipelining of the instructions, each target has its own hook for scheduling and calculating heuristics to produce optimized, faster code. After calculating the best possible way to arrange instructions, the scheduler emits the machine instructions in the machine basic block, and finally destroys the DAG.

Register allocation

The registers allocated are virtual registers after the machine instructions are emitted. Practically, an infinite number of virtual registers can be allocated, but the actual target has a limited number of registers. These limited registers need to be allocated efficiently. If this is not done, some registers have to be spilled onto the memory, and this may result in redundant load/store operations. This will also result in wastage of CPU cycles, slowing down the execution as well as increasing the memory footprint.

There are various register allocation algorithms. An important analysis is done when allocating registers—liveness of variables and live interval analysis. If two variables live in the same interval (that is, if there exists an interval interference), then they cannot be allocated the same register. An interference graph is created by analyzing liveness, and a graph coloring algorithm can be used to allocate the registers. This algorithm, however, takes quadratic time to run. Hence, it may result in longer compilation time.

LLVM employs a greedy approach for register allocation, where variables that have large live ranges are allocated registers first. Small ranges fit into the gaps of registers available, resulting in less spill weight. Spilling is a load-store operation that occurs because no registers are available to be allocated. Spill weight is the cost of operations involved in the spilling. Sometimes, live range splitting also takes place to accommodate variables into the registers.

Note that the instructions are in the SSA form before register allocation. Now, the SSA form cannot exist in the real world because of the limited number of registers available. In some types of architecture, some instructions require fixed registers.

Code emission

Now that the original high-level code has been translated into machine instructions, the next step is to emit the code. LLVM does this in two ways; the first is JIT, which directly emits the code to the memory. The second way is by using the MC framework to emit assembly and object files for all backend targets.The `LLVMTargetMachine::addPassesToEmitFile` function is responsible for defining the sequence of actions required to emit an object file. The actual MI-to-MCInst translation is done in the `EmitInstruction` function of the `AsmPrinter` interface. The static compiler tool, llc, generates assembly instructions for a target. Object file (or assembly code) emission is done by implementing the `MCStreamer` interface.

Visualizing LLVM IR CFG using GraphViz

The LLVM IR control flow graph can be visualized using the **GraphViz** tool. It gives a visual depiction of the nodes formed and how the code flow follows in the IR generated. Since the important data structures in LLVM are graphs, this can be a very useful way to understand the IR flow when writing a custom pass or studying the behavior of the IR pattern.

Getting ready

1. To install `graphviz` on Ubuntu, first add its `ppa` repository:

   ```
   $ sudo apt-add-repository ppa:dperry/ppa-graphviz-test
   ```

2. Update the package repository:

   ```
   $ sudo apt-get update
   ```

3. Install `graphviz`:

   ```
   $ sudo apt-get install graphviz
   ```

If you get the `graphviz : Depends: libgraphviz4 (>= 2.18) but it is not going to be installed` error, run the following commands:

```
$ sudo apt-get remove libcdt4
$ sudo apt-get remove libpathplan4
```

Then install `graphviz` again with the following command:

```
$ sudo apt-get install graphviz
```

How to do it...

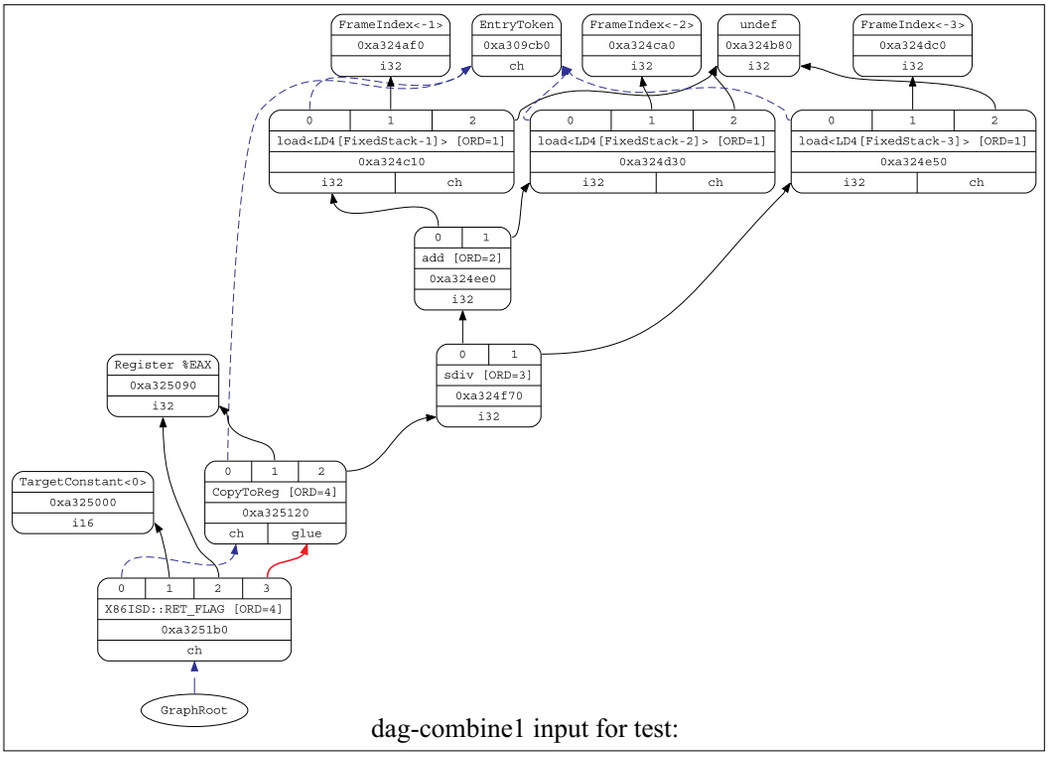
1. Once the IR has been converted to DAG, it can be viewed in different phases. Create a test.ll file with the following code:

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, %c
  ret i32 %div
}
```

2. To display the DAG after it is built, before the first optimization pass, enter the following command:

```
$ llc -view-dag-combine1-dags test.ll
```

The following diagram shows the DAG before the first optimization pass:

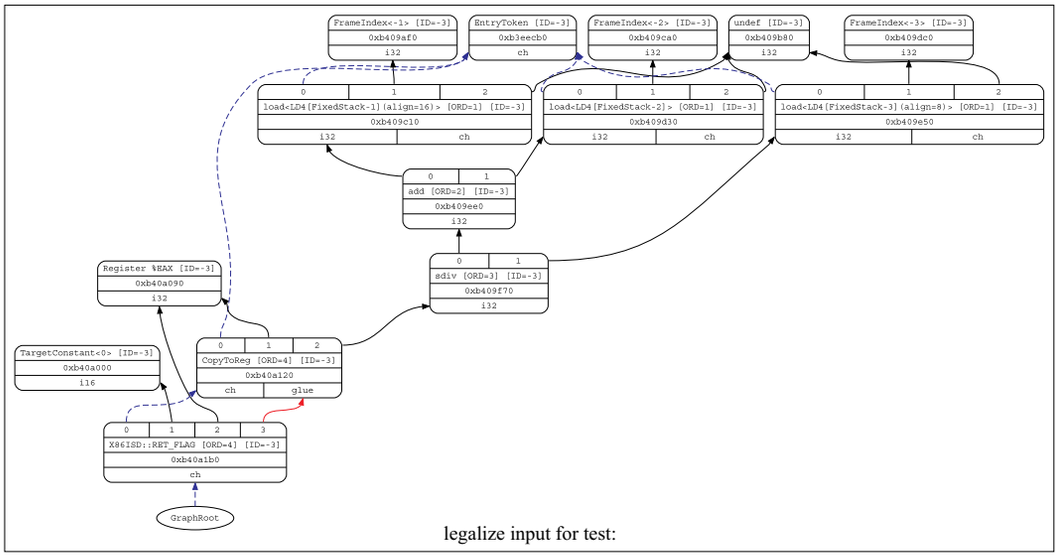


dag-combine1 input for test:

3. To display the DAG before legalization, run this command:

```
$ llc -view-legalize-dags test.ll
```

Here is a diagram that shows the DAG before the legalization phase:

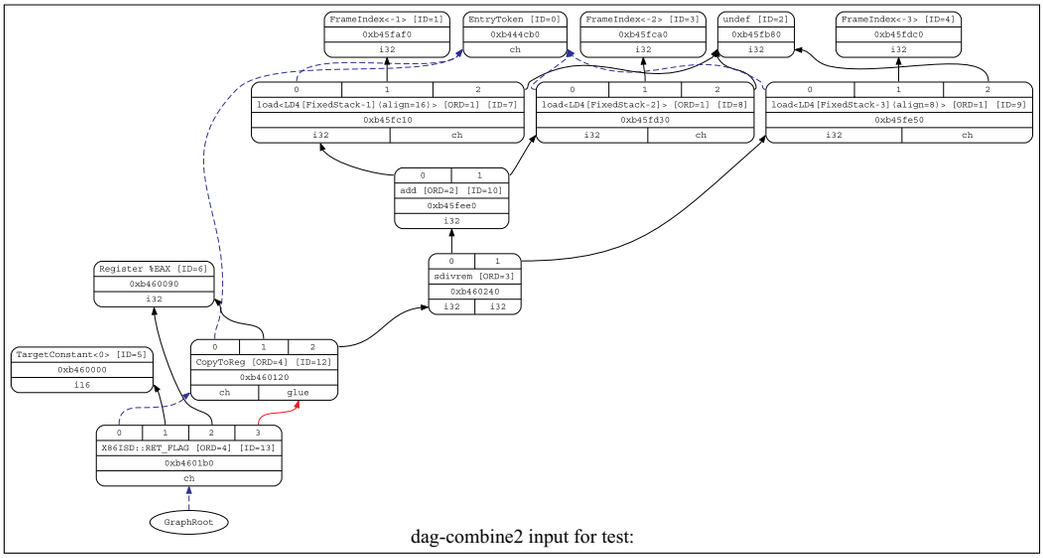


legalize input for test:

4. To display the DAG before the second optimization pass, run the following command:

```
$ llc -view-dag-combine2-dags test.ll
```

The following diagram shows the DAG before the second optimization pass:

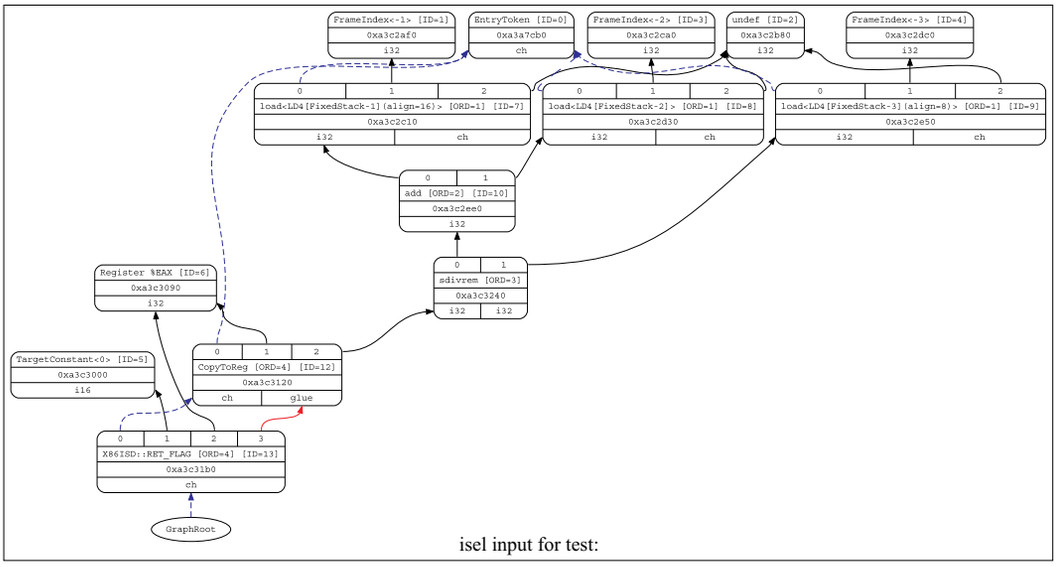


dag-combine2 input for test:

5. To display the DAG before the selection phase, enter this command:

```
$ llc -view-isel-dags test.ll
```

Here is a diagram that shows the DAG before the selection phase:

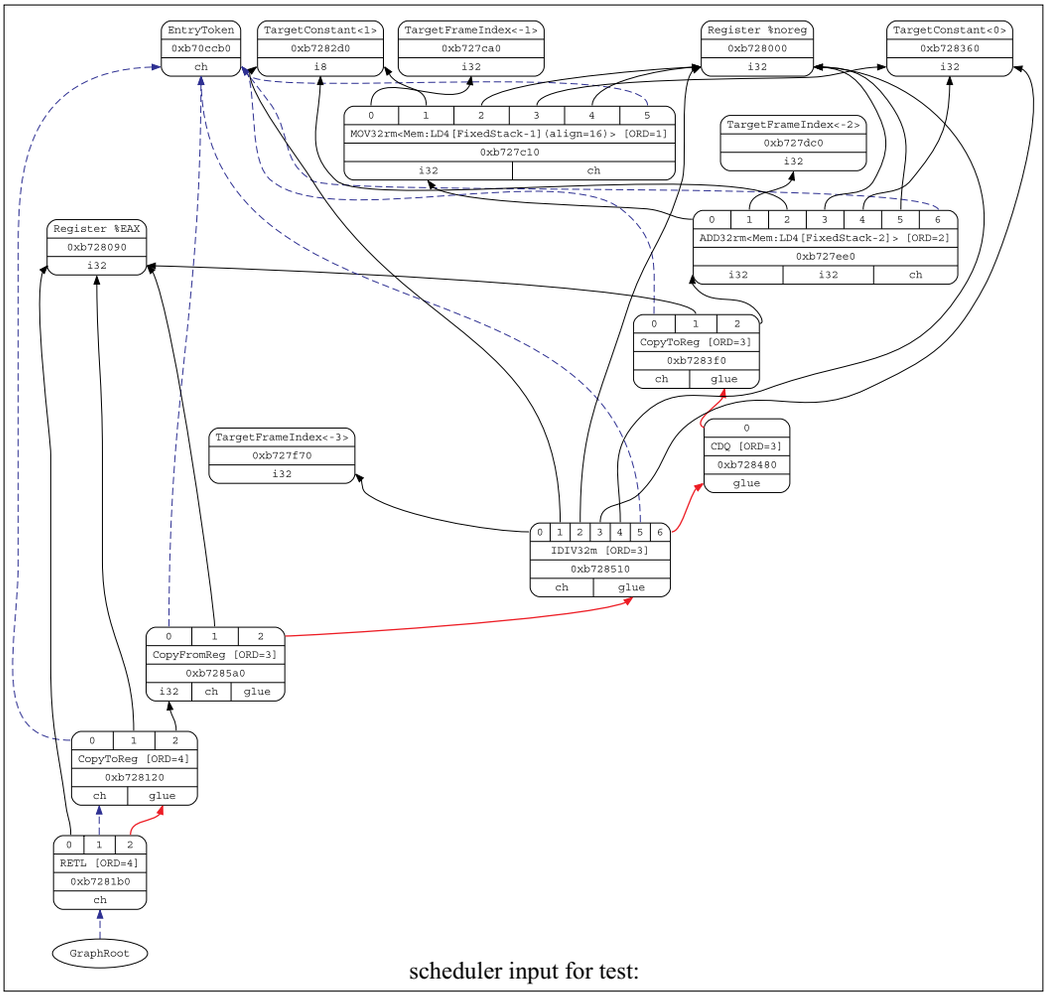


isel input for test:

6. To display the DAG before scheduling, run the following command:

```
$ llc -view-sched-dags test.ll
```

The following diagram shows the DAG before the scheduling phase:

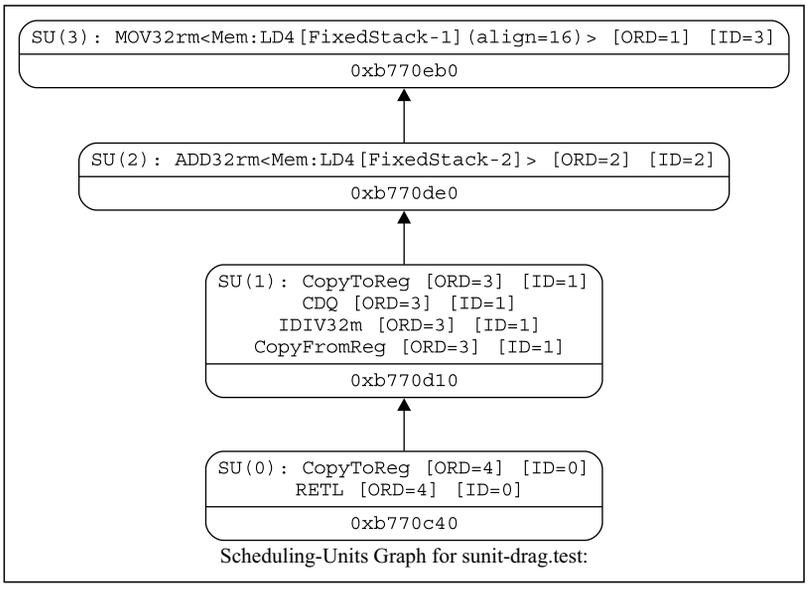


scheduler input for test:

7. To display the scheduler's dependency graph, run this command:

```
$ llc -view-sunit-dags test.ll
```

This diagram shows the scheduler's dependency graph:



Notice the difference in the DAG before and after the legalize phase. The `sdiv` node has been converted into an `sdivrem` node. The x86 target doesn't support the `sdiv` node but supports the `sdivrem` instruction. In a way, the `sdiv` instruction is illegal for the x86 target. The legalize phase converted it into an `sdivrem` instruction, which is supported by the x86 target.

Also note the difference in the DAG before and after the instruction selection (ISel) phase. Target-machine-independent instructions such as `Load` are converted into the `MOV32rm` machine code (which means, move 32-bit data from the memory to the register). The ISel phase is an important phase that will be described in later recipes.

Observe the scheduling units for the DAG. Each unit is linked to other units, which shows the dependency between them. This dependency information is very important for deciding scheduling algorithms. In the preceding case, scheduling unit 0 (SU0) is dependent on scheduling unit 1 (SU1). So, the instructions in SU0 cannot be scheduled before the instructions in SU1. SU1 is dependent on SU2, and so is SU2 on SU3.

See also

- For more details on how to view graphs in debug mode, go to `http://llvm.org/docs/ProgrammersManual.html#viewing-graphs-while-debugging-code`

Describing targets using TableGen

The target architecture can be described in terms of the registers present, the instruction set, and so on. Describing each of them manually is a tedious task. `TableGen` is a tool for backend developers that describes their target machine with a declarative language—`*.td`. The `*.td` files will be converted to enums, DAG-pattern matching functions, instruction encoding/decoding functions, and so on, which can then be used in other C++ files for coding.

To define registers and the register set in the target description's `.td` files, `tablegen` will convert the intended `.td` file into `.inc` files, which will be `#include` syntax in our `.cpp` files referring to the registers.

Getting ready

Let's assume that the sample target machine has four registers, `r0-r3`; a stack register, `sp`; and a link register, `lr`. These can be specified in the `SAMPLERegisterInfo.td` file. `TableGen` provides the `Register` class, which can be extended to specify registers.

How to do it

1. Create a new folder in `lib/Target` named `SAMPLE`:

   ```
   $ mkdir llvm_root_directory/lib/Target/SAMPLE
   ```

2. Create a new file called `SAMPLERegisterInfo.td` in the new `SAMPLE` folder:

   ```
   $ cd llvm_root_directory/lib/Target/SAMPLE
   $ vi SAMPLERegisterInfo.td
   ```

3. Define the hardware encoding, namespace, registers, and register class:

   ```
   class SAMPLEReg<bits<16> Enc, string n> : Register<n> {
     let HWEncoding = Enc;
     let Namespace = "SAMPLE";
   }

   foreach i = 0-3 in {
     def R#i : R<i, "r"#i >;
   }

   def SP  : SAMPLEReg<13, "sp">;
   def LR  : SAMPLEReg<14, "lr">;

   def GRRegs : RegisterClass<"SAMPLE", [i32], 32,
     (add R0, R1, R2, R3, SP)>;
   ```

How it works

`TableGen` processes this `.td` file to generate the `.inc` files, which have registers represented in the form of enums that can be used in the `.cpp` files. These `.inc` files will be generated when we build the LLVM project.

See also

- To get more details on how registers are defined for more advanced architecture, such as the x86, refer to the `X86RegisterInfo.td` file located at `llvm_source_code/lib/Target/X86/`

Defining an instruction set

The instruction set of an architecture varies according to various features present in the architecture. This recipe demonstrates how instruction sets are defined for the target architecture.

Getting ready

Three things are defined in the instruction target description file: operands, an assembly string, and an instruction pattern. The specification contains a list of definitions or outputs and a list of uses or inputs. There can be different operand classes such as the register class, and immediate or more complex `register + imm` operands.

Here, a simple add instruction definition is demonstrated. It takes two registers for the input and one register for the output.

How to do it...

1. Create a new file called `SAMPLEInstrInfo.td` in the `lib/Target/SAMPLE` folder:

   ```
   $ vi SAMPLEInstrInfo.td
   ```

2. Specify the operands, assembly string, and instruction pattern for the add instruction between two register operands:

   ```
   def ADDrr : InstSAMPLE<(outs GRRegs:$dst),
             (ins GRRegs:$src1, GRRegs:$src2),
              "add $dst, $src1, $src2",
              [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
   ```

How it works...

The `add` register instruction specifies `$dst` as the resultant operand, which belongs to the general register type class; the `$src1` and `$src2` inputs as two input operands, which also belong to the general register class; and the instruction assembly string as `add $dst, $src1, $src2`, which is of the 32-bit integer type.

So, an assembly will be generated for add between two registers, like this:

```
add r0, r0, r1
```

This tells us to add the `r0` and `r1` registers' content and store the result in the `r0` register.

See also

- For more detailed information on various types of instruction sets for advanced architecture, such as the x86, refer to the `X86InstrInfo.td` file located at `lib/Target/X86/`
- Detailed information of how target-specific things are defined will be covered in *Chapter 8*, *Writing an LLVM Backend*. Some concepts might get repetitive, as the preceding recipes were described in brief to get a glimpse of the target architecture description and get a foretaste of the upcoming recipes

Adding a machine code descriptor

The LLVM IR has functions, which have basic blocks. Basic blocks in turn have instructions. The next logical step is to convert those IR abstract blocks into machine-specific blocks. LLVM code is translated into a machine-specific representation formed from the `MachineFunction`, `MachineBasicBlock`, and `MachineInstr` instances. This representation contains instructions in their most abstract form—that is, having an opcode and a series of operands.

How it's done...

Now the LLVM IR instruction has to be represented in the machine instruction. Machine instructions are instances of the `MachineInstr` class. This class is an extremely abstract way of representing machine instructions. In particular, it only keeps track of an opcode number and a set of operands. The opcode number is a simple unsigned integer that has a meaning only for a specific backend.

Let's look at some important functions defined in the `MachineInstr.cpp` file:

The `MachineInstr` constructor:

```
MachineInstr::MachineInstr(MachineFunction &MF, const MCInstrDesc
&tid, const DebugLoc dl, bool NoImp)
  : MCID(&tid), Parent(nullptr), Operands(nullptr),
  NumOperands(0),
    Flags(0), AsmPrinterFlags(0),
    NumMemRefs(0), MemRefs(nullptr), debugLoc(dl) {
  // Reserve space for the expected number of operands.
  if (unsigned NumOps = MCID->getNumOperands() +
    MCID->getNumImplicitDefs() + MCID->getNumImplicitUses()) {
    CapOperands = OperandCapacity::get(NumOps);
    Operands = MF.allocateOperandArray(CapOperands);
  }

  if (!NoImp)
    addImplicitDefUseOperands(MF);
}
```

This constructor creates an object of `MachineInstr` class and adds the implicit operands. It reserves space for the number of operands specified by the `MCInstrDesc` class.

One of the important functions is `addOperand`. It adds the specified operand to the instruction. If it is an implicit operand, it is added at the end of the operand list. If it is an explicit operand, it is added at the end of the explicit operand list, as shown here:

```
void MachineInstr::addOperand(MachineFunction &MF, const
MachineOperand &Op) {
  assert(MCID && "Cannot add operands before providing an instr
descriptor");
  if (&Op >= Operands && &Op < Operands + NumOperands) {
    MachineOperand CopyOp(Op);
    return addOperand(MF, CopyOp);
  }
  unsigned OpNo = getNumOperands();
  bool isImpReg = Op.isReg() && Op.isImplicit();
  if (!isImpReg && !isInlineAsm()) {
    while (OpNo && Operands[OpNo-1].isReg() && Operands[OpNo-
1].isImplicit()) {
      --OpNo;
      assert(!Operands[OpNo].isTied() && "Cannot move tied
operands");
    }
  }
```

```
#ifndef NDEBUG
  bool isMetaDataOp = Op.getType() == MachineOperand::MO_Metadata;
  assert((isImpReg || Op.isRegMask() || MCID->isVariadic() ||
          OpNo < MCID->getNumOperands() || isMetaDataOp) &&
         "Trying to add an operand to a machine instr that is
already done!");
#endif

  MachineRegisterInfo *MRI = getRegInfo();
  OperandCapacity OldCap = CapOperands;
  MachineOperand *OldOperands = Operands;
  if (!OldOperands || OldCap.getSize() == getNumOperands()) {
    CapOperands = OldOperands ? OldCap.getNext() : OldCap.get(1);
    Operands = MF.allocateOperandArray(CapOperands);
    if (OpNo)
      moveOperands(Operands, OldOperands, OpNo, MRI);
  }
  if (OpNo != NumOperands)
    moveOperands(Operands + OpNo + 1, OldOperands + OpNo,
NumOperands - OpNo,
                 MRI);
  ++NumOperands;
  if (OldOperands != Operands && OldOperands)
    MF.deallocateOperandArray(OldCap, OldOperands);
  MachineOperand *NewMO = new (Operands + OpNo)
MachineOperand(Op);
  NewMO->ParentMI = this;
  if (NewMO->isReg()) {
        NewMO->Contents.Reg.Prev = nullptr;
    NewMO->TiedTo = 0;
        if (MRI)
      MRI->addRegOperandToUseList(NewMO);
        if (!isImpReg) {
            if (NewMO->isUse()) {
        int DefIdx = MCID->getOperandConstraint(OpNo,
MCOI::TIED_TO);
        if (DefIdx != -1)
          tieOperands(DefIdx, OpNo);
      }
           if (MCID->getOperandConstraint(OpNo,
MCOI::EARLY_CLOBBER) != -1)
        NewMO->setIsEarlyClobber(true);
    }
  }
}
```

The target architecture has some memory operands as well. To add those memory operands, a function called `addMemOperands()` is defined:

```
void MachineInstr::addMemOperand(MachineFunction &MF,
                                 MachineMemOperand *MO) {
  mmo_iterator OldMemRefs = MemRefs;
  unsigned OldNumMemRefs = NumMemRefs;
  unsigned NewNum = NumMemRefs + 1;
  mmo_iterator NewMemRefs = MF.allocateMemRefsArray(NewNum);
  std::copy(OldMemRefs, OldMemRefs + OldNumMemRefs, NewMemRefs);
  NewMemRefs[NewNum - 1] = MO;
  setMemRefs(NewMemRefs, NewMemRefs + NewNum);
}
```

The `setMemRefs()` function is the primary method for setting up a `MachineInstr MemRefs` list.

How it works...

The `MachineInstr` class has an MCID member, with the `MCInstrDesc` type for describing the instruction, a `uint8_t` flags member, a memory reference member (`mmo_iterator MemRefs`), and a vector member of the `std::vector<MachineOperand>` operands. In terms of methods, the `MachineInstr` class provides the following:

- A basic set of `get**` and `set**` functions for information queries, for example, `getOpcode()`, `getNumOperands()`, and so on
- Bundle-related operations, for example, `isInsideBundle()`
- Checking whether the instruction has certain properties, for example, `isVariadic()`, `isReturn()`, `isCall()`, and so on
- Machine instruction manipulation, for example, `eraseFromParent()`
- Register-related operations, such as `ubstituteRegister()`, `addRegisterKilled()`, and so on
- Machine-instruction-creating methods, for example, `addOperand()`, `setDesc()`, and so on

Note that, although the `MachineInstr` class provides machine-instruction-creating methods, a dedicated function called `BuildMI()`, based on the `MachineInstrBuilder` class, is more convenient.

Implementing the MachineInstrBuilder class

The `MachineInstrBuilder` class exposes a function called `BuildMI()`. This function is used to build machine instructions.

How to do it...

Machine instructions are created by using the `BuildMI` functions, located in the `include/llvm/CodeGen/MachineInstrBuilder.h` file. The `BuildMI` functions make it easy to build arbitrary machine instructions.

For example, you can use `BuildMI` in code snippets for the following purposes:

1. To create a `DestReg = mov 42` (rendered in the x86 assembly as `mov DestReg, 42`) instruction:

```
MachineInstr *MI = BuildMI(X86::MOV32ri, 1, DestReg).addImm(42);
```

2. To create the same instruction, but insert it at the end of a basic block:

```
MachineBasicBlock &MBB =
BuildMI(MBB, X86::MOV32ri, 1, DestReg).addImm(42);
```

3. To create the same instruction, but insert it before a specified iterator point:

```
MachineBasicBlock::iterator MBBI =
BuildMI(MBB, MBBI, X86::MOV32ri, 1, DestReg).addImm(42)
```

4. To create a self-looping branch instruction:

```
BuildMI(MBB, X86::JNE, 1).addMBB(&MBB);
```

How it works...

The `BuildMI()` function is required for specifying the number of operands that the machine instruction will take, which facilitates efficient memory allocation. It is also required to specify whether operands use values or definitions.

Implementing the MachineBasicBlock class

Similar to basic blocks in the LLVM IR, a `MachineBasicBlock` class has a set of machine instructions in sequential order. Mostly, a `MachineBasicBlock` class maps to a single LLVM IR basic block. However, there can be cases where multiple `MachineBasicBlocks` classes map to a single LLVM IR basic block. The `MachineBasicBlock` class has a method, called `getBasicBlock()`, that returns the IR basic block to which it is mapping.

How to do it...

The following steps show how machine basic blocks are added:

1. The `getBasicBlock` method will return only the current basic block:

```
const BasicBlock *getBasicBlock() const { return BB; }
```

2. The basic blocks have successor as well as predecessor basic blocks. To keep track of those, vectors are defined as follows:

```
std::vector<MachineBasicBlock *> Predecessors;
std::vector<MachineBasicBlock *> Successors;
```

3. An `insert` function should be added to insert a machine instruction into the basic block:

```
  MachineBasicBlock::insert(instr_iterator I, MachineInstr
*MI) {
assert(!MI->isBundledWithPred() && !MI->isBundledWithSucc()
&& "Cannot insert instruction with bundle flags");

if (I != instr_end() && I->isBundledWithPred()) {
    MI->setFlag(MachineInstr::BundledPred);
    MI->setFlag(MachineInstr::BundledSucc);
}
    return Insts.insert(I, MI);
}
```

4. A function called `SplitCriticalEdge()` splits the critical edges from this block to the given successor block, and returns the newly created block, or null if splitting is not possible. This function updates the `LiveVariables`, `MachineDominatorTree`, and `MachineLoopInfo` classes:

```
MachineBasicBlock *
MachineBasicBlock::SplitCriticalEdge(MachineBasicBlock
*Succ, Pass *P) {
…
…
…
}
```

The full implementation of the preceding code is shown in the `MachineBasicBlock.cpp` file located at `lib/CodeGen/`.

How it works...

As listed previously, several representative functions of different categories form the interface definition of the `MachineBasicBlock` class. The `MachineBasicBlock` class keeps a list of machine instructions such as `typedef ilist<MachineInstr>` instructions, instructions `Insts`, and the original LLVM BB (basic block). It also provides methods for purposes such as these:

- BB information querying (for example, `getBasicBlock()` and `setHasAddressTaken()`)
- BB-level manipulation (for example, `moveBefore()`, `moveAfter()`, and `addSuccessor()`)
- Instruction-level manipulation (for example, `push_back()`, `insertAfter()`, and so on)

See also

- To see a detailed implementation of the `MachineBasicBlock class`, go through the `MachineBasicBlock.cpp` file located at `lib/CodeGen/`

Implementing the MachineFunction class

Similar to the LLVM IR `FunctionBlock` class, a `MachineFunction` class contains a series of `MachineBasicBlocks` classes. These `MachineFunction` classes map to LLVM IR functions that are given as input to the instruction selector. In addition to a list of basic blocks, the `MachineFunction` class contains the `MachineConstantPool`, `MachineFrameInfo`, `MachineFunctionInfo`, and `MachineRegisterInfo` classes.

How to do it...

Many functions are defined in the `MachineFunction` class, which does specific tasks. There are also many class member objects that keep information, such as the following:

- `RegInfo` keeps information about each register that is in use in the function:
  ```
  MachineRegisterInfo *RegInfo;
  ```
- `MachineFrameInfo` keeps track of objects allocated on the stack:
  ```
  MachineFrameInfo *FrameInfo;
  ```
- `ConstantPool` keeps track of constants that have been spilled to the memory:
  ```
  MachineConstantPool *ConstantPool;
  ```

- `JumpTableInfo` keeps track of jump tables for switch instructions:

```
MachineJumpTableInfo *JumpTableInfo;
```

- The list of machine basic blocks in the function:

```
typedef ilist<MachineBasicBlock> BasicBlockListType;
BasicBlockListType BasicBlocks;
```

- The `getFunction` function returns the LLVM function that the current machine code represents:

```
const Function *getFunction() const { return Fn; }
```

- `CreateMachineInstr` allocates a new `MachineInstr` class:

```
MachineInstr *CreateMachineInstr(const MCInstrDesc &MCID,
DebugLoc DL,
bool NoImp = false);
```

How it works...

The `MachineFunction` class primarily contains a list of `MachineBasicBlock` objects (`typedef ilist<MachineBasicBlock> BasicBlockListType; BasicBlockListType BasicBlocks;`), and defines various methods for retrieving information about the machine function and manipulating the objects in the basic blocks member. A very important point to note is that the `MachineFunction` class maintains the **control flow graph** (**CFG**) of all basic blocks in a function. Control flow information in CFG is crucial for many optimizations and analyses. So, it is important to know how the `MachineFunction` objects and the corresponding CFGs are constructed.

See also

- A detailed implementation of the `MachineFunction` class can be found in the `MachineFunction.cpp` file located at `lib/Codegen/`

Writing an instruction selector

LLVM uses the `SelectionDAG` representation to represent the LLVM IR in a low-level data-dependence DAG for instruction selection. Various simplifications and target-specific optimizations can be applied to the `SelectionDAG` representation. This representation is target-independent. It is a significant, simple, and powerful representation used to implement IR lowering to target instructions.

How to do it...

The following code shows a brief skeleton of the `SelectionDAG` class, its data members, and various methods used to set/retrieve useful information from this class. The `SelectionDAG` class is defined as follows:

```
class SelectionDAG {
const TargetMachine &TM;
const TargetLowering &TLI;
const TargetSelectionDAGInfo &TSI;
MachineFunction *MF;
LLVMContext *Context;
CodeGenOpt::Level OptLevel;

SDNode EntryNode;
// Root - The root of the entire DAG.
SDValue Root;

// AllNodes - A linked list of nodes in the current DAG.
ilist<SDNode> AllNodes;

// NodeAllocatorType - The AllocatorType for allocating SDNodes.
We use

typedef RecyclingAllocator<BumpPtrAllocator, SDNode,
sizeof(LargestSDNode),
AlignOf<MostAlignedSDNode>::Alignment>
NodeAllocatorType;

BumpPtrAllocator OperandAllocator;

BumpPtrAllocator Allocator;

SDNodeOrdering *Ordering;

public:

struct DAGUpdateListener {

DAGUpdateListener *const Next;

SelectionDAG &DAG;

explicit DAGUpdateListener(SelectionDAG &D)
```

```
: Next(D.UpdateListeners), DAG(D) {
DAG.UpdateListeners = this;
}

private:

friend struct DAGUpdateListener;

DAGUpdateListener *UpdateListeners;


void init(MachineFunction &mf);

// Function to set root node of SelectionDAG
const SDValue &setRoot(SDValue N) {
  assert((!N.getNode() || N.getValueType() == MVT::Other) &&
    "DAG root value is not a chain!");
  if (N.getNode())
    checkForCycles(N.getNode());
  Root = N;
  if (N.getNode())
    checkForCycles(this);
  return Root;
}

void Combine(CombineLevel Level, AliasAnalysis &AA,
CodeGenOpt::Level OptLevel);

SDValue getConstant(uint64_t Val, EVT VT, bool isTarget = false);

SDValue getConstantFP(double Val, EVT VT, bool isTarget = false);

SDValue getGlobalAddress(const GlobalValue *GV, DebugLoc DL, EVT
VT, int64_t offset = 0, bool isTargetGA = false,
unsigned char TargetFlags = 0);

SDValue getFrameIndex(int FI, EVT VT, bool isTarget = false);

SDValue getTargetIndex(int Index, EVT VT, int64_t Offset = 0,
unsigned char TargetFlags = 0);

// Function to return Basic Block corresponding to this
MachineBasicBlock
```

```
SDValue getBasicBlock(MachineBasicBlock *MBB);

SDValue getBasicBlock(MachineBasicBlock *MBB, DebugLoc dl);

SDValue getExternalSymbol(const char *Sym, EVT VT);

SDValue getExternalSymbol(const char *Sym, DebugLoc dl, EVT VT);

SDValue getTargetExternalSymbol(const char *Sym, EVT VT,
unsigned char TargetFlags = 0);

// Return the type of the value this SelectionDAG node corresponds
// to
SDValue getValueType(EVT);

SDValue getRegister(unsigned Reg, EVT VT);

SDValue getRegisterMask(const uint32_t *RegMask);

SDValue getEHLabel(DebugLoc dl, SDValue Root, MCSymbol *Label);

SDValue getBlockAddress(const BlockAddress *BA, EVT VT,
int64_t Offset = 0, bool isTarget = false,
unsigned char TargetFlags = 0);

SDValue getSExtOrTrunc(SDValue Op, DebugLoc DL, EVT VT);

SDValue getZExtOrTrunc(SDValue Op, DebugLoc DL, EVT VT);

SDValue getZeroExtendInReg(SDValue Op, DebugLoc DL, EVT SrcTy);

SDValue getNOT(DebugLoc DL, SDValue Val, EVT VT);

// Function to get SelectionDAG node.
SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N1,
SDValue N2);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT,
SDValue N1, SDValue N2, SDValue N3);
```

```
SDValue getMemcpy(SDValue Chain, DebugLoc dl, SDValue Dst, SDValue
Src,SDValue Size, unsigned Align, bool isVol, bool AlwaysInline,
MachinePointerInfo DstPtrInfo,MachinePointerInfo SrcPtrInfo);

SDValue getAtomic(unsigned Opcode, DebugLoc dl, EVT MemVT, SDValue
Chain,
SDValue Ptr, SDValue Cmp, SDValue Swp,
MachinePointerInfo PtrInfo, unsigned Alignment,
AtomicOrdering Ordering,
SynchronizationScope SynchScope);

SDNode *UpdateNodeOperands(SDNode *N, SDValue Op);

SDNode *UpdateNodeOperands(SDNode *N, SDValue Op1, SDValue Op2);

SDNode *UpdateNodeOperands(SDNode *N, SDValue Op1, SDValue Op2,
SDValue Op3);

SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT);

SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT,
SDValue Op1);

SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT,
SDValue Op1, SDValue Op2);

MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT
VT);
MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT
VT,
SDValue Op1);

MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT
VT,
SDValue Op1, SDValue Op2);

void ReplaceAllUsesWith(SDValue From, SDValue Op);

void ReplaceAllUsesWith(SDNode *From, SDNode *To);

void ReplaceAllUsesWith(SDNode *From, const SDValue *To);

bool isBaseWithConstantOffset(SDValue Op) const;
```

```
bool isKnownNeverNaN(SDValue Op) const;

bool isKnownNeverZero(SDValue Op) const;

bool isEqualTo(SDValue A, SDValue B) const;

SDValue UnrollVectorOp(SDNode *N, unsigned ResNE = 0);

bool isConsecutiveLoad(LoadSDNode *LD, LoadSDNode *Base,

unsigned Bytes, int Dist) const;

unsigned InferPtrAlignment(SDValue Ptr) const;

private:

bool RemoveNodeFromCSEMaps(SDNode *N);

void AddModifiedNodeToCSEMaps(SDNode *N);

SDNode *FindModifiedNodeSlot(SDNode *N, SDValue Op, void
*&InsertPos);

SDNode *FindModifiedNodeSlot(SDNode *N, SDValue Op1, SDValue Op2,
void *&InsertPos);

SDNode *FindModifiedNodeSlot(SDNode *N, const SDValue *Ops,
unsigned NumOps,void *&InsertPos);

SDNode *UpdadeDebugLocOnMergedSDNode(SDNode *N, DebugLoc loc);

void DeleteNodeNotInCSEMaps(SDNode *N);

void DeallocateNode(SDNode *N);

unsigned getEVTAlignment(EVT MemoryVT) const;

void allnodes_clear();

std::vector<SDVTList> VTList;

std::vector<CondCodeSDNode*> CondCodeNodes;
```

```
std::vector<SDNode*> ValueTypeNodes;

std::map<EVT, SDNode*, EVT::compareRawBits>
ExtendedValueTypeNodes;

StringMap<SDNode*> ExternalSymbols;

std::map<std::pair<std::string, unsigned char>,SDNode*>
TargetExternalSymbols;
};
```

How it works...

From the preceding code, it can be seen that the `SelectionDAG` class provides lots of target-independent methods to create `SDNode` of various kinds, and retrieves/computes useful information from the nodes in the `SelectionDAG` graph. There are also update and replace methods provided in the `SelectionDAG` class. Most of these methods are defined in the `SelectionDAG.cpp` file. Note that the `SelectionDAG` graph and its node type, `SDNode`, are designed in a way that is capable of storing both target-independent and target-specific information. For example, the `isTargetOpcode()` and `isMachineOpcode()` methods in the `SDNode` class can be used to determine whether an opcode is a target opcode or a machine opcode (target-independent). This is because the same class type, `NodeType`, is used to represent both the opcode of a real target and the opcode of a machine instruction, but with separate ranges.

Legalizing SelectionDAG

A `SelectionDAG` representation is a target-independent representation of instructions and operands. However, a target may not always support the instruction or data type represented by `SelectionDAG`. In that sense, the initial `SelectionDAG` graph constructed can be called illegal. The DAG legalize phase converts the illegal DAG into a legal DAG supported by the target architecture.

A DAG legalize phase can follow two ways to convert unsupported data types into supported data types—by promoting smaller data types to larger data types, or by truncating larger data types into smaller ones. For example, suppose that a type of target architecture supports only i32 data types. In that case, smaller data types such as i8 and i16 need to be promoted to the i32 type. A larger data type, such as i64, can be expanded to give two i32 data types. The `Sign` and `Zero` extensions can be added so that the result remains consistent in the process of promoting or expanding data types.

Similarly, vector types can be legalized to supported vector types by either splitting the vector into smaller sized vectors (by extracting the elements from the vector), or by widening smaller vector types to larger, supported vector types. If vectors are not supported in the target architecture, then every element of the vector in the IR needs to be extracted in the scalar form.

The legalize phase can also instruct the kind of classes of registers supported for given data.

How to do it...

The `SelectionDAGLegalize` class consists of various data members, tracking data structures to keep a track of legalized nodes, and various methods that are used to operate on nodes to legalize them. A sample snapshot of the legalize phase code from the LLVM trunk shows the basic skeleton of implementation of the legalize phase, as follows:

```
namespace {
class SelectionDAGLegalize : public
SelectionDAG::DAGUpdateListener {

const TargetMachine &TM;

const TargetLowering &TLI;

SelectionDAG &DAG;

SelectionDAG::allnodes_iterator LegalizePosition;

// LegalizedNodes - The set of nodes which have already been
legalized.
SmallPtrSet<SDNode *, 16> LegalizedNodes;

public:
explicit SelectionDAGLegalize(SelectionDAG &DAG);
void LegalizeDAG();

private:

void LegalizeOp(SDNode *Node);

SDValue OptimizeFloatStore(StoreSDNode *ST);

// Legalize Load operations
void LegalizeLoadOps(SDNode *Node);

// Legalize Store operations
```

```
void LegalizeStoreOps(SDNode *Node);

// Main legalize function which operates on Selection DAG node
void SelectionDAGLegalize::LegalizeOp(SDNode *Node) {
// A target node which is constant need not be legalized further
  if (Node->getOpcode() == ISD::TargetConstant)
    return;

  for (unsigned i = 0, e = Node->getNumValues(); i != e; ++i)
    assert(TLI.getTypeAction(*DAG.getContext(), Node->getValueType(i))
== TargetLowering::TypeLegal && "Unexpected illegal type!");

  for (unsigned i = 0, e = Node->getNumOperands(); i != e; ++i)
    assert((TLI.getTypeAction(*DAG.getContext(),
    Node->getOperand(i).getValueType()) ==
TargetLowering::TypeLegal ||
      Node->getOperand(i).getOpcode() == ISD::TargetConstant) &&
      "Unexpected illegal type!");

  TargetLowering::LegalizeAction Action = TargetLowering::Legal;
  bool SimpleFinishLegalizing = true;

// Legalize based on instruction opcode
  switch (Node->getOpcode()) {
    case ISD::INTRINSIC_W_CHAIN:
    case ISD::INTRINSIC_WO_CHAIN:
    case ISD::INTRINSIC_VOID:
    case ISD::STACKSAVE:
          Action = TLI.getOperationAction(Node->getOpcode(),
MVT::Other);
    break;
…
…
}
```

How it works...

Many function members of the `SelectionDAGLegalize` class, such as `LegalizeOp`, rely on target-specific information provided by the `const TargetLowering &TLI` member (other function members may also depend on the `const TargetMachine &TM` member) in the `SelectionDAGLegalize` class. Let's take an example to demonstrate how legalization works.

There are two types of legalization: type legalization and instruction legalization. Let's first see how type legalization works. Create a `test.ll` file using the following commands:

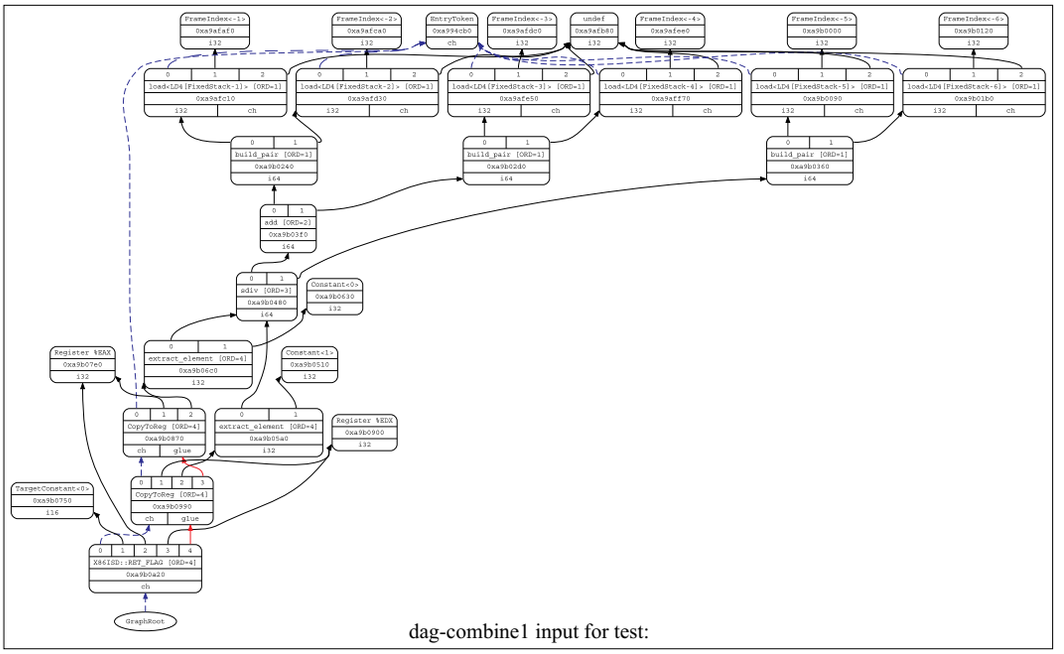
```
$ cat test.ll
define i64 @test(i64 %a, i64 %b, i64 %c) {
  %add = add nsw i64 %a, %b
  %div = sdiv i64 %add, %c
  ret i64 %div
}
```

The data type in this case is i64. For the x86 target, which supports only the 32-bit data type, the data type you just saw is illegal. To run the preceding code, the data type has to be converted to i32. This is done by the DAG Legalization phase.

To view the DAG before type legalization, run the following command line:

```
$ llc -view-dag-combine1-dags test.ll
```

The following figure shows the DAG before type legalization:

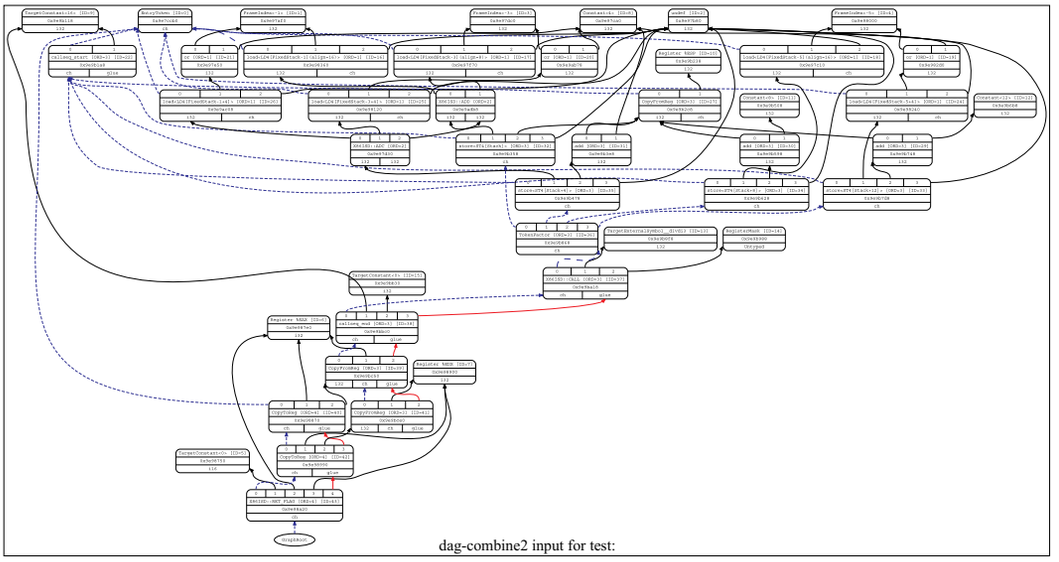


dag-combine1 input for test:

To see DAG after type legalization, enter the following command line:

```
$ llc -view-dag-combine2-dags test.ll
```

The following figure shows the DAG after type legalization:



On observing the DAG nodes carefully, you can see that every operation before legalization had the i64 type. This was because the IR had the data type i64—one-to-one mapping from the IR instruction to the DAG nodes. However, the target x86 machine supports only the i32 type (32-bit integer type). The DAG legalize phase converts unsupported i64 types to supported i32 types. This operation is called expanding—splitting larger types into smaller types. For example, in a target accepting only i32 values, all i64 values are broken down to pairs of i32 values. So, after legalization, you can see that all the operations now have i32 as the data type.

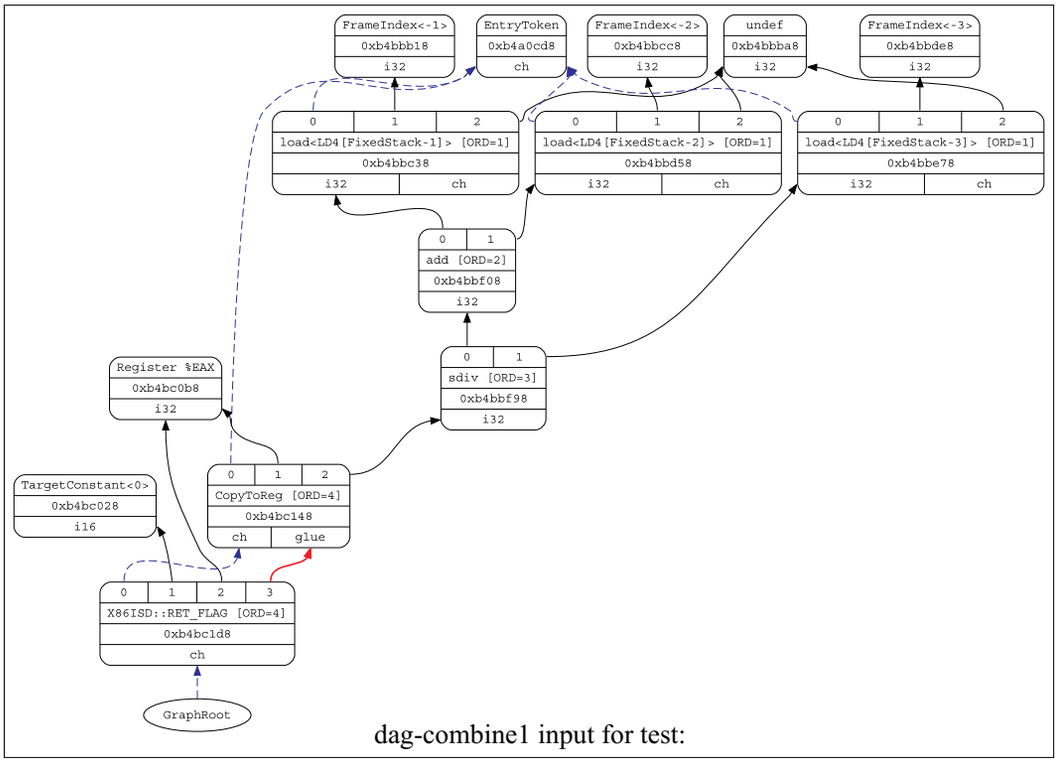
Let's see how instructions are legalized; create a `test.ll` file using the following commands:

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, %c
  ret i32 %div
}
```

To view the DAG before legalization, enter the following command:

```
$ llc –view-dag-combine1-dags test.ll
```

The following figure shows the DAG before legalization:

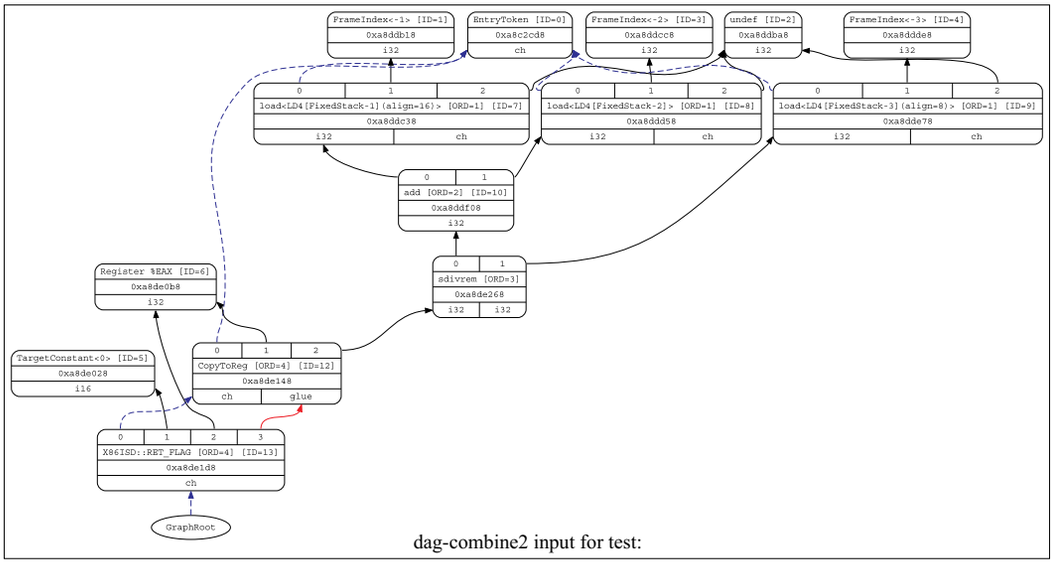


dag-combine1 input for test:

To view the DAG after legalization, enter the following command:

```
$ llc -view-dag-combine2-dags test.ll
```

The following figure shows the DAG after the legalization phase:



The DAG, before instruction legalization, consists of `sdiv` instructions. Now, the x86 target does not support the `sdiv` instruction, hence it is illegal for the target. It does, however, support the `sdivrem` instruction. So, the legalization phase involves conversion of the `sdiv` instruction to the `sdivrem` instruction, as visible in the preceding two DAGs.

Optimizing SelectionDAG

A `SelectionDAG` representation shows data and instructions in the form of nodes. Similar to the `InstCombine` pass in the LLVM IR, these nodes can be combined and optimized to form a minimized `SelectionDAG`. But, it's not just a `DAGCombine` operation that optimizes the SelectionDAG. A `DAGLegalize` phase may generate some unnecessary DAG nodes, which are cleaned up by subsequent runs of the DAG optimization pass. This finally represents the `SelectionDAG` in a more simple and elegant way.

How to do it...

There are lots and lots of function members (most of them are named like this: `visit**()`) provided in the `DAGCombiner` class to perform optimizations by folding, reordering, combining, and modifying `SDNode` nodes. Note that, from the `DAGCombiner` constructor, we can guess that some optimizations require alias analysis information:

```
class DAGCombiner {
SelectionDAG &DAG;
const TargetLowering &TLI;
CombineLevel Level;
CodeGenOpt::Level OptLevel;
bool LegalOperations;
bool LegalTypes;

SmallPtrSet<SDNode*, 64> WorkListContents;
SmallVector<SDNode*, 64> WorkListOrder;

AliasAnalysis &AA;

// Add SDnodes users to worklist
void AddUsersToWorkList(SDNode *N) {
  for (SDNode::use_iterator UI = N->use_begin(),
        UE = N->use_end(); UI != UE; ++UI)
    AddToWorkList(*UI);
}
SDValue visit(SDNode *N);

public:

void AddToWorkList(SDNode *N) {
  WorkListContents.insert(N);
  WorkListOrder.push_back(N);
}

void removeFromWorkList(SDNode *N) {
  WorkListContents.erase(N);
}

// SDnode combine operations.
SDValue CombineTo(SDNode *N, const SDValue *To, unsigned NumTo,
bool AddTo = true);

SDValue CombineTo(SDNode *N, SDValue Res, bool AddTo = true) {
  return CombineTo(N, &Res, 1, AddTo);
}

SDValue CombineTo(SDNode *N, SDValue Res0, SDValue Res1,
bool AddTo = true) {
  SDValue To[] = { Res0, Res1 };
  return CombineTo(N, To, 2, AddTo);
```

```
  }
  void CommitTargetLoweringOpt(const TargetLowering::TargetLoweringOpt
  &TLO);

  private:

  bool SimplifyDemandedBits(SDValue Op) {
    unsigned BitWidth =
  Op.getValueType().getScalarType().getSizeInBits();
    APInt Demanded = APInt::getAllOnesValue(BitWidth);
    return SimplifyDemandedBits(Op, Demanded);
  }
  bool SimplifyDemandedBits(SDValue Op, const APInt &Demanded);

  bool CombineToPreIndexedLoadStore(SDNode *N);

  bool CombineToPostIndexedLoadStore(SDNode *N);

  void ReplaceLoadWithPromotedLoad(SDNode *Load, SDNode *ExtLoad);

  SDValue PromoteOperand(SDValue Op, EVT PVT, bool &Replace);

  SDValue SExtPromoteOperand(SDValue Op, EVT PVT);

  SDValue ZExtPromoteOperand(SDValue Op, EVT PVT);

  SDValue PromoteIntBinOp(SDValue Op);

  SDValue PromoteIntShiftOp(SDValue Op);

  SDValue PromoteExtend(SDValue Op);

  bool PromoteLoad(SDValue Op);

  void ExtendSetCCUses(SmallVector<SDNode*, 4> SetCCs,
  SDValue Trunc, SDValue ExtLoad, DebugLoc DL,
  ISD::NodeType ExtType);

  SDValue combine(SDNode *N);

  // Various visit functions operating on instructions represented
  // by SD node. Similar to instruction combining at IR level.
  SDValue visitTokenFactor(SDNode *N);
```

```
SDValue visitMERGE_VALUES(SDNode *N);

SDValue visitADD(SDNode *N);
SDValue visitSUB(SDNode *N);
SDValue visitADDC(SDNode *N);
SDValue visitSUBC(SDNode *N);
SDValue visitADDE(SDNode *N);
SDValue visitSUBE(SDNode *N);
SDValue visitMUL(SDNode *N);

public:

DAGCombiner(SelectionDAG &D, AliasAnalysis &A, CodeGenOpt::Level
OL)
: DAG(D), TLI(D.getTargetLoweringInfo()),
Level(BeforeLegalizeTypes),
  OptLevel(OL), LegalOperations(false), LegalTypes(false), AA(A)
{}

// Selection DAG transformation for following ops
SDValue DAGCombiner::visitMUL(SDNode *N) {
  SDValue N0 = N->getOperand(0);
  SDValue N1 = N->getOperand(1);
  ConstantSDNode *N0C = dyn_cast<ConstantSDNode>(N0);
  ConstantSDNode *N1C = dyn_cast<ConstantSDNode>(N1);
  EVT VT = N0.getValueType();
  if (VT.isVector()) {
    SDValue FoldedVOp = SimplifyVBinOp(N);
    if (FoldedVOp.getNode()) return FoldedVOp;
  }
  if (N0.getOpcode() == ISD::UNDEF || N1.getOpcode() ==
ISD::UNDEF)
    return DAG.getConstant(0, VT);

  if (N0C && N1C)
    return DAG.FoldConstantArithmetic(ISD::MUL, VT, N0C, N1C);

  if (N0C && !N1C)
    return DAG.getNode(ISD::MUL, N->getDebugLoc(), VT, N1, N0);

  if (N1C && N1C->isNullValue())
    return N1;

  if (N1C && N1C->isAllOnesValue())
    return DAG.getNode(ISD::SUB, N->getDebugLoc(), VT,
```

```
DAG.getConstant(0, VT), N0);
  if (N1C && N1C->getAPIntValue().isPowerOf2())
    return DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N0,
           DAG.getConstant(N1C->getAPIntValue().logBase2(),
               getShiftAmountTy(N0.getValueType())));

  if (N1C && (-N1C->getAPIntValue()).isPowerOf2()) {
    unsigned Log2Val = (-N1C->getAPIntValue()).logBase2();
    return DAG.getNode(ISD::SUB, N->getDebugLoc(), VT,
DAG.getConstant(0, VT),
    DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N0,
    DAG.getConstant(Log2Val,
getShiftAmountTy(N0.getValueType()))));
  }

  if (N1C && N0.getOpcode() == ISD::SHL &&
      isa<ConstantSDNode>(N0.getOperand(1))) {
    SDValue C3 = DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N1,
N0.getOperand(1));
    AddToWorkList(C3.getNode());
    return DAG.getNode(ISD::MUL, N->getDebugLoc(), VT,
    N0.getOperand(0), C3);
  }

  if (N0.getOpcode() == ISD::SHL &&
isa<ConstantSDNode>(N0.getOperand(1)) &&
      N0.getNode()->hasOneUse()) {
    Sh = N0; Y = N1;
  } else if (N1.getOpcode() == ISD::SHL &&
isa<ConstantSDNode>(N1.getOperand(1)) &&
           N1.getNode()->hasOneUse()) {
    Sh = N1; Y = N0;
  }
  if (Sh.getNode()) {
  SDValue Mul = DAG.getNode(ISD::MUL, N->getDebugLoc(), VT,
Sh.getOperand(0), Y);
    return DAG.getNode(ISD::SHL, N->getDebugLoc(), VT,
    Mul, Sh.getOperand(1));
    }
  }
  if (N1C && N0.getOpcode() == ISD::ADD && N0.getNode()-
>hasOneUse() &&
      isa<ConstantSDNode>(N0.getOperand(1)))
    return DAG.getNode(ISD::ADD, N->getDebugLoc(), VT,
DAG.getNode(ISD::MUL, N0.getDebugLoc(),
```

```
    VT, N0.getOperand(0), N1), DAG.getNode(ISD::MUL,
N1.getDebugLoc(), VT, N0.getOperand(1), N1));

  SDValue RMUL = ReassociateOps(ISD::MUL, N->getDebugLoc(), N0,
N1);

  if (RMUL.getNode() != 0) return RMUL;
  return SDValue();
}
```

How it works...

As seen in the preceding code, some `DAGCombine` passes search for a pattern and then fold the patterns into a single DAG. This basically reduces the number of DAGs, while lowering DAGs. The result is an optimized `SelectionDAG` class.

See also

- For a more detailed implementation of the optimized `SelectionDAG` class, see the `DAGCombiner.cpp` file located at `lib/CodeGen/SelectionDAG/`

Selecting instruction from the DAG

After legalization and DAG combination, the `SelectionDAG` representation is in the optimized phase. However, the instructions represented are still target-independent and need to be mapped on target-specific instructions. The instruction selection phase takes the target-independent DAG nodes as the input, matches patterns in them, and gives the output DAG nodes, which are target-specific.

The `TableGen` DAG instruction selector generator reads the instruction patterns from the `.td` file, and automatically builds parts of the pattern matching code.

How to do it...

`SelectionDAGISel` is the common base class used for pattern-matching instruction selectors that are based on `SelectionDAG`. It inherits the `MachineFunctionPass` class. It has various functions used to determine the legality and profitability of operations such as folding. The basic skeleton of this class is as follows:

```
class SelectionDAGISel : public MachineFunctionPass {
public:
const TargetMachine &TM;
const TargetLowering &TLI;
const TargetLibraryInfo *LibInfo;
```

```
FunctionLoweringInfo *FuncInfo;
MachineFunction *MF;
MachineRegisterInfo *RegInfo;
SelectionDAG *CurDAG;
SelectionDAGBuilder *SDB;
AliasAnalysis *AA;
GCFunctionInfo *GFI;
CodeGenOpt::Level OptLevel;
static char ID;

explicit SelectionDAGISel(const TargetMachine &tm,
CodeGenOpt::Level OL = CodeGenOpt::Default);

virtual ~SelectionDAGISel();

const TargetLowering &getTargetLowering() { return TLI; }

virtual void getAnalysisUsage(AnalysisUsage &AU) const;

virtual bool runOnMachineFunction(MachineFunction &MF);

virtual void EmitFunctionEntryCode() {}

virtual void PreprocessISelDAG() {}

virtual void PostprocessISelDAG() {}

virtual SDNode *Select(SDNode *N) = 0;

virtual bool SelectInlineAsmMemoryOperand(const SDValue &Op,
char ConstraintCode,
std::vector<SDValue> &OutOps) {
  return true;
}

virtual bool IsProfitableToFold(SDValue N, SDNode *U, SDNode
*Root) const;

static bool IsLegalToFold(SDValue N, SDNode *U, SDNode *Root,
CodeGenOpt::Level OptLevel,
bool IgnoreChains = false);

enum BuiltinOpcodes {
OPC_Scope,
```

```
OPC_RecordNode,
OPC_CheckOpcode,
OPC_SwitchOpcode,
OPC_CheckFoldableChainNode,
OPC_EmitInteger,
OPC_EmitRegister,
OPC_EmitRegister2,
OPC_EmitConvertToTarget,
OPC_EmitMergeInputChains,
};

static inline int getNumFixedFromVariadicInfo(unsigned Flags) {
  return ((Flags&OPFL_VariadicInfo) >> 4)-1;
}

protected:
// DAGSize - Size of DAG being instruction selected.
unsigned DAGSize;

void ReplaceUses(SDValue F, SDValue T) {
  CurDAG->ReplaceAllUsesOfValueWith(F, T);
}

void ReplaceUses(const SDValue *F, const SDValue *T, unsigned Num)
{
  CurDAG->ReplaceAllUsesOfValuesWith(F, T, Num);
}

void ReplaceUses(SDNode *F, SDNode *T) {
  CurDAG->ReplaceAllUsesWith(F, T);
}

void SelectInlineAsmMemoryOperands(std::vector<SDValue> &Ops);

public:
bool CheckAndMask(SDValue LHS, ConstantSDNode *RHS,
int64_t DesiredMaskS) const;

bool CheckOrMask(SDValue LHS, ConstantSDNode *RHS,
int64_t DesiredMaskS) const;

virtual bool CheckPatternPredicate(unsigned PredNo) const {
  llvm_unreachable("Tblgen should generate the implementation of
this!");
```

```
}

virtual bool CheckNodePredicate(SDNode *N, unsigned PredNo) const
{
  llvm_unreachable("Tblgen should generate the implementation of
this!");
}

private:

SDNode *Select_INLINEASM(SDNode *N);

SDNode *Select_UNDEF(SDNode *N);

void CannotYetSelect(SDNode *N);

void DoInstructionSelection();

SDNode *MorphNode(SDNode *Node, unsigned TargetOpc, SDVTList VTs,
const SDValue *Ops, unsigned NumOps, unsigned EmitNodeInfo);

void PrepareEHLandingPad();

void SelectAllBasicBlocks(const Function &Fn);

bool TryToFoldFastISelLoad(const LoadInst *LI, const Instruction
*FoldInst, FastISel *FastIS);

void FinishBasicBlock();

void SelectBasicBlock(BasicBlock::const_iterator Begin,
BasicBlock::const_iterator End,
bool &HadTailCall);

void CodeGenAndEmitDAG();

void LowerArguments(const BasicBlock *BB);

void ComputeLiveOutVRegInfo();
  ScheduleDAGSDNodes *CreateScheduler();
};
```

How it works...

The instruction selection phase involves converting target-independent instructions to target-specific instructions. The `TableGen` class helps select target-specific instructions. This phase basically matches target-independent input nodes, which gives an output consisting of target-supported nodes.

The `CodeGenAndEmitDAG()` function calls the `DoInstructionSelection()` function, which visits each DAG node and calls the `Select()` function for each node, like this:

```
SDNode *ResNode = Select(Node);
```

The `Select()` function is an abstract method implemented by the targets.
The x86 target implements it in the `X86DAGToDAGISel::Select()` function. The `X86DAGToDAGISel::Select()` function intercepts some nodes for manual matching, but delegates the bulk of the work to the `X86DAGToDAGISel::SelectCode()` function.

The `X86DAGToDAGISel::SelectCod`e function is autogenerated by `TableGen`. It contains the matcher table, followed by a call to the generic `SelectionDAGISel::SelectCodeCommon()` function, passing it the table.

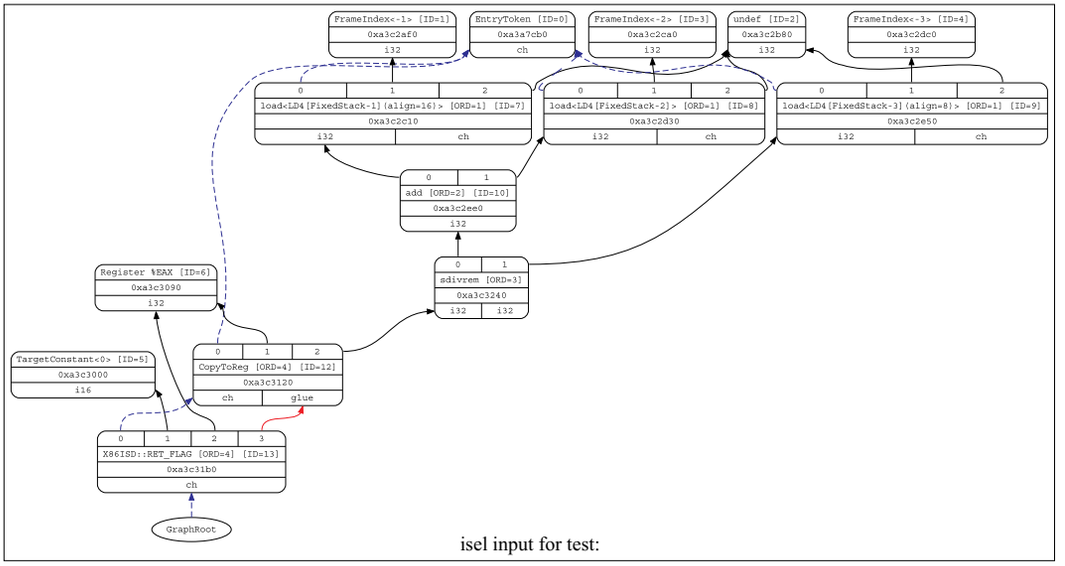
For example:

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, %c
  ret i32 %div
}
```

To see the DAG before instruction selection, enter the following command line:

```
$ llc –view-isel-dags test.ll
```

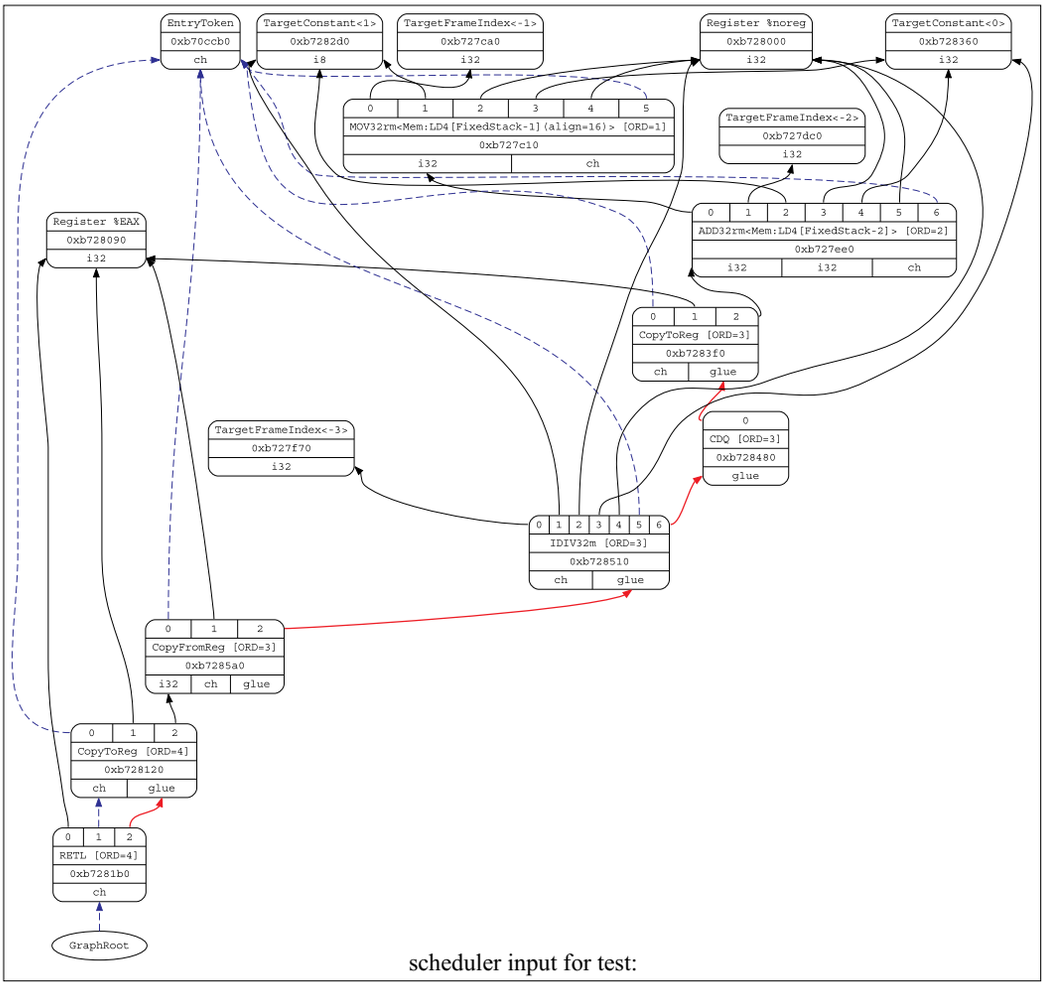
The following figure shows the DAG before the instruction selection:



To see how DAG looks like after the instruction selection, enter the following command:

```
$ llc -view-sched-dags test.ll
```

The following figure shows the DAG after the instruction selection:



scheduler input for test:

As seen, the `Load` operation is converted into the `MOV32rm` machine code by the instruction selection phase.

See also

- To see the detailed implementation of the instruction selection, take a look at the `SelectionDAGISel.cpp` file located at `lib/CodeGen/SelectionDAG/`

Scheduling instructions in SelectionDAG

So far, we have had `SelectionDAG` nodes consisting of target-supported instructions and operands. However, the code is still in DAG representation. The target architecture executes instructions in sequential form. So, the next logical step is to schedule the `SelectionDAG` nodes.

A scheduler assigns the order of execution of instructions from the DAG. In this process, it takes into account various heuristics, such as register pressure, to optimize the execution order of instructions and to minimize latencies in instruction execution. After assigning the order of execution to the DAG nodes, the nodes are converted into a list of `MachineInstrs` and the `SelectionDAG` nodes are destroyed.

How to do it...

There are several basic structures that are defined in the `ScheduleDAG.h` file and implemented in the `ScheduleDAG.cpp` file. The `ScheduleDAG` class is a base class for other schedulers to inherit, and it provides only graph-related manipulation operations such as an iterator, DFS, topological sorting, functions for moving nodes around, and so on:

```
class ScheduleDAG {
public:
  const TargetMachine &TM;              // Target processor
  const TargetInstrInfo *TII;           // Target instruction
  const TargetRegisterInfo *TRI;        // Target processor
register info
  MachineFunction &MF;                  // Machine function
  MachineRegisterInfo &MRI;             // Virtual/real
register map
  std::vector<SUnit> SUnits;            // The scheduling
units.
  SUnit EntrySU;                        // Special node for
the region entry.
  SUnit ExitSU;                         // Special node for
the region exit.

  explicit ScheduleDAG(MachineFunction &mf);

  virtual ~ScheduleDAG();

  void clearDAG();

const MCInstrDesc *getInstrDesc(const SUnit *SU) const {
  if (SU->isInstr()) return &SU->getInstr()->getDesc();
```

```
    return getNodeDesc(SU->getNode());
}

virtual void dumpNode(const SUnit *SU) const = 0;

private:

const MCInstrDesc *getNodeDesc(const SDNode *Node) const;
};

class SUnitIterator : public
std::iterator<std::forward_iterator_tag,
SUnit, ptrdiff_t> {
};

template <> struct GraphTraits<SUnit*> {
  typedef SUnit NodeType;
  typedef SUnitIterator ChildIteratorType;
  static inline NodeType *getEntryNode(SUnit *N) {
    return N;
  }
  static inline ChildIteratorType child_begin(NodeType *N) {
  return SUnitIterator::begin(N);
  }

static inline ChildIteratorType child_end(NodeType *N) {
  return SUnitIterator::end(N);
  }
};

template <> struct GraphTraits<ScheduleDAG*> : public
GraphTraits<SUnit*> {
…};

// Topological sorting of DAG to linear set of instructions
class ScheduleDAGTopologicalSort {
  std::vector<SUnit> &SUnits;
  SUnit *ExitSU;
  std::vector<int> Index2Node;
  std::vector<int> Node2Index;
  BitVector Visited;
// DFS to be run on DAG to sort topologically
  void DFS(const SUnit *SU, int UpperBound, bool& HasLoop);
```

```
    void Shift(BitVector& Visited, int LowerBound, int UpperBound);

    void Allocate(int n, int index);

  public:

    ScheduleDAGTopologicalSort(std::vector<SUnit> &SUnits, SUnit
*ExitSU);

    void InitDAGTopologicalSorting();

    bool IsReachable(const SUnit *SU, const SUnit *TargetSU);

    bool WillCreateCycle(SUnit *SU, SUnit *TargetSU);

    void AddPred(SUnit *Y, SUnit *X);

    void RemovePred(SUnit *M, SUnit *N);

    typedef std::vector<int>::iterator iterator;

    typedef std::vector<int>::const_iterator const_iterator;

    iterator begin() { return Index2Node.begin(); }

    const_iterator begin() const { return Index2Node.begin(); }

    iterator end() { return Index2Node.end();}}
```

How it works...

The scheduling algorithm implements the scheduling of instructions in the `SelectionDAG` class, which involves a variety of algorithms such as topological sorting, depth-first searching, manipulating functions, moving nodes, and iterating over a list of instructions. It takes into account various heuristics, such as register pressure, spilling cost, live interval analysis, and so on to determine the best possible scheduling of instructions.

See also

- For a detailed implementation of scheduling instructions, see the `ScheduleDAGSDNodes.cpp`, `ScheduleDAGSDNodes.h`, `ScheduleDAGRRList.cpp`, `ScheduleDAGFast.cpp`, and `ScheduleDAGVLIW.cpp` files located in the `lib/CodeGen/SelectionDAG` folder

7
Optimizing the Machine Code

In this chapter, we will cover the following recipes:

- Eliminating common subexpressions from machine code
- Analyzing live intervals
- Allocating registers
- Inserting the prologue-epilogue code
- Code emission
- Tail call optimization
- Sibling call optimization

Introduction

The machine code generated so far is yet to be assigned real target architecture registers. The registers seen so far have been virtual registers, which are infinite in number. The machine code generated is in the SSA form. However, the target registers are limited in number. Hence, register allocation algorithms require a lot of heuristic calculations to allocate registers in an optimal way.

But, before register allocation, there exists opportunities for code optimization. The machine code being in the SSA form also makes it easy to apply optimizing algorithms. The algorithms for some optimizing techniques, such as machine dead code elimination and machine common subexpression elimination, are almost the same as in the LLVM IR. The difference lies in the constraints to be checked.

Here, one of the machine code optimization techniques implemented in the LLVM trunk code repository—machine CSE— will be discussed so that you can understand how algorithms are implemented for machine code.

Eliminating common subexpression from machine code

The aim of the CSE algorithm is to eliminate common subexpressions to make machine code compact and remove unnecessary, duplicate code. Let's look at the code in the LLVM trunk to understand how it is implemented. The detailed code is in the `lib/CodeGen/MachineCSE.cpp` file.

How to do it...

1. The `MachineCSE` class runs on a machine function, and hence it should inherit the `MachineFunctionPass` class. It has various members, such as `TargetInstructionInfo`, which is used to get information about the target instruction (used in performing CSE); `TargetRegisterInfo`, which is used to get information about the target register (whether it belongs to a reserved register class, or to more such similar classes; and `MachineDominatorTree`, which is used to get information about the dominator tree for the machine block:

```
class MachineCSE : public MachineFunctionPass {
    const TargetInstrInfo *TII;
    const TargetRegisterInfo *TRI;
    AliasAnalysis *AA;
    MachineDominatorTree *DT;
    MachineRegisterInfo *MRI;
```

2. The constructor for this class is defined as follows, which initializes the pass:

```
public:
    static char ID; // Pass identification
    MachineCSE() : MachineFunctionPass(ID),
LookAheadLimit(5), CurrVN(0) {
      initializeMachineCSEPass(*PassRegistry::getPassRegistry());

    }
```

3. The `getAnalysisUsage()` function determines which passes will run before this pass to get statistics that can be used in this pass:

```
    void getAnalysisUsage(AnalysisUsage &AU) const override {
      AU.setPreservesCFG();
      MachineFunctionPass::getAnalysisUsage(AU);
      AU.addRequired<AliasAnalysis>();
```

```
    AU.addPreservedID(MachineLoopInfoID);
    AU.addRequired<MachineDominatorTree>();
    AU.addPreserved<MachineDominatorTree>();
  }
```

4. Declare some helper functions in this pass to check for simple copy propagation and trivially dead definitions, check for the liveness of physical registers and their definition uses, and so on:

```
  private:
…..
…..

bool PerformTrivialCopyPropagation(MachineInstr *MI,
                                   MachineBasicBlock *MBB);

bool isPhysDefTriviallyDead(unsigned Reg,
        MachineBasicBlock::const_iterator I,
        MachineBasicBlock::const_iterator E) const;

bool hasLivePhysRegDefUses(const MachineInstr *MI,
                    const MachineBasicBlock *MBB,
                  SmallSet<unsigned,8> &PhysRefs,
              SmallVectorImpl<unsigned> &PhysDefs,
                           bool &PhysUseDef) const;

bool PhysRegDefsReach(MachineInstr *CSMI, MachineInstr *MI,
                     SmallSet<unsigned,8> &PhysRefs,
                     SmallVectorImpl<unsigned> &PhysDefs,
                     bool &NonLocal) const;
```

5. Some more helper functions help to determine the legality and profitability of the expression being a CSE candidate:

```
    bool isCSECandidate(MachineInstr *MI);
    bool isProfitableToCSE(unsigned CSReg, unsigned Reg,
                    MachineInstr *CSMI, MachineInstr *MI);

Actual CSE performing function
    bool PerformCSE(MachineDomTreeNode *Node);
```

Let's look at the actual implementation of a CSE function:

1. The `runOnMachineFunction()` function is called first as the pass runs:

```
bool MachineCSE::runOnMachineFunction(MachineFunction &MF){
  if (skipOptnoneFunction(*MF.getFunction()))
    return false;

  TII = MF.getSubtarget().getInstrInfo();
  TRI = MF.getSubtarget().getRegisterInfo();
  MRI = &MF.getRegInfo();
  AA = &getAnalysis<AliasAnalysis>();
  DT = &getAnalysis<MachineDominatorTree>();
  return PerformCSE(DT->getRootNode());
}
```

2. The `PerformCSE()` function is called next. It takes the root node of the `DomTree`, performs a DFS walk on the `DomTree` (starting from the root node), and populates a work list consisting of the nodes of the `DomTree`. After the DFS traverses through the `DomTree`, it processes the `MachineBasicBlock` class corresponding to each node in the work list:

```
bool MachineCSE::PerformCSE(MachineDomTreeNode *Node) {
  SmallVector<MachineDomTreeNode*, 32> Scopes;
  SmallVector<MachineDomTreeNode*, 8> WorkList;
  DenseMap<MachineDomTreeNode*, unsigned> OpenChildren;

  CurrVN = 0;
// DFS to populate worklist
  WorkList.push_back(Node);
  do {
    Node = WorkList.pop_back_val();
    Scopes.push_back(Node);
    const std::vector<MachineDomTreeNode*> &Children =
Node->getChildren();
    unsigned NumChildren = Children.size();
    OpenChildren[Node] = NumChildren;
    for (unsigned i = 0; i != NumChildren; ++i) {
      MachineDomTreeNode *Child = Children[i];
      WorkList.push_back(Child);
    }
  } while (!WorkList.empty());

  // perform CSE.
  bool Changed = false;
  for (unsigned i = 0, e = Scopes.size(); i != e; ++i) {
    MachineDomTreeNode *Node = Scopes[i];
```

```
    MachineBasicBlock *MBB = Node->getBlock();
    EnterScope(MBB);
    Changed |= ProcessBlock(MBB);
    ExitScopeIfDone(Node, OpenChildren);
  }

  return Changed;
}
```

3. The next important function is the `ProcessBlock()` function, which acts on the machine basic block. The instructions in the `MachineBasicBlock` class are iterated and checked for legality and profitability if they can be a CSE candidate:

```
bool MachineCSE::ProcessBlock(MachineBasicBlock *MBB) {
  bool Changed = false;

  SmallVector<std::pair<unsigned, unsigned>, 8> CSEPairs;
  SmallVector<unsigned, 2> ImplicitDefsToUpdate;

// Iterate over each Machine instructions in the MachineBasicBlock
  for (MachineBasicBlock::iterator I = MBB->begin(), E =
MBB->end(); I != E; ) {
    MachineInstr *MI = &*I;
    ++I;

// Check if this can be a CSE candidate.
    if (!isCSECandidate(MI))
      continue;


    bool FoundCSE = VNT.count(MI);
    if (!FoundCSE) {
      // Using trivial copy propagation to find more CSE
opportunities.
      if (PerformTrivialCopyPropagation(MI, MBB)) {
        Changed = true;

        // After coalescing MI itself may become a copy.
        if (MI->isCopyLike())
          continue;

        // Try again to see if CSE is possible.
        FoundCSE = VNT.count(MI);
      }
    }
```

```
    bool Commuted = false;
    if (!FoundCSE && MI->isCommutable()) {
      MachineInstr *NewMI = TII->commuteInstruction(MI);
      if (NewMI) {
        Commuted = true;
        FoundCSE = VNT.count(NewMI);
        if (NewMI != MI) {
          // New instruction. It doesn't need to be kept.
          NewMI->eraseFromParent();
          Changed = true;
        } else if (!FoundCSE)
          // MI was changed but it didn't help, commute it
back!
          (void)TII->commuteInstruction(MI);
      }
    }

    // If the instruction defines physical registers and
the values *may* be
    // used, then it's not safe to replace it with a common
subexpression.
    // It's also not safe if the instruction uses physical
registers.
    bool CrossMBBPhysDef = false;
    SmallSet<unsigned, 8> PhysRefs;
    SmallVector<unsigned, 2> PhysDefs;
    bool PhysUseDef = false;

// Check if this instruction has been marked for CSE. Check
if it is using physical register, if yes then mark as non-
CSE candidate
 if (FoundCSE && hasLivePhysRegDefUses(MI, MBB, PhysRefs,
                                                PhysDefs,
PhysUseDef)) {
      FoundCSE = false;
…
…
    }

    if (!FoundCSE) {
      VNT.insert(MI, CurrVN++);
      Exps.push_back(MI);
      continue;
    }
```

```
    // Finished job of determining if there exists a common
subexpression.
  // Found a common subexpression, eliminate it.
    unsigned CSVN = VNT.lookup(MI);
    MachineInstr *CSMI = Exps[CSVN];
    DEBUG(dbgs() << "Examining: " << *MI);
    DEBUG(dbgs() << "*** Found a common subexpression: " <<
*CSMI);

    // Check if it's profitable to perform this CSE.
    bool DoCSE = true;
    unsigned NumDefs = MI->getDesc().getNumDefs() +
                       MI->getDesc().getNumImplicitDefs();

    for (unsigned i = 0, e = MI->getNumOperands(); NumDefs
&& i != e; ++i) {
      MachineOperand &MO = MI->getOperand(i);
      if (!MO.isReg() || !MO.isDef())
        continue;
      unsigned OldReg = MO.getReg();
      unsigned NewReg = CSMI->getOperand(i).getReg();

      // Go through implicit defs of CSMI and MI, if a def
is not dead at MI,
      // we should make sure it is not dead at CSMI.
      if (MO.isImplicit() && !MO.isDead() && CSMI->getOperand(i).
isDead())
        ImplicitDefsToUpdate.push_back(i);
      if (OldReg == NewReg) {
        --NumDefs;
        continue;
      }

      assert(TargetRegisterInfo::isVirtualRegister(OldReg)
&&
             TargetRegisterInfo::isVirtualRegister(NewReg)
&&
             "Do not CSE physical register defs!");

      if (!isProfitableToCSE(NewReg, OldReg, CSMI, MI)) {
        DEBUG(dbgs() << "*** Not profitable, avoid
CSE!\n");
        DoCSE = false;
        break;
```

```
      }

      // Don't perform CSE if the result of the old
instruction cannot exist
      // within the register class of the new instruction.
      const TargetRegisterClass *OldRC = MRI->getRegClass(OldReg);
      if (!MRI->constrainRegClass(NewReg, OldRC)) {
        DEBUG(dbgs() << "*** Not the same register class,
avoid CSE!\n");
        DoCSE = false;
        break;
      }

      CSEPairs.push_back(std::make_pair(OldReg, NewReg));
      --NumDefs;
    }

    // Actually perform the elimination.
    if (DoCSE) {
      for (unsigned i = 0, e = CSEPairs.size(); i != e;
++i) {
        MRI->replaceRegWith(CSEPairs[i].first, CSEPairs[i].
second);
        MRI->clearKillFlags(CSEPairs[i].second);
      }

      // Go through implicit defs of CSMI and MI, if a def
is not dead at MI,
      // we should make sure it is not dead at CSMI.
      for (unsigned i = 0, e = ImplicitDefsToUpdate.size();
i != e; ++i)
        CSMI->getOperand(ImplicitDefsToUpdate[i]).
setIsDead(false);

      if (CrossMBBPhysDef) {
        // Add physical register defs now coming in from a
predecessor to MBB
        // livein list.
        while (!PhysDefs.empty()) {
          unsigned LiveIn = PhysDefs.pop_back_val();
          if (!MBB->isLiveIn(LiveIn))
            MBB->addLiveIn(LiveIn);
        }
        ++NumCrossBBCSEs;
      }
```

```
      MI->eraseFromParent();
      ++NumCSEs;
      if (!PhysRefs.empty())
        ++NumPhysCSEs;
      if (Commuted)
        ++NumCommutes;
      Changed = true;
    } else {
      VNT.insert(MI, CurrVN++);
      Exps.push_back(MI);
    }
    CSEPairs.clear();
    ImplicitDefsToUpdate.clear();
  }

  return Changed;
}
```

4. Let's also look into the legality and profitability functions to determine the CSE candidates:

```
bool MachineCSE::isCSECandidate(MachineInstr *MI) {
// If Machine Instruction is PHI, or inline ASM or implicit
defs, it is not a candidate for CSE.

  if (MI->isPosition() || MI->isPHI() || MI-
>isImplicitDef() || MI->isKill() ||
      MI->isInlineAsm() || MI->isDebugValue())
    return false;

  // Ignore copies.
  if (MI->isCopyLike())
    return false;

  // Ignore instructions that we obviously can't move.
  if (MI->mayStore() || MI->isCall() || MI->isTerminator()
|| MI->hasUnmodeledSideEffects())
    return false;

  if (MI->mayLoad()) {
    // Okay, this instruction does a load. As a refinement,
we allow the target
    // to decide whether the loaded value is actually a
constant. If so, we can
    // actually use it as a load.
```

```
    if (!MI->isInvariantLoad(AA))
      return false;
  }
  return true;
}
```

5. The profitability function is written as follows:

```
bool MachineCSE::isProfitableToCSE(unsigned CSReg, unsigned
Reg,
  MachineInstr *CSMI, MachineInstr *MI) {

  // If CSReg is used at all uses of Reg, CSE should not
increase register
  // pressure of CSReg.
  bool MayIncreasePressure = true;
  if (TargetRegisterInfo::isVirtualRegister(CSReg) &&
      TargetRegisterInfo::isVirtualRegister(Reg)) {
    MayIncreasePressure = false;
    SmallPtrSet<MachineInstr*, 8> CSUses;
    for (MachineInstr &MI : MRI->use_nodbg_instructions(CSReg)) {
      CSUses.insert(&MI);
    }
    for (MachineInstr &MI : MRI-
>use_nodbg_instructions(Reg))
{
      if (!CSUses.count(&MI)) {
        MayIncreasePressure = true;
        break;
      }
    }
  }
  if (!MayIncreasePressure) return true;

  // Heuristics #1: Don't CSE "cheap" computation if the
def is not local or in
  // an immediate predecessor. We don't want to increase
register pressure and
  // end up causing other computation to be spilled.
  if (TII->isAsCheapAsAMove(MI)) {
    MachineBasicBlock *CSBB = CSMI->getParent();
    MachineBasicBlock *BB = MI->getParent();
    if (CSBB != BB && !CSBB->isSuccessor(BB))
      return false;
  }
```

```
  // Heuristics #2: If the expression doesn't not use a vr
and the only use
  // of the redundant computation are copies, do not cse.
  bool HasVRegUse = false;
  for (unsigned i = 0, e = MI->getNumOperands(); i != e;
++i) {
    const MachineOperand &MO = MI->getOperand(i);
    if (MO.isReg() && MO.isUse() &&
        TargetRegisterInfo::isVirtualRegister(MO.getReg()))
{
      HasVRegUse = true;
      break;
    }
  }
  if (!HasVRegUse) {
    bool HasNonCopyUse = false;
    for (MachineInstr &MI : MRI-
>use_nodbg_instructions(Reg)) {
      // Ignore copies.
      if (!MI.isCopyLike()) {
        HasNonCopyUse = true;
        break;
      }
    }
    if (!HasNonCopyUse)
      return false;
  }

  // Heuristics #3: If the common subexpression is used by
PHIs, do not reuse
  // it unless the defined value is already used in the BB
of the new use.
  bool HasPHI = false;
  SmallPtrSet<MachineBasicBlock*, 4> CSBBs;
  for (MachineInstr &MI : MRI-
>use_nodbg_instructions(CSReg)) {
    HasPHI |= MI.isPHI();
    CSBBs.insert(MI.getParent());
  }

  if (!HasPHI)
    return true;
  return CSBBs.count(MI->getParent());
}
```

How it works...

The `MachineCSE` pass runs on a machine function. It gets the `DomTree` information and then traverses the `DomTree` in the DFS way, creating a work list of nodes that are essentially `MachineBasicBlocks`. It then processes each block for CSE. In each block, it iterates through all the instructions and checks whether any instruction is a candidate for CSE. Then it checks whether it is profitable to eliminate the identified expression. Once it has found that the identified CSE is profitable to eliminate, it eliminates the `MachineInstruction` class from the `MachineBasicBlock` class. It also performs a simple copy propagation of the machine instruction. In some cases, the `MachineInstruction` may not be a candidate for CSE in its initial run, but may become one after copy propagation.

See more

To see more machine code optimization in the SSA form, look into the implementation of the machine dead code elimination pass in the `lib/CodeGen/DeadMachineInstructionElim.cpp file`.

Analyzing live intervals

Further on in this chapter, we will be looking into register allocation. Before we head to that, however, you must understand the concepts of **live variable** and **live interval**. By live intervals, we mean the range in which a variable is live, that is, from the point where a variable is defined to its last use. For this, we need to calculate the set of registers that are immediately dead after the instruction (the last use of a variable), and the set of registers that are used by the instruction but not after the instruction. We calculate live variable information for each virtual register and physical register in the function. Using SSA to sparsely compute the lifetime information for the virtual registers enables us to only track the physical registers within a block. Before register allocation, LLVM assumes that physical registers are live only within a single basic block. This enables it to perform a single, local analysis to resolve physical register lifetimes within each basic block. After performing the live variable analysis, we have the information required for performing live interval analysis and building live intervals. For this, we start numbering the basic block and machine instructions. After that live-in values, typically arguments in registers are handled. Live intervals for virtual registers are computed for some ordering of the machine instructions (*1*, *N*). A live interval is an interval (*i*, *j*) for which a variable is live, where *1 >= i >= j > N*.

In this recipe, we will take a sample program and see how we can list down the live intervals for that program. We will look at how LLVM works to calculate these intervals.

Getting ready

To get started, we need a piece of test code on which we will be performing live interval analysis. For simplicity, we will use C code and then convert it into LLVM IR:

1. Write a test program with an `if-else` block:

```
$ cat interval.c
void donothing(int a) {
  return;
}

int func(int i) {
  int a = 5;
  donothing(a);
  int m = a;
  donothing(m);
  a = 9;
  if (i < 5) {
    int b = 3;
    donothing(b);
    int z = b;
    donothing(z);
  }
  else {
    int k = a;
    donothing(k);
  }

  return m;
}
```

2. Use Clang to convert the C code into IR, and then view the generated IR using the `cat` command:

```
$ clang -cc1 -emit-llvm interval.c

$ cat interval.ll
; ModuleID = 'interval.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
define void @donothing(i32 %a) #0 {
  %1 = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  ret void
}

; Function Attrs: nounwind
define i32 @func(i32 %i) #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %m = alloca i32, align 4
  %b = alloca i32, align 4
  %z = alloca i32, align 4
  %k = alloca i32, align 4
  store i32 %i, i32* %1, align 4
  store i32 5, i32* %a, align 4
  %2 = load i32, i32* %a, align 4
  call void @donothing(i32 %2)
  %3 = load i32, i32* %a, align 4
  store i32 %3, i32* %m, align 4
  %4 = load i32, i32* %m, align 4
  call void @donothing(i32 %4)
  store i32 9, i32* %a, align 4
  %5 = load i32, i32* %1, align 4
  %6 = icmp slt i32 %5, 5
  br i1 %6, label %7, label %11

; <label>:7                                ; preds = %0
  store i32 3, i32* %b, align 4
  %8 = load i32, i32* %b, align 4
  call void @donothing(i32 %8)
  %9 = load i32, i32* %b, align 4
```

```
  store i32 %9, i32* %z, align 4
  %10 = load i32, i32* %z, align 4
  call void @donothing(i32 %10)
  br label %14

; <label>:11                                ; preds = %0
  %12 = load i32, i32* %a, align 4
  store i32 %12, i32* %k, align 4
  %13 = load i32, i32* %k, align 4
  call void @donothing(i32 %13)
  br label %14

; <label>:14                                ; preds = %11, %7
  %15 = load i32, i32* %m, align 4
  ret i32 %15
}

attributes #0 = { nounwind "less-precise-fpmad"="false"
"no-frame-pointer-elim"="false" "no-infs-fp-math"="false"
"no-nans-fp-math"="false" "no-realign-stack" "stack-
protector-buffer-size"="8" "unsafe-fp-math"="false" "use-
soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.7.0 (trunk 234045)"}
```

How to do it...

1. To list the live intervals, we will need to modify the code of the `LiveIntervalAnalysis.cpp` file by adding code to print the live intervals. We will add the following lines (marked with a + symbol before each added line):

```
void LiveIntervals::computeVirtRegInterval(LiveInterval
&LI) {
  assert(LRCalc && "LRCalc not initialized.");
  assert(LI.empty() && "Should only compute empty
intervals.");
  LRCalc->reset(MF, getSlotIndexes(), DomTree,
&getVNInfoAllocator());
```

```
  LRCalc->calculate(LI, MRI->shouldTrackSubRegLiveness(LI.reg));
  computeDeadValues(LI, nullptr);

/**** add the following code ****/
+ llvm::outs() << "********** INTERVALS **********\n";

  // Dump the regunits.
  + for (unsigned i = 0, e = RegUnitRanges.size(); i != e;
++i)
    + if (LiveRange *LR = RegUnitRanges[i])
      + llvm::outs() << PrintRegUnit(i, TRI) << ' ' << *LR
<< '\n';

  // Dump the virtregs.
  + llvm::outs() << "virtregs:";
  + for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e;
++i) {
    + unsigned Reg = TargetRegisterInfo::index2VirtReg(i);
    + if (hasInterval(Reg))
      + llvm::outs() << getInterval(Reg) << '\n';
  + }
```

2. Build LLVM after modifying the preceding source file, and install it on the path.
3. Now compile the test code in the IR form using the `llc` command. You will get the live intervals:

```
$ llc interval.ll
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
%vreg2 [144r,192r:0)  0@144r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
```

```
%vreg2 [144r,192r:0)  0@144r
%vreg5 [544r,592r:0)  0@544r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
%vreg2 [144r,192r:0)  0@144r
%vreg5 [544r,592r:0)  0@544r
%vreg6 [352r,368r:0)  0@352r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
%vreg2 [144r,192r:0)  0@144r
%vreg5 [544r,592r:0)  0@544r
%vreg6 [352r,368r:0)  0@352r
%vreg7 [416r,464r:0)  0@416r
********** INTERVALS **********
virtregs:%vreg0 [16r,32r:0)  0@16r
%vreg1 [80r,96r:0)  0@80r
%vreg2 [144r,192r:0)  0@144r
%vreg5 [544r,592r:0)  0@544r
%vreg6 [352r,368r:0)  0@352r
%vreg7 [416r,464r:0)  0@416r
%vreg8 [656r,672r:0)  0@656r
```

How it works...

In the preceding example, we saw how live intervals are associated with each virtual register. The program points at the beginning and the end of live intervals are marked in square brackets. The process of generating these live intervals starts from the `LiveVariables::runOnMachineFunction(MachineFunction &mf)` function in the `lib/CodeGen/LiveVariables.cpp file`, where it assigns the definition and usage of the registers using the `HandleVirtRegUse` and `HandleVirtRegDef` functions. It gets the `VarInfo` object for the given virtual register using the `getVarInfo` function.

The `LiveInterval` and `LiveRange` classes are defined in `LiveInterval.cpp`. The functions in this file takes the information on the liveliness of each variable and then checks whether they overlap or not.

In the `LiveIntervalAnalysis.cpp` file, we have the implementation of the live interval analysis pass, which scans the basic blocks (ordered in a linear fashion) in depth-first order, and creates a live interval for each virtual and physical register. This analysis is used by the register allocators, which will be discussed in next recipe.

See also

- If you want to see in detail how the virtual registers for different basic blocks get generated, and see the lifetime of these virtual registers, use the `-debug-only=regalloc` command-line option with the `llc` tool when compiling the test case. You need a debug build of the LLVM for this.
- To get more detail on live intervals, go through these code files:
  - `Lib/CodeGen/ LiveInterval.cpp`
  - `Lib/CodeGen/ LiveIntervalAnalysis.cpp`
  - `Lib/CodeGen/ LiveVariables.cpp`

Allocating registers

Register allocation is the task of assigning physical registers to virtual registers. Virtual registers can be infinite, but the physical registers for a machine are limited. So, register allocation is aimed at maximizing the number of physical registers getting assigned to virtual registers. In this recipe, we will see how registers are represented in LLVM, how can we tinker with the register information, the steps taking place, and built-in register allocators.

Getting ready

You need to build and install LLVM.

How to do it...

1. To see how registers are represented in LLVM, open the `build-folder/lib/Target/X86/X86GenRegisterInfo.inc` file and check out the first few lines, which show that registers are represented as integers:

```
namespace X86 {
enum {
  NoRegister,
  AH = 1,
  AL = 2,
  AX = 3,
  BH = 4,
  BL = 5,
```

```
  BP = 6,
  BPL = 7,
  BX = 8,
  CH = 9,
…
```

2. For architectures that have registers that share the same physical location, check out the `RegisterInfo.td` file of that architecture for alias information. Let's check out the `lib/Target/X86/X86RegisterInfo.td` file. By looking at the following code snippet, we see how the `EAX`, `AX`, and `AL` registers are aliased (we only specify the smallest register alias):

```
def AL : X86Reg<"al", 0>;
def DL : X86Reg<"dl", 2>;
def CL : X86Reg<"cl", 1>;
def BL : X86Reg<"bl", 3>;

def AH : X86Reg<"ah", 4>;
def DH : X86Reg<"dh", 6>;
def CH : X86Reg<"ch", 5>;
def BH : X86Reg<"bh", 7>;

def AX : X86Reg<"ax", 0, [AL,AH]>;
def DX : X86Reg<"dx", 2, [DL,DH]>;
def CX : X86Reg<"cx", 1, [CL,CH]>;
def BX : X86Reg<"bx", 3, [BL,BH]>;

// 32-bit registers
let SubRegIndices = [sub_16bit] in {
def EAX : X86Reg<"eax", 0, [AX]>, DwarfRegNum<[-2, 0, 0]>;
def EDX : X86Reg<"edx", 2, [DX]>, DwarfRegNum<[-2, 2, 2]>;
def ECX : X86Reg<"ecx", 1, [CX]>, DwarfRegNum<[-2, 1, 1]>;
def EBX : X86Reg<"ebx", 3, [BX]>, DwarfRegNum<[-2, 3, 3]>;
def ESI : X86Reg<"esi", 6, [SI]>, DwarfRegNum<[-2, 6, 6]>;
def EDI : X86Reg<"edi", 7, [DI]>, DwarfRegNum<[-2, 7, 7]>;
def EBP : X86Reg<"ebp", 5, [BP]>, DwarfRegNum<[-2, 4, 5]>;
def ESP : X86Reg<"esp", 4, [SP]>, DwarfRegNum<[-2, 5, 4]>;
def EIP : X86Reg<"eip", 0, [IP]>, DwarfRegNum<[-2, 8, 8]>;
…
```

3. To change the number of physical registers available, go to the `TargetRegisterInfo.td` file and manually comment out some of the registers, which are the last parameters of the `RegisterClass`. Open the `X86RegisterInfo.cpp` file and remove the registers `AH`, `CH`, and `DH`:

```
def GR8 : RegisterClass<"X86", [i8],  8,
                        (add AL, CL, DL, AH, CH, DH, BL,
BH, SIL, DIL, BPL, SPL,
                             R8B, R9B, R10B, R11B, R14B,
R15B, R12B, R13B)> {
```

4. When you build LLVM, the `.inc` file in the first step will have been changed and will not contain the `AH`, `CH`, and DH registers.

5. Use the test case from the previous recipe, *Analyzing live intervals*, in which we performed live interval analysis, and run the register allocation techniques provided by LLVM, namely `fast`, `basic`, `greedy`, and `pbqp`. Let's run two of them here and compare the results:

```
$ llc -regalloc=basic interval.ll -o intervalregbasic.s
```

Next, create the `intervalregbasic.s` file as shown:

```
$ cat intervalregbasic.s
  .text
  .file  "interval.ll"
  .globl  donothing
  .align  16, 0x90
  .type  donothing,@function
donothing:                              # @donothing
# BB#0:
  movl  %edi, -4(%rsp)
  retq
.Lfunc_end0:
  .size  donothing, .Lfunc_end0-donothing

  .globl  func
  .align  16, 0x90
  .type  func,@function
func:                                   # @func
# BB#0:
  subq  $24, %rsp
  movl  %edi, 20(%rsp)
```

```
  movl  $5, 16(%rsp)
  movl  $5, %edi
  callq  donothing
  movl  16(%rsp), %edi
  movl  %edi, 12(%rsp)
  callq  donothing
  movl  $9, 16(%rsp)
  cmpl  $4, 20(%rsp)
  jg  .LBB1_2
# BB#1:
  movl  $3, 8(%rsp)
  movl  $3, %edi
  callq  donothing
  movl  8(%rsp), %edi
  movl  %edi, 4(%rsp)
  jmp  .LBB1_3
.LBB1_2:
  movl  16(%rsp), %edi
  movl  %edi, (%rsp)
.LBB1_3:
  callq  donothing
  movl  12(%rsp), %eax
  addq  $24, %rsp
  retq
.Lfunc_end1:
  .size  func, .Lfunc_end1-func
```

Next, run the following command to compare the two files:

```
$ llc -regalloc=pbqp interval.ll -o intervalregpbqp.s
```

Create the `intervalregbqp.s` file:

```
$cat intervalregpbqp.s
  .text
  .file  "interval.ll"
  .globl  donothing
  .align  16, 0x90
  .type  donothing,@function
```

```
donothing:                              # @donothing
# BB#0:
  movl  %edi, %eax
  movl  %eax, -4(%rsp)
  retq
.Lfunc_end0:
  .size  donothing, .Lfunc_end0-donothing

  .globl  func
  .align  16, 0x90
  .type  func,@function
func:                                   # @func
# BB#0:
  subq  $24, %rsp
  movl  %edi, %eax
  movl  %eax, 20(%rsp)
  movl  $5, 16(%rsp)
  movl  $5, %edi
  callq  donothing
  movl  16(%rsp), %eax
  movl  %eax, 12(%rsp)
  movl  %eax, %edi
  callq  donothing
  movl  $9, 16(%rsp)
  cmpl  $4, 20(%rsp)
  jg  .LBB1_2
# BB#1:
  movl  $3, 8(%rsp)
  movl  $3, %edi
  callq  donothing
  movl  8(%rsp), %eax
  movl  %eax, 4(%rsp)
  jmp  .LBB1_3
.LBB1_2:
  movl  16(%rsp), %eax
  movl  %eax, (%rsp)
```

```
.LBB1_3:
  movl  %eax, %edi
  callq  donothing
  movl  12(%rsp), %eax
  addq  $24, %rsp
  retq
.Lfunc_end1:
  .size  func, .Lfunc_end1-func
```

6. Now, use a `diff` tool and compare the two assemblies side by side.

How it works...

The mapping of virtual registers on physical registers can be done in two ways:

- **Direct Mapping**: By making use of the `TargetRegisterInfo` and `MachineOperand` classes. This depends on the developer, who needs to provide the location where load and store instructions should be inserted in order to get and store values in the memory.
- **Indirect Mapping**: This depends on the `VirtRegMap` class to insert loads and stores, and to get and set values from the memory. Use the `VirtRegMap::assignVirt2Phys(vreg, preg)` function to map a virtual register on a physical one.

Another important role that the register allocator plays is in SSA form deconstruction. As traditional instruction sets do not support the `phi` instruction, we must replace it with other instructions to generate the machine code. The traditional way was to replace the `phi` instruction with the `copy` instruction.

After this stage, we do the actual mapping on the physical registers. We have four implementations of register allocation in LLVM, which have their algorithms for mapping the virtual registers on the physical registers. It is not possible to cover in detail any of those algorithms here. If you want to try and understand them, refer to the next section.

See also

- To learn more about the algorithms used in LLVM, look through the source codes located at `lib/CodeGen/`:
  - `lib/CodeGen/RegAllocBasic.cpp`
  - `lib/CodeGen/ RegAllocFast.cpp`
  - `lib/CodeGen/ RegAllocGreedy.cpp`
  - `lib/CodeGen/ RegAllocPBQP.cpp`

Inserting the prologue-epilogue code

Inserting the prologue-epilogue code involves stack unwinding, finalizing the function layout, saving callee-saved registers and emitting the prologue and epilogue code. It also replaces abstract frame indexes with appropriate references. This pass runs after the register allocation phase.

How to do it...

The skeleton and the important functions defined in the `PrologueEpilogueInserter` class are as follows:

- The prologue epilogue inserter pass runs on a machine function, hence it inherits the `MachineFunctionPass` class. Its constructor initializes the pass:

```
class PEI : public MachineFunctionPass {
  public:
    static char ID;
    PEI() : MachineFunctionPass(ID) {
      initializePEIPass(*PassRegistry::getPassRegistry());
    }
```

- There are various helper functions defined in this class that help insert the prologue and epilogue code:

```
    void calculateSets(MachineFunction &Fn);
    void calculateCallsInformation(MachineFunction &Fn);
    void calculateCalleeSavedRegisters(MachineFunction &Fn);
    void insertCSRSpillsAndRestores(MachineFunction &Fn);
    void calculateFrameObjectOffsets(MachineFunction &Fn);
    void replaceFrameIndices(MachineFunction &Fn);
    void replaceFrameIndices(MachineBasicBlock *BB,
MachineFunction &Fn,
                             int &SPAdj);
    void scavengeFrameVirtualRegs(MachineFunction &Fn);
```

- The main function, `insertPrologEpilogCode()`, does the task of inserting the prologue and epilogue code:

```
    void insertPrologEpilogCode(MachineFunction &Fn);
```

- The first function to execute in this pass is the `runOnFunction()` function. The comments in the code show the various operations carried out, such as calculating the call frame size, adjusting the stack variables, inserting the spill code for the callee-saved register for modified registers, calculating the actual frame offset, inserting the prologue and epilogue code for the function, replacing the abstract frame index with the actual offsets, and so on:

```
bool PEI::runOnMachineFunction(MachineFunction &Fn) {
  const Function* F = Fn.getFunction();
  const TargetRegisterInfo *TRI = Fn.getSubtarget().
getRegisterInfo();
  const TargetFrameLowering *TFI = Fn.getSubtarget().
getFrameLowering();

  assert(!Fn.getRegInfo().getNumVirtRegs() && "Regalloc
must assign all vregs");

  RS = TRI->requiresRegisterScavenging(Fn) ? new
RegScavenger() : nullptr;
  FrameIndexVirtualScavenging = TRI->requiresFrameIndexScavenging(
Fn);

  // Calculate the MaxCallFrameSize and AdjustsStack
variables for the
  // function's frame information. Also eliminates call
frame pseudo
  // instructions.
  calculateCallsInformation(Fn);

  // Allow the target machine to make some adjustments to
the function
  // e.g. UsedPhysRegs before
calculateCalleeSavedRegisters.
  TFI->processFunctionBeforeCalleeSavedScan(Fn, RS);

  // Scan the function for modified callee saved registers
and insert spill code
  // for any callee saved registers that are modified.
  calculateCalleeSavedRegisters(Fn);

  // Determine placement of CSR spill/restore code:
  // place all spills in the entry block, all restores in
return blocks.
  calculateSets(Fn);

  // Add the code to save and restore the callee saved
registers
  if (!F->hasFnAttribute(Attribute::Naked))
    insertCSRSpillsAndRestores(Fn);

  // Allow the target machine to make final modifications
to the function
  // before the frame layout is finalized.
```

```
  TFI->processFunctionBeforeFrameFinalized(Fn, RS);

  // Calculate actual frame offsets for all abstract stack
objects...
  calculateFrameObjectOffsets(Fn);

  // Add prolog and epilog code to the function.  This
function is required
  // to align the stack frame as necessary for any stack
variables or
  // called functions.  Because of this,
calculateCalleeSavedRegisters()
  // must be called before this function in order to set
the AdjustsStack
  // and MaxCallFrameSize variables.
  if (!F->hasFnAttribute(Attribute::Naked))
    insertPrologEpilogCode(Fn);

  // Replace all MO_FrameIndex operands with physical
register references
  // and actual offsets.
  replaceFrameIndices(Fn);

  // If register scavenging is needed, as we've enabled
doing it as a
  // post-pass, scavenge the virtual registers that frame
index elimination
  // inserted.
  if (TRI->requiresRegisterScavenging(Fn) &&
FrameIndexVirtualScavenging)
    scavengeFrameVirtualRegs(Fn);

  // Clear any vregs created by virtual scavenging.
  Fn.getRegInfo().clearVirtRegs();

  // Warn on stack size when we exceeds the given limit.
  MachineFrameInfo *MFI = Fn.getFrameInfo();
  uint64_t StackSize = MFI->getStackSize();
  if (WarnStackSize.getNumOccurrences() > 0 &&
WarnStackSize < StackSize) {
    DiagnosticInfoStackSize DiagStackSize(*F, StackSize);
    F->getContext().diagnose(DiagStackSize);
  }
  delete RS;
  ReturnBlocks.clear();
  return true;
}
```

- The main function that inserts prologue-epilogue code is the `insertPrologEpilogCode()` function. This function first takes the `TargetFrameLowering` object and then emits a prologue code for that function corresponding to that target. After that, for each basic block in that function, it checks whether there is a return statement. If there is a return statement, then it emits an epilogue code for that function:

```
void PEI::insertPrologEpilogCode(MachineFunction &Fn) {
  const TargetFrameLowering &TFI = *Fn.getSubtarget().
getFrameLowering();

  // Add prologue to the function.
  TFI.emitPrologue(Fn);

  // Add epilogue to restore the callee-save registers in
each exiting block
  for (MachineFunction::iterator I = Fn.begin(), E =
Fn.end(); I != E; ++I) {
    // If last instruction is a return instruction, add an
epilogue
    if (!I->empty() && I->back().isReturn())
      TFI.emitEpilogue(Fn, *I);
  }

  // Emit additional code that is required to support
segmented stacks, if
  // we've been asked for it.  This, when linked with a
runtime with support
  // for segmented stacks (libgcc is one), will result in
allocating stack
  // space in small chunks instead of one large contiguous
block.
  if (Fn.shouldSplitStack())
    TFI.adjustForSegmentedStacks(Fn);

  // Emit additional code that is required to explicitly
handle the stack in
  // HiPE native code (if needed) when loaded in the
Erlang/OTP runtime. The
  // approach is rather similar to that of Segmented
Stacks, but it uses a
  // different conditional check and another BIF for
allocating more stack
  // space.
  if (Fn.getFunction()->getCallingConv() ==
CallingConv::HiPE)
    TFI.adjustForHiPEPrologue(Fn);
}
```

How it works...

The preceding code invokes the `emitEpilogue()` and the `emitPrologue()` functions in the `TargetFrameLowering` class, which will be discussed in the target-specific frame lowering recipes in later chapters.

Code emission

The code emission phase lowers the code from code generator abstractions (such as `MachineFunction` class, `MachineInstr` class, and so on) to machine code layer abstractions (`MCInst` class, `MCStreamer` class, and so on). The important classes in this phase are the target-independent `AsmPrinter` class, target-specific subclasses of `AsmPrinter`, and the `TargetLoweringObjectFile` class.

The MC layer is responsible for emitting object files, which consist of labels, directives, and instructions; while the `CodeGen` layer consists of `MachineFunctions`, `MachineBasicBlock` and `MachineInstructions`. A key class used at this point in time is the `MCStreamer` class, which consists of assembler APIs. The `MCStreamer` class has functions such as `EmitLabel`, `EmitSymbolAttribute`, `SwitchSection`, and so on, which directly correspond to the aforementioned assembly-level directives.

There are four important things that need to be implemented for the target in order to emit code:

- Define a subclass of the `AsmPrinter` class for the target. This class implements the general lowering process, converting the `MachineFunctions` functions into MC label constructs. The `AsmPrinter` base class methods and routines help implement a target-specific `AsmPrinter` class. The `TargetLoweringObjectFile` class implements much of the common logic for the `ELF`, `COFF`, or `MachO` targets.
- Implement an instruction printer for the target. The instruction printer takes an `MCInst` class and renders it into a `raw_ostream` class as text. Most of this is automatically generated from the `.td` file (when you specify something like add `$dst`, `$src1`, `$src2` in the instructions), but you need to implement routines to print operands.
- Implement code that lowers a `MachineInstr` class to an `MCInst class`, usually implemented in `<target>MCInstLower.cpp`. This lowering process is often target-specific, and is responsible for turning jump table entries, constant pool indices, global variable addresses, and so on into `MCLabels`, as appropriate. The instruction printer or the encoder takes the `MCInsts` that are generated.
- Implement a subclass of `MCCodeEmitter` that lowers `MCInsts` to machine code bytes and relocations. This is important if you want to support direct `.o` file emission, or want to implement an assembler for your target.

How to do it...

Let's visit some important functions in the `AsmPrinter` base class in the `lib/CodeGen/AsmPrinter/AsmPrinter.cpp` file:

- `EmitLinkage()`: This emits the linkage of the given variables or functions:

```
void AsmPrinter::EmitLinkage(const GlobalValue *GV,
MCSymbol *GVSym) const ;
```

- `EmitGlobalVariable()`: This emits the specified global variable to the `.s` file:

```
void AsmPrinter::EmitGlobalVariable(const GlobalVariable *GV);
```

- `EmitFunctionHeader()`: This emits the header of the current function:

```
void AsmPrinter::EmitFunctionHeader();
```

- `EmitFunctionBody()`: This method emits the body and trailer of a function:

```
void AsmPrinter::EmitFunctionBody();
```

- `EmitJumpTableInfo()`: This prints assembly representations of the jump tables used by the current function to the current output stream:

```
void AsmPrinter::EmitJumpTableInfo();
```

- `EmitJumpTableEntry()`: This emits a jump table entry for the specified `MachineBasicBlock` class to the current stream:

```
void AsmPrinter::EmitJumpTableEntry(const
MachineJumpTableInfo *MJTI, const MachineBasicBlock *MBB,
unsigned UID) const;
```

- Emit integer types of 8, 16, or 32 bit size:

```
void AsmPrinter::EmitInt8(int Value) const {
  OutStreamer.EmitIntValue(Value, 1);
}

void AsmPrinter::EmitInt16(int Value) const {
  OutStreamer.EmitIntValue(Value, 2);
}

void AsmPrinter::EmitInt32(int Value) const {
OutStreamer.EmitIntValue(Value, 4);
}
```

For detailed implementation on code emission , see the `lib/CodeGen/AsmPrinter/AsmPrinter.cpp` file. One important thing to note is that this class uses the `OutStreamer` class object to output assembly instructions. The details of target-specific code emission will be covered in later chapters.

Tail call optimization

In this recipe, we will see how **tail call optimization** is done in LLVM. Tail call optimization is a technique where the callee reuses the stack of the caller instead of adding a new stack frame to the call stack, hence saving stack space and the number of returns when dealing with mutually recursive functions.

Getting ready

We need to make sure of the following:

- The `llc` tool must be installed in `$PATH`
- The `tailcallopt` option must be enabled
- The test code must have a tail call

How to do it...

1. Write the test code for checking tail call optimization:

```
$ cat tailcall.ll
declare fastcc i32 @tailcallee(i32 inreg %a1, i32 inreg %a2,
i32 %a3, i32 %a4)

define fastcc i32 @tailcaller(i32 %in1, i32 %in2) {
  %l1 = add i32 %in1, %in2
  %tmp = tail call fastcc i32 @tailcallee(i32 inreg %in1, i32
inreg %in2, i32 %in1, i32 %l1)
  ret i32 %tmp
}
```

2. Run the `llc` tool with the `-tailcallopt` option on the test code to generate the assembly file with the tailcall-optimized code:

```
$ llc -tailcallopt tailcall.ll
```

3. Display the output generated:

```
$ cat tailcall.s
  .text
  .file  "tailcall.ll"
  .globl  tailcaller
  .align  16, 0x90
  .type  tailcaller,@function
```

```
tailcaller:                             # @tailcaller
  .cfi_startproc
# BB#0:
  pushq  %rax
.Ltmp0:
  .cfi_def_cfa_offset 16
                    # kill: ESI<def> ESI<kill> RSI<def>
                    # kill: EDI<def> EDI<kill> RDI<def>
  leal   (%rdi,%rsi), %ecx
                    # kill: ESI<def> ESI<kill> RSI<kill>
  movl   %edi, %edx
  popq   %rax
  jmp  tailcallee              # TAILCALL
.Lfunc_end0:
  .size  tailcaller, .Lfunc_end0-tailcaller
  .cfi_endproc

  .section  ".note.GNU-stack","",@progbits
```

4. Using the `llc` tool, generate the assembly again but without using the `-tailcallopt` option:

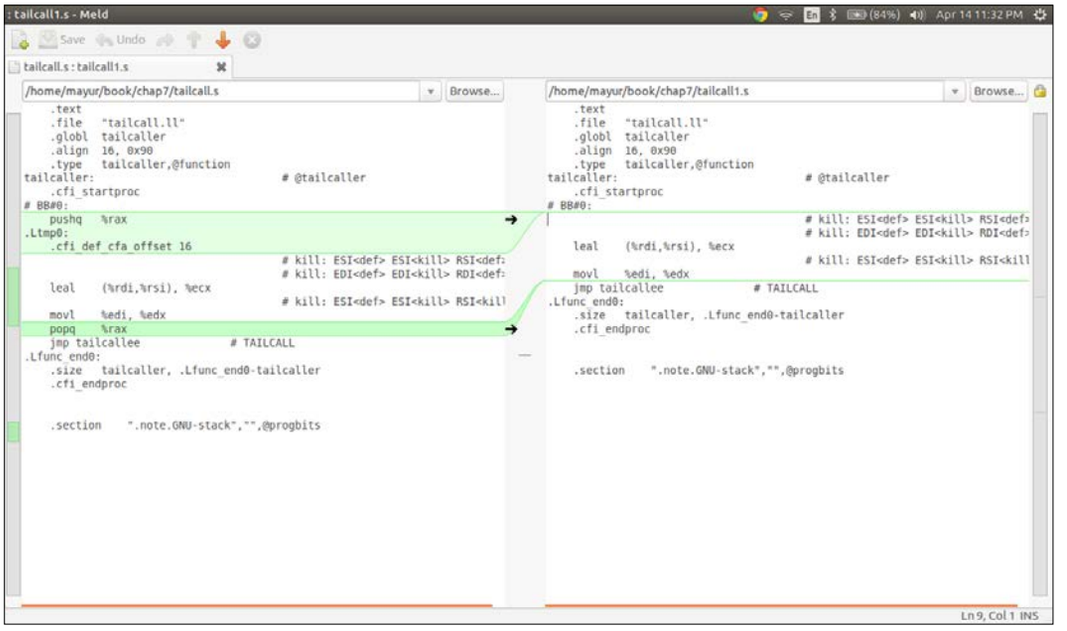
```
$ llc tailcall.ll -o tailcall1.s
```

5. Display the output using the `cat` command:

```
$ cat tailcall1.s
  .text
  .file  "tailcall.ll"
  .globl  tailcaller
  .align  16, 0x90
  .type  tailcaller,@function
tailcaller:                             # @tailcaller
  .cfi_startproc
# BB#0:
                    # kill: ESI<def> ESI<kill> RSI<def>
                    # kill: EDI<def> EDI<kill> RDI<def>
  leal   (%rdi,%rsi), %ecx
                    # kill: ESI<def> ESI<kill> RSI<kill>
```

```
    movl  %edi, %edx
    jmp  tailcallee                # TAILCALL
.Lfunc_end0:
    .size  tailcaller, .Lfunc_end0-tailcaller
    .cfi_endproc
    .section  ".note.GNU-stack","",@progbits
```

Compare the two assemblies using a diff tool. We used the meld tool here:



How it works...

The tail call optimization is a compiler optimization technique, which a compiler can use to make a call to a function and take up no additional stack space; we don't need to create a new stack frame for this function call. This happens if the last instruction executed in a function is a call to another function. A point to note is that the caller function now does not need the stack space; it simply calls a function (another function or itself) and returns whatever value the called function would have returned. This optimization can make recursive calls take up constant and limited space. In this optimization, the code might not always be in the form for which a tail call is possible. It tries and modifies the source to see whether a tail call is possible or not.

In the preceding test case, we see that a push-and-pop instruction is added due to tail call optimization. In LLVM, the tail call optimization is handled by the architecture-specific `ISelLowering.cpp` file; for x86, it is the `X86ISelLowering.cpp` file:

```
The code in function SDValue X86TargetLowering::LowerCall (…….)
bool IsMustTail = CLI.CS && CLI.CS->isMustTailCall();
  if (IsMustTail) {
    // Force this to be a tail call.  The verifier rules are
enough to ensure
    // that we can lower this successfully without moving the
return address
    // around.
    isTailCall = true;
  } else if (isTailCall) {
    // Check if it's really possible to do a tail call.
    isTailCall = IsEligibleForTailCallOptimization(Callee,
CallConv,
                    isVarArg, SR != NotStructReturn,
                    MF.getFunction()->hasStructRetAttr(), CLI.
RetTy,
                    Outs, OutVals, Ins, DAG);
```

The preceding code is used to call the `IsEligibleForTailCallOptimization()` function when the `tailcallopt` flag is passed. The `IsEligibleForTailCallOptimization()` function decides whether or not the piece of code is eligible for tail call optimization. If it is, then the code generator will make the necessary changes.

Sibling call optimisation

In this recipe, we will see how **sibling call optimization** works in LLVM. Sibling call optimization can be looked at as an optimized tail call, the only constraint being that the functions should share a similar function signature, that is, matching return types and matching function arguments.

Getting ready

Write a test case for sibling call optimization, making sure that the caller and callee have the same calling conventions (in either C or **fastcc**), and that the call in the tail position is a tail call:

```
$ cat sibcall.ll
declare i32 @bar(i32, i32)

define i32 @foo(i32 %a, i32 %b, i32 %c) {
```

```
  entry:
    %0 = tail call i32 @bar(i32 %a, i32 %b)
  ret i32 %0
}
```

How to do it...

1. Run the `llc` tool to generate the assembly:

```
$ llc sibcall.ll
```

2. View the generated assembly using the `cat` command:

```
$ cat sibcall.s
    .text
    .file    "sibcall.ll"
    .globl    foo
    .align    16, 0x90
    .type    foo,@function
foo:                                    # @foo
    .cfi_startproc
# BB#0:                                 # %entry
    jmp    bar                      # TAILCALL
.Lfunc_end0:
    .size    foo, .Lfunc_end0-foo
    .cfi_endproc

    .section    ".note.GNU-stack","",@progbits
```

How it works...

Sibling call optimization is a restricted version of tail call optimization that can be performed on tail calls without passing the `tailcallopt` option. Sibling call optimization works in a similar way to tail call optimization, except that the sibling calls are automatically detected and do not need any ABI changes. The similarity needed in the function signatures is because when the caller function (which calls a tail recursive function) tries to clean up the callee's argument, after the callee has done its work, this may lead to memory leak if the callee exceeds the argument space to perform a sibling call to a function requiring more stack space for arguments.

8
Writing an LLVM Backend

In this chapter, we will cover the following recipes:

- Defining registers and register sets
- Defining the calling convention
- Defining the instruction set
- Implementing frame lowering
- Printing an instruction
- Selecting an instruction
- Adding instruction encoding
- Supporting a subtarget
- Lowering to multiple instructions
- Registering a target

Introduction

The ultimate goal of a compiler is to produce a target code, or an assembly code that can be converted into object code and executed on the actual hardware. To generate the assembly code, the compiler needs to know the various aspects of the architecture of the target machine—the registers, instruction set, calling convention, pipeline, and so on. There are lots of optimizations that can be done in this phase as well.

LLVM has its own way of defining the target machine. It uses `tablegen` to specify the target registers, instructions, calling convention, and so on. The `tablegen` function eases the way we describe a large set of architecture properties in a programmatic way.

LLVM has a pipeline structure for the backend, where instructions travel through phases like this; from the LLVM IR to `SelectionDAG`, then to `MachineDAG`, then to `MachineInstr`, and finally to `MCInst`.

The IR is converted into SelectionDAG (**DAG** stands for **Directed Acyclic Graph**). Then SelectionDAG legalization occurs where illegal instructions are mapped on the legal operations permitted by the target machine. After this stage, SelectionDAG is converted to MachineDAG, which is basically an instruction selection supported by the backend.

CPUs execute a linear sequence of instructions. The goal of the scheduling step is to linearize the DAG by assigning an order to its operations. LLVM's code generator employs clever heuristics (such as register pressure reduction) to try and produce a schedule that will result in faster code. Register allocation policies also play an important role in producing better LLVM code.

This chapter describes how to build an LLVM toy backend from scratch. By the end of this chapter, we will be able to generate assembly code for a sample toy backend.

A sample backend

The sample backend considered in this chapter is a simple RISC-type architecture, with a few registers (say r0-r3), a stack pointer (sp), and a link register (lr), for storing the return address.

The calling convention of this toy backend is similar to the ARM architecture—arguments passed to the function will be stored in register sets r0-r1, and the return value will be stored in r0.

Defining registers and registers sets

This recipe shows you how to define registers and register sets in `.td` files. The `tablegen` function will convert this `.td` file into `.inc` files, which will be the `#include` declarative in our `.cpp` files and refer to registers.

Getting ready

We have defined our toy target machine to have four registers (r0-r3), a stack register (sp), and a link register (lr). These can be specified in the `TOYRegisterInfo.td` file. The `tablegen` function provides the `Register` class, which can be extended to specify the registers.

How to do it...

To define the backend architecture using target descriptor files, proceed with the following steps.

1. Create a new folder in `lib/Target` named `TOY`:

   ```
   $ mkdir llvm_root_directory/lib/Target/TOY
   ```

2. Create a new `TOYRegisterInfo.td file` in the new `TOY` folder:

   ```
   $ cd llvm_root_directory/lib/Target/TOY
   $ vi TOYRegisterInfo.td
   ```

3. Define the hardware encoding, namespace, registers, and the register class:

   ```
   class TOYReg<bits<16> Enc, string n> : Register<n> {
     let HWEncoding = Enc;
     let Namespace = "TOY";
   }

   foreach i = 0-3 in {
       def R#i : R<i, "r"#i >;
   }

   def SP  : TOYReg<13, "sp">;
   def LR  : TOYReg<14, "lr">;

   def GRRegs : RegisterClass<"TOY", [i32], 32,
     (add R0, R1, R2, R3, SP)>;
   ```

How it works...

The `tablegen` function processes this `.td` file to generate the `.inc` file, which generally has enums generated for these registers. These enums can be used in the `.cpp` files, in which the registers can be referenced as `TOY::R0`. These `.inc` files will be generated when we build the LLVM project.

See also

- To get more details about how registers are defined for more advanced architecture, such as ARM, refer to the `lib/Target/ARM/ARMRegisterInfo.td` file in the source code of LLVM.

Defining the calling convention

The calling convention specifies how values are passed to and from a function call. Our TOY architecture specifies that two arguments are passed in two registers, r0 and r1, while the remaining ones are passed to the stack. This recipe shows you how to define the calling convention, which will be used in `ISelLowering` (the instruction selection lowering phase discussed in *Chapter 6*, *Target Independent Code Generator*) via function pointers.

The calling convention will be defined in the `TOYCallingConv.td` file, which will have primarily two sections—one for defining the return value convention, and the other for defining the argument passing convention. The return value convention specifies how the return values will reside and in which registers. The argument passing convention will specify how the arguments passed will reside and in which registers. The `CallingConv` class is inherited while defining the calling convention of the toy architecture.

How to do it...

To implement the calling convention, proceed with the following steps:

1. Create a new `TOYCallingConv.td` file in the `lib/Target/TOY` folder:

```
$ vi TOYCallingConv.td
```

2. In that file, define the return value convention, as follows:

```
def RetCC_TOY : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[R0]>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;
```

3. Also, define the argument passing convention, like this:

```
def CC_TOY : CallingConv<[
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[R0, R1]>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;
```

4. Define the callee saved register set:

```
def CC_Save : CalleeSavedRegs<(add R2, R3)>;
```

How it works...

In the `.td` file you just read about, it has been specified that the return values of the integer type of 32 bits are stored in the r0 register. Whenever arguments are passed to a function, the first two arguments will be stored in the r0 and r1 registers. It is also specified that whenever any data type, such as an integer of 8 bits or 16 bits, will be encountered, it will be promoted to the 32-bit integer type.

The `tablegen` function generates a `TOYCallingConv.inc` file, which will be referred to in the `TOYISelLowering.cpp` file. The two target `hook` functions used to define argument handling are `LowerFormalArguments()` and `LowerReturn()`.

See also

- To see a detailed implementation of advanced architectures, such as ARM, look into the `lib/Target/ARM/ARMCallingConv.td` file

Defining the instruction set

The instruction set of an architecture varies according to various features present in the architecture. This recipe demonstrates how instruction sets are defined for target architecture.

Three things are defined in the instruction target description file: operands, the assembly string and the instruction pattern. The specification contains a list of definitions or outputs, and a list of uses or inputs. There can be different operand classes, such as the `Register` class, and the immediate and more complex `register + imm` operands.

Here, a simple add instruction definition that takes two registers as operands is demonstrated.

How to do it...

To define an instruction set using target descriptor files, proceed with the following steps.

1. Create a new file called `TOYInstrInfo.td` in the `lib/Target/TOY` folder:

   ```
   $ vi TOYInstrInfo.td
   ```

2. Specify the operands, assembly string, and instruction pattern for the `add` instruction between two register operands:

   ```
   def ADDrr : InstTOY<(outs GRRegs:$dst),
                       (ins GRRegs:$src1, GRRegs:$src2),
                        "add $dst, $src1,z$src2",
   [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
   ```

How it works...

The `add` register to the register instruction specifies `$dst` as the result operand, which belongs to the `General Register` type class; inputs `$src1` and `$src2` as two input operands, which also belong to the `General Register` type class; and the instruction assembly string as `"add $dst, $src1, $src2"` of the 32-bit integer type.

So, an assembly will be generated for `add` between two registers, like this:

```
add r0, r0, r1
```

The preceding code indicates to add the r0 and r1 register contents and store the result in the r0 register.

See also

- Many instructions will have the same type of instruction pattern—ALU instructions such as `add`, `sub`, and so on. In cases such as this multiclass can be used to define the common properties. For more detailed information about the various types of instruction sets for advanced architecture, such as ARM, refer to the `lib/Target/ARM/ARMInstrInfo.td file`

Implementing frame lowering

This recipe talks about frame lowering for target architecture. Frame lowering involves emitting the prologue and epilogue of the function call.

Getting ready

Two functions need to be defined for frame lowering, namely `TOYFrameLowering::emitPrologue()` and `TOYFrameLowering::emitEpilogue()`.

How to do it...

The following functions are defined in the `TOYFrameLowering.cpp` file in the `lib/Target/TOY` folder:

1. The `emitPrologue` function can be defined as follows:

```
void TOYFrameLowering::emitPrologue(MachineFunction &MF)
const {
  const TargetInstrInfo &TII =
*MF.getSubtarget().getInstrInfo();
  MachineBasicBlock &MBB = MF.front();
```

```
  MachineBasicBlock::iterator MBBI = MBB.begin();
  DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() :
  DebugLoc();
  uint64_t StackSize = computeStackSize(MF);
  if (!StackSize) {
    return;
  }
  unsigned StackReg = TOY::SP;
  unsigned OffsetReg = materializeOffset(MF, MBB, MBBI,
  (unsigned)StackSize);
  if (OffsetReg) {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::SUBrr), StackReg)
        .addReg(StackReg)
        .addReg(OffsetReg)
        .setMIFlag(MachineInstr::FrameSetup);
  } else {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::SUBri), StackReg)
        .addReg(StackReg)
        .addImm(StackSize)
        .setMIFlag(MachineInstr::FrameSetup);
  }
}
```

2. The `emitEpilogue` function can be defined like this:

```
void TOYFrameLowering::emitEpilogue(MachineFunction &MF,
                                    MachineBasicBlock &MBB)
  const {

  const TargetInstrInfo &TII =
  *MF.getSubtarget().getInstrInfo();
MachineBasicBlock::iterator MBBI =
MBB.getLastNonDebugInstr();
  DebugLoc dl = MBBI->getDebugLoc();
  uint64_t StackSize = computeStackSize(MF);
  if (!StackSize) {
    return;
  }
  unsigned StackReg = TOY::SP;
  unsigned OffsetReg = materializeOffset(MF, MBB, MBBI,
  (unsigned)StackSize);
  if (OffsetReg) {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::ADDrr), StackReg)
        .addReg(StackReg)
        .addReg(OffsetReg)
        .setMIFlag(MachineInstr::FrameSetup);
```

```
  } else {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::ADDri), StackReg)
        .addReg(StackReg)
        .addImm(StackSize)
        .setMIFlag(MachineInstr::FrameSetup);
  }
}
```

3. Here are some helper functions used to determine the offset for the `ADD` stack operation:

```
static unsigned materializeOffset(MachineFunction &MF,
MachineBasicBlock &MBB, MachineBasicBlock::iterator MBBI,
unsigned Offset) {
  const TargetInstrInfo &TII =
  *MF.getSubtarget().getInstrInfo();
  DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() :
  DebugLoc();
  const uint64_t MaxSubImm = 0xfff;
  if (Offset <= MaxSubImm) {
    return 0;
  } else {
    unsigned OffsetReg = TOY::R2;
    unsigned OffsetLo = (unsigned)(Offset & 0xffff);
    unsigned OffsetHi = (unsigned)((Offset & 0xffff0000) >>
  16);
    BuildMI(MBB, MBBI, dl, TII.get(TOY::MOVLOi16),
  OffsetReg)
        .addImm(OffsetLo)
        .setMIFlag(MachineInstr::FrameSetup);
    if (OffsetHi) {
      BuildMI(MBB, MBBI, dl, TII.get(TOY::MOVHIi16),
  OffsetReg)
          .addReg(OffsetReg)
          .addImm(OffsetHi)
          .setMIFlag(MachineInstr::FrameSetup);
    }
    return OffsetReg;
  }
}
```

4. The following are some more helper functions used to compute the stack size:

```
uint64_t TOYFrameLowering::computeStackSize(MachineFunction
&MF) const {
  MachineFrameInfo *MFI = MF.getFrameInfo();
  uint64_t StackSize = MFI->getStackSize();
```

```
  unsigned StackAlign = getStackAlignment();
  if (StackAlign > 0) {
    StackSize = RoundUpToAlignment(StackSize, StackAlign);
  }
  return StackSize;
}
```

How it works...

The `emitPrologue` function first computes the stack size to determine whether the prologue is required at all. Then it adjusts the stack pointer by calculating the offset. For the epilogue, it first checks whether the epilogue is required or not. Then it restores the stack pointer to what it was at the beginning of the function.

For example, consider this input IR:

```
%p = alloca i32, align 4
store i32 2, i32* %p
%b = load i32* %p, align 4
%c = add nsw i32 %a, %b
```

The TOY assembly generated will look like this:

```
sub sp, sp, #4 ; prologue
movw r1, #2
str r1, [sp]
add r0, r0, #2
add sp, sp, #4 ; epilogue
```

See also

- For advanced architecture frame lowering, such as in ARM, refer to the `lib/Target/ARM/ARMFrameLowering.cpp` file.

Printing an instruction

Printing an assembly instruction is an important step in generating target code. Various classes are defined that work as a gateway to the streamers. The instruction string is provided by the `.td` file defined earlier.

Getting ready

The first and foremost step for printing instructions is to define the instruction string in the `.td` file, which was done in the *Defining the instruction set* recipe.

How to do it...

Perform the following steps:

1. Create a new folder called `InstPrinter` inside the `TOY` folder:

```
$ cd lib/Target/TOY
$ mkdir InstPrinter
```

2. In a new file, called `TOYInstrFormats.td`, define the `AsmString` variable:

```
class InstTOY<dag outs, dag ins, string asmstr, list<dag>
pattern>
    : Instruction {
  field bits<32> Inst;
  let Namespace = "TOY";
  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString   = asmstr;
  let Pattern = pattern;
  let Size = 4;
}
```

3. Create a new file called `TOYInstPrinter.cpp`, and define the `printOperand` function, as follows:

```
void TOYInstPrinter::printOperand(const MCInst *MI,
unsigned OpNo, raw_ostream &O) {
  const MCOperand &Op = MI->getOperand(OpNo);
  if (Op.isReg()) {
    printRegName(O, Op.getReg());
    return;
  }

  if (Op.isImm()) {
    O << "#" << Op.getImm();
    return;
  }
  assert(Op.isExpr() && "unknown operand kind in
  printOperand");
  printExpr(Op.getExpr(), O);
}
```

4. Also, define a function to print the register names:

```
void TOYInstPrinter::printRegName(raw_ostream &OS, unsigned
RegNo) const {
  OS << StringRef(getRegisterName(RegNo)).lower();
}
```

5. Define a function to print the instruction:

```
void TOYInstPrinter::printInst(const MCInst *MI,
raw_ostream &O,StringRef Annot) {
  printInstruction(MI, O);
  printAnnotation(O, Annot);
}
```

6. It also requires `MCASMinfo` to be specified to print the instruction. This can be done by defining the `TOYMCAsmInfo.h` and `TOYMCAsmInfo.cpp` files.

   The `TOYMCAsmInfo.h` file can be defined as follows:

```
#ifndef TOYTARGETASMINFO_H
#define TOYTARGETASMINFO_H

#include "llvm/MC/MCAsmInfoELF.h"

namespace llvm {
class StringRef;
class Target;

class TOYMCAsmInfo : public MCAsmInfoELF {
  virtual void anchor();

public:
  explicit TOYMCAsmInfo(StringRef TT);
};

} // namespace llvm
#endif
```

   The `TOYMCAsmInfo.cpp` file can be defined like this:

```
#include "TOYMCAsmInfo.h"
#include "llvm/ADT/StringRef.h"
using namespace llvm;

void TOYMCAsmInfo::anchor() {}

TOYMCAsmInfo::TOYMCAsmInfo(StringRef TT) {
  SupportsDebugInformation = true;
  Data16bitsDirective = "\t.short\t";
  Data32bitsDirective = "\t.long\t";
  Data64bitsDirective = 0;
  ZeroDirective = "\t.space\t";
  CommentString = "#";
```

```
    AscizDirective = ".asciiz";

    HiddenVisibilityAttr = MCSA_Invalid;
    HiddenDeclarationVisibilityAttr = MCSA_Invalid;
    ProtectedVisibilityAttr = MCSA_Invalid;
  }
```

7. Define the `LLVMBuild.txt` file for the instruction printer:

```
[component_0]
type = Library
name = TOYAsmPrinter
parent = TOY
required_libraries = MC Support
add_to_library_groups = TOY
```

8. Define `CMakeLists.txt`:

```
add_llvm_library(LLVMTOYAsmPrinter
  TOYInstPrinter.cpp
  )
```

How it works...

When the final compilation takes place, the **llc** tool—a static compiler—will generate the assembly of the TOY architecture.

For example, the following IR, when given to the llc tool, will generate an assembly as shown:

```
target datalayout = "e-m:e-p:32:32-i1:8:32-i8:8:32-
i16:16:32-i64:32-f64:32-a:0:32-n32"
target triple = "toy"
define i32 @foo(i32 %a, i32 %b) {
   %c = add nsw i32 %a, %b
   ret i32 %c
}

$ llc foo.ll
.text
.file "foo.ll"
.globl foo
.type foo,@function
foo:     # @foo
# BB#0:   # %entry
add r0, r0, r1
```

```
b lr
.Ltmp0:
.size foo, .Ltmp0-foo
```

Selecting an instruction

An IR instruction in DAG needs to be lowered to a target-specific instruction. The SDAG node contains IR, which needs to be mapped on machine-specific DAG nodes. The outcome of the selection phase is ready for scheduling.

Getting ready

1. For selecting a machine-specific instruction, a separate class, `TOYDAGToDAGISel`, needs to be defined. To compile the file containing this class definition, add the filename to the `CMakeLists.txt` file in the `TOY` folder:

```
$ vi CMakeLists .txt
add_llvm_target(...
...
TOYISelDAGToDAG.cpp
...
)
```

2. A pass entry needs to be added in the `TOYTargetMachine.h` and `TOYTargetMachine.cpp` files:

```
$ vi TOYTargetMachine.h
const TOYInstrInfo *getInstrInfo() const override {
return getSubtargetImpl()->getInstrInfo();
}
```

3. The following code in `TOYTargetMachine.cpp` will create a pass in the instruction selection stage:

```
class TOYPassConfig : public TargetPassConfig {
public:
...
virtual bool addInstSelector();
};
...
bool TOYPassConfig::addInstSelector() {
addPass(createTOYISelDag(getTOYTargetMachine()));
return false;
}
```

How to do it...

To define an instruction selection function, proceed with the following steps:

1. Create a file called `TOYISelDAGToDAG.cpp`:

   ```
   $ vi TOYISelDAGToDAG.cpp
   ```

2. Include the following files:

   ```
   #include "TOY.h"
   #include "TOYTargetMachine.h"
   #include "llvm/CodeGen/SelectionDAGISel.h"
   #include "llvm/Support/Compiler.h"
   #include "llvm/Support/Debug.h"
   #include "TOYInstrInfo.h"
   ```

3. Define a new class called `TOYDAGToDAGISel` as follows, which will inherit from the `SelectionDAGISel` class:

   ```
   class TOYDAGToDAGISel : public SelectionDAGISel {
     const TOYSubtarget &Subtarget;

   public:
     explicit TOYDAGToDAGISel(TOYTargetMachine &TM,
     CodeGenOpt::Level OptLevel)
   : SelectionDAGISel(TM, OptLevel),   Subtarget(*TM.
   getSubtargetImpl()) {}
   };
   ```

4. The most important function to define in this class is `Select()`, which will return an `SDNode` object specific to the machine instruction:

   Declare it in the class:

   ```
   SDNode *Select(SDNode *N);
   ```

   Define it further as follows:

   ```
   SDNode *TOYDAGToDAGISel::Select(SDNode *N) {
     return SelectCode(N);
   }
   ```

5. Another important function is used to define the address selection function, which will calculate the base and offset of the address for load and store operations.

   Declare it as shown here:

   ```
     bool SelectAddr(SDValue Addr, SDValue &Base, SDValue
   &Offset);
   ```

Define it further, like this:

```
bool TOYDAGToDAGISel::SelectAddr(SDValue Addr, SDValue
&Base, SDValue &Offset) {
  if (FrameIndexSDNode *FIN =
dyn_cast<FrameIndexSDNode>(Addr)) {
        Base = CurDAG->getTargetFrameIndex(FIN->getIndex(),
                                           getTargetLowering()-
  >getPointerTy());
        Offset = CurDAG->getTargetConstant(0, MVT::i32);
        return true;
    }
  if (Addr.getOpcode() == ISD::TargetExternalSymbol ||
        Addr.getOpcode() == ISD::TargetGlobalAddress ||
        Addr.getOpcode() == ISD::TargetGlobalTLSAddress) {
        return false; // direct calls.
  }

    Base = Addr;
    Offset = CurDAG->getTargetConstant(0, MVT::i32);
    return true;
}
```

6. The `createTOYISelDag` pass converts a legalized DAG into a toy-specific DAG, ready for instruction scheduling in the same file:

```
FunctionPass *llvm::createTOYISelDag(TOYTargetMachine &TM,
CodeGenOpt::Level OptLevel) {
return new TOYDAGToDAGISel(TM, OptLevel);
}
```

How it works...

The `TOYDAGToDAGISel::Select()` function of `TOYISelDAGToDAG.cpp` is used for the selection of the OP code DAG node, while `TOYDAGToDAGISel::SelectAddr()` is used for the selection of the DATA DAG node with the `addr` type. Note that if the address is global or external, we return false for the address, since its address is calculated in the global context.

See also

- For details on the selection of DAG for machine instructions of complex architectures, such as ARM, look into the `lib/Target/ARM/ARMISelDAGToDAG.cpp` file in the LLVM source code.

Adding instruction encoding

If the instructions need to be specific for how they are encoded with respect to bit fields, this can be done by specifying the bit field in the `.td` file when defining an instruction.

How to do it...

To include instruction encoding while defining instructions, proceed with the following steps:

1. A register operand that will be used to register the `add` instruction will have some defined encoding for its instruction. The size of the instruction is 32 bits, and the encoding for it is as follows:

```
bits 0 to 3 -> src2, second register operand
bits 4 to 11 -> all zeros
bits 12 to 15 -> dst, for destination register
bits 16 to 19 -> src1, first register operand
bit 20 -> zero
bit 21 to 24 -> for opcode
bit 25 to 27 -> all zeros
bit 28 to 31 -> 1110
```

   This can be achieved by specifying the preceding bit pattern in the `.td` files

2. In the `TOYInstrFormats.td` file, define a new variable, called `Inst`:

```
class InstTOY<dag outs, dag ins, string asmstr, list<dag>
pattern>
      : Instruction {
  field bits<32> Inst;

  let Namespace = "TOY";
    …
   …
    let AsmString   = asmstr;
   …
 …
 }
```

3. In the `TOYInstrInfo.td` file, define an instruction encoding:

```
def ADDrr : InstTOY<(outs GRRegs:$dst),(ins GRRegs:$src1,
GRRegs:$src2) ... > {
bits<4> src1;
bits<4> src2;
bits<4> dst;
let Inst{31-25} = 0b1100000;
```

```
let Inst{24-21} = 0b1100; // Opcode
let Inst{20} = 0b0;
let Inst{19-16} = src1; // Operand 1
let Inst{15-12} = dst; // Destination
let Inst{11-4} = 0b00000000;
let Inst{3-0} = src2;
}
```

4. In the `TOY/MCTargetDesc` folder, in the `TOYMCCodeEmitter.cpp` file, the encoding function will be called if the machine instruction operand is a register:

```
unsigned TOYMCCodeEmitter::getMachineOpValue(const MCInst
&MI,
                                             const
  MCOperand &MO,

SmallVectorImpl<MCFixup> &Fixups,
                                             const
  MCSubtargetInfo &STI) const {
    if (MO.isReg()) {
      return CTX.getRegisterInfo()-
  >getEncodingValue(MO.getReg());
  }
```

5. Also, in the same file, a function used to encode the instruction is specified:

```
void TOYMCCodeEmitter::EncodeInstruction(const MCInst &MI,
raw_ostream &OS, SmallVectorImpl<MCFixup> &Fixups, const
MCSubtargetInfo &STI) const {
      const MCInstrDesc &Desc = MCII.get(MI.getOpcode());
      if (Desc.getSize() != 4) {
        llvm_unreachable("Unexpected instruction size!");
  }

      const uint32_t Binary = getBinaryCodeForInstr(MI,
Fixups, STI);

  EmitConstant(Binary, Desc.getSize(), OS);
 ++MCNumEmitted;
}
```

How it works...

In the `.td` files, the encoding of an instruction has been specified—the bits for the operands, the destination, flag conditions, and opcode of the instruction. The machine code emitter gets these encodings from the `.inc` file generated by `tablegen` from the `.td` files through function calls. It encodes these instructions and emits the same for instruction printing.

See also

- For complex architecture such as ARM, see the `ARMInstrInfo.td` and `ARMInstrInfo.td` files in the `lib/Target/ARM` directory of the LLVM trunk

Supporting a subtarget

A target may have a subtarget—typically, a variant with instructions—way of handling operands, among others. This subtarget feature can be supported in the LLVM backend. A subtarget may contain some additional instructions, registers, scheduling models, and so on. ARM has subtargets such as NEON and THUMB, while x86 has subtarget features such as SSE, AVX, and so on. The instruction set differs for the subtarget feature, for example, NEON for ARM and SSE/AVX for subtarget features that support vector instructions. SSE and AVX also support the vector instruction set, but their instructions differ from each other.

How to do it...

This recipe will demonstrate how to add a support subtarget feature in the backend. A new class that will inherit the `TargetSubtargetInfo` class has to be defined:

1. Create a new file called `TOYSubtarget.h`:

```
$ vi TOYSubtarget.h
```

2. Include the following files:

```
#include "TOY.h"
#include "TOYFrameLowering.h"
#include "TOYISelLowering.h"
#include "TOYInstrInfo.h"
#include "TOYSelectionDAGInfo.h"
#include "TOYSubtarget.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetSubtargetInfo.h"
#include "TOYGenSubtargetInfo.inc"
```

3. Define a new class, called `TOYSubtarget`, with some private members that have information on the data layout, target lowering, target selection DAG, target frame lowering, and so on:

```
class TOYSubtarget : public TOYGenSubtargetInfo {
  virtual void anchor();

private:
```

```
  const DataLayout DL;       // Calculates type size & alignment.
  TOYInstrInfo InstrInfo;
  TOYTargetLowering TLInfo;
  TOYSelectionDAGInfo TSInfo;
  TOYFrameLowering FrameLowering;
  InstrItineraryData InstrItins;
```

4. Declare its constructor:

```
TOYSubtarget(const std::string &TT, const std::string &CPU,
const std::string &FS, TOYTargetMachine &TM);
```

This constructor initializes the data members to match that of the specified triplet.

5. Define some helper functions to return the class-specific data:

```
const InstrItineraryData *getInstrItineraryData() const override {
  return &InstrItins;
}

const TOYInstrInfo *getInstrInfo() const override { return
&InstrInfo; }

const TOYRegisterInfo *getRegisterInfo() const override {
  return &InstrInfo.getRegisterInfo();
}

const TOYTargetLowering *getTargetLowering() const override {
  return &TLInfo;
}

const TOYFrameLowering *getFrameLowering() const override {
  return &FrameLowering;
}

const TOYSelectionDAGInfo *getSelectionDAGInfo() const override {
  return &TSInfo;
}

const DataLayout *getDataLayout() const override { return &DL; }

void ParseSubtargetFeatures(StringRef CPU, StringRef FS);

TO LC,

Please maintain the representation of the above code EXACTLY as
seen above.
```

6. Create a new file called `TOYSubtarget.cpp`, and define the constructor as follows:

```
TOYSubtarget::TOYSubtarget(const std::string &TT, const
std::string &CPU, const std::string &FS, TOYTargetMachine &TM)
      DL("e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32-
  f64:32-a:0:32-n32"),
      InstrInfo(), TLInfo(TM), TSInfo(DL), FrameLowering()
  {}
```

The subtarget has its own data layout defined, with other information such as frame lowering, instruction information, subtarget information, and so on.

See also

- To dive into the details of subtarget implementation, refer to the `lib/Target/ARM/ARMSubtarget.cpp` file in the LLVM source code

Lowering to multiple instructions

Let's take an example of implementing a 32-bit immediate load with high/low pairs, where MOVW implies moving a 16-bit low immediate and a clear 16 high bit, and MOVT implies moving a 16-bit high immediate.

How to do it...

There can be various ways to implement this multiple instruction lowering. We can do this by using pseudo-instructions or in the selection DAG-to-DAG phase.

1. To do it without pseudo-instructions, define some constraints. The two instructions must be ordered. MOVW clears the high 16 bits. Its output is read by MOVT to fill the high 16 bits. This can be done by specifying the constraints in tablegen:

```
def MOVLOi16 : MOV<0b1000, "movw", (ins i32imm:$imm),
                   [(set i32:$dst, i32imm_lo:$imm)]>;
def MOVHIi16 : MOV<0b1010, "movt", (ins GRRegs:$src1,
i32imm:$imm),
                   [/* No Pattern */]>;
```

   The second way is to define a pseudo-instruction in the `.td` file:

```
def MOVi32 : InstTOY<(outs GRRegs:$dst), (ins i32imm:$src), "",
[(set i32:$dst, (movei32 imm:$src))]> {
  let isPseudo = 1;
}
```

2. The pseudo-instruction is then lowered by a target function in the `TOYInstrInfo.cpp` file:

```
bool
TOYInstrInfo::expandPostRAPseudo(MachineBasicBlock::iterato
r MI) const {
  if (MI->getOpcode() == TOY::MOVi32){
    DebugLoc DL = MI->getDebugLoc();
    MachineBasicBlock &MBB = *MI->getParent();

    const unsigned DstReg = MI->getOperand(0).getReg();
    const bool DstIsDead = MI->getOperand(0).isDead();

    const MachineOperand &MO = MI->getOperand(1);

    auto LO16 = BuildMI(MBB, MI, DL, get(TOY::MOVLOi16),
  DstReg);
    auto HI16 = BuildMI(MBB, MI, DL, get(TOY::MOVHIi16))
                    .addReg(DstReg, RegState::Define |
  getDeadRegState(DstIsDead))
                    .addReg(DstReg);

  MBB.erase(MI);
    return true;
  }
}
```

3. Compile the entire LLVM project:

   For example, an `ex.ll` file with IR will look like this:

```
define i32 @foo(i32 %a) #0 {
%b = add nsw i32 %a, 65537 ; 0x00010001
ret i32 %b
}
```

   The assembly generated will look like this:

```
movw r1, #1
movt r1, #1
add r0, r0, r1
b lr
```

How it works...

The first instruction, `movw`, will move 1 in the lower 16 bits and clear the high 16 bits. So in r1, `0x00000001` will be written by the first instruction. In the next instruction, `movt`, the higher 16 bits will be written. So in r1, `0x0001XXXX` will be written, without disturbing the lower bits. Finally, the r1 register will have `0x00010001` in it. Whenever a pseudo-instruction is encountered as specified in the `.td` file, its expand function is called to specify what the pseudo-instruction will expand to.

In the preceding case, the `mov32` immediate was to be implemented by two instructions: `movw` (the lower 16 bits) and `movt` (the higher 16 bits). It was marked as a pseudo-instruction in the `.td` file. When this pseudo-instruction needs to be emitted, its expand function is called, which builds two machine instructions: `MOVLOi16` and `MOVHIi16`. These map to the `movw` and `movt` instructions of the target architecture.

See also

- To dive deep into implementing such lowering of multiple instructions, look at the ARM target implementation in the LLVM source code in the `lib/Target/ARM/ARMInstrInfo.td` file.

Registering a target

For running the llc tool in the TOY target architecture, it has to be registered with the llc tool. This recipe demonstrates which configuration files need to be modified to register a target. The build files are modified in this recipe.

How to do it...

To register a target with a static compiler, follow these steps:

1. First, add the entry of the TOY backend to `llvm_root_dir/CMakeLists.txt`:

```
set(LLVM_ALL_TARGETS
  AArch64
  ARM
  …
  …
  TOY
  )
```

2. Then add the toy entry to `llvm_root_dir/include/llvm/ADT/Triple.h`:

```
class Triple {
public:
```

```
  enum ArchType {
    UnknownArch,

    arm,        // ARM (little endian): arm, armv.*, xscale
    armeb,      // ARM (big endian): armeb
    aarch64,    // AArch64 (little endian): aarch64
    …
   …

toy     // TOY: toy
};
```

3. Add the toy entry to `llvm_root_dir/include/llvm/ MC/MCExpr.h`:

```
class MCSymbolRefExpr : public MCExpr {
public:
enum VariantKind {
...
VK_TOY_LO,
VK_TOY_HI,
};
```

4. Add the toy entry to `llvm_root_dir/include/llvm/ Support/ELF.h`:

```
enum {
  EM_NONE          = 0, // No machine
  EM_M32           = 1, // AT&T WE 32100
  …
  …
  EM_TOY           = 220 // whatever is the next number
};
```

5. Then, add the toy entry to `lib/MC/MCExpr.cpp`:

```
StringRef MCSymbolRefExpr::getVariantKindName(VariantKind
Kind) {
switch (Kind) {

  …
  …
  case VK_TOY_LO: return "TOY_LO";
  case VK_TOY_HI: return "TOY_HI";
  }
…
}
```

6. Next, add the toy entry to `lib/Support/Triple.cpp`:

```
const char *Triple::getArchTypeName(ArchType Kind) {
  switch (Kind) {
 …
 …
 case toy:         return "toy";

}

const char *Triple::getArchTypePrefix(ArchType Kind) {
  switch (Kind) {
 …
 …
case toy:         return "toy";
  }
}

Triple::ArchType Triple::getArchTypeForLLVMName(StringRef
Name) {
…
…
    .Case("toy", toy)
…
}

static Triple::ArchType parseArch(StringRef ArchName) {
…
…
    .Case("toy", Triple::toy)
…
}


static unsigned
getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
…
…
case llvm::Triple::toy:
    return 32;

…
…
}
```

```
Triple Triple::get32BitArchVariant() const {
…
…
case Triple::toy:
    // Already 32-bit.
    break;
…
}

Triple Triple::get64BitArchVariant() const {
…
…
case Triple::toy:
    T.setArch(UnknownArch);
    break;

…
…
}
```

7. Add the toy directory entry to `lib/Target/LLVMBuild.txt`:

```
[common]
subdirectories = ARM AArch64 CppBackend Hexagon MSP430 … …
TOY
```

8. Create a new file called `TOY.h` in the `lib/Target/TOY` folder:

```
#ifndef TARGET_TOY_H
#define TARGET_TOY_H

#include "MCTargetDesc/TOYMCTargetDesc.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
class TargetMachine;
class TOYTargetMachine;

FunctionPass *createTOYISelDag(TOYTargetMachine &TM,
                               CodeGenOpt::Level OptLevel);
} // end namespace llvm;

#endif
```

9. Create a new folder called `TargetInfo` in the `lib/Target/TOY` folder. Inside that folder, create a new file called `TOYTargetInfo.cpp`, as follows:

```
#include "TOY.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

Target llvm::TheTOYTarget;

extern "C" void LLVMInitializeTOYTargetInfo() {
  RegisterTarget<Triple::toy> X(TheTOYTarget, "toy",
  "TOY");
}
```

10. In the same folder, create the `CMakeLists.txt` file:

```
add_llvm_library(LLVMTOYInfo
  TOYTargetInfo.cpp
  )
```

11. Create an `LLVMBuild.txt` file:

```
[component_0]
type = Library
name = TOYInfo
parent = TOY
required_libraries = Support
add_to_library_groups = TOY
```

12. In the `lib/Target/TOY` folder, create a file called `TOYTargetMachine.cpp`:

```
#include "TOYTargetMachine.h"
#include "TOY.h"
#include "TOYFrameLowering.h"
#include "TOYInstrInfo.h"
#include TOYISelLowering.h"
#include "TOYSelectionDAGInfo.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/IR/Module.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

TOYTargetMachine::TOYTargetMachine(const Target &T, StringRef TT,
StringRef CPU, StringRef FS, const TargetOptions &Options,
Reloc::Model RM, CodeModel::Model CM,
                                   CodeGenOpt::Level OL)
```

```
    : LLVMTargetMachine(T, TT, CPU, FS, Options, RM, CM,
  OL),
      Subtarget(TT, CPU, FS, *this) {
  initAsmInfo();
}

namespace {
class TOYPassConfig : public TargetPassConfig {
public:
  TOYPassConfig(TOYTargetMachine *TM, PassManagerBase &PM)
      : TargetPassConfig(TM, PM) {}

  TOYTargetMachine &getTOYTargetMachine() const {
    return getTM<TOYTargetMachine>();
  }

  virtual bool addPreISel();
  virtual bool addInstSelector();
  virtual bool addPreEmitPass();
};
} // namespace

TargetPassConfig
*TOYTargetMachine::createPassConfig(PassManagerBase &PM) {
  return new TOYPassConfig(this, PM);
}

bool TOYPassConfig::addPreISel() { return false; }

bool TOYPassConfig::addInstSelector() {
  addPass(createTOYISelDag(getTOYTargetMachine(),
  getOptLevel()));
  return false;
}

bool TOYPassConfig::addPreEmitPass() { return false; }

// Force static initialization.
extern "C" void LLVMInitializeTOYTarget() {
  RegisterTargetMachine<TOYTargetMachine> X(TheTOYTarget);
}

void TOYTargetMachine::addAnalysisPasses(PassManagerBase
&PM) {}
```

13. Create a new folder called `MCTargetDesc` and a new file called `TOYMCTargetDesc.h`:

```
#ifndef TOYMCTARGETDESC_H
#define TOYMCTARGETDESC_H

#include "llvm/Support/DataTypes.h"

namespace llvm {
class Target;
class MCInstrInfo;
class MCRegisterInfo;
class MCSubtargetInfo;
class MCContext;
class MCCodeEmitter;
class MCAsmInfo;
class MCCodeGenInfo;
class MCInstPrinter;
class MCObjectWriter;
class MCAsmBackend;

class StringRef;
class raw_ostream;

extern Target TheTOYTarget;

MCCodeEmitter *createTOYMCCodeEmitter(const MCInstrInfo &MCII,
const MCRegisterInfo &MRI, const MCSubtargetInfo &STI, MCContext
&Ctx);

MCAsmBackend *createTOYAsmBackend(const Target &T, const
MCRegisterInfo &MRI, StringRef TT, StringRef   CPU);

MCObjectWriter *createTOYELFObjectWriter(raw_ostream &OS,
uint8_t OSABI);

} // End llvm namespace

#define GET_REGINFO_ENUM
```

```
#include "TOYGenRegisterInfo.inc"

#define GET_INSTRINFO_ENUM
#include "TOYGenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "TOYGenSubtargetInfo.inc"

#endif
```

14. Create one more file, called `TOYMCTargetDesc.cpp`, in the same folder:

```
#include "TOYMCTargetDesc.h"
#include "InstPrinter/TOYInstPrinter.h"
#include "TOYMCAsmInfo.h"
#include "llvm/MC/MCCodeGenInfo.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/TargetRegistry.h"

#define GET_INSTRINFO_MC_DESC
#include "TOYGenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "TOYGenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "TOYGenRegisterInfo.inc"

using namespace llvm;

static MCInstrInfo *createTOYMCInstrInfo() {
  MCInstrInfo *X = new MCInstrInfo();
  InitTOYMCInstrInfo(X);
  return X;
}

static MCRegisterInfo *createTOYMCRegisterInfo(StringRef
TT) {
  MCRegisterInfo *X = new MCRegisterInfo();
  InitTOYMCRegisterInfo(X, TOY::LR);
```

```
  return X;
}

static MCSubtargetInfo *createTOYMCSubtargetInfo(StringRef
TT, StringRef CPU,
                                                 StringRef
  FS) {
  MCSubtargetInfo *X = new MCSubtargetInfo();
  InitTOYMCSubtargetInfo(X, TT, CPU, FS);
  return X;
}

static MCAsmInfo *createTOYMCAsmInfo(const MCRegisterInfo
&MRI, StringRef TT) {
  MCAsmInfo *MAI = new TOYMCAsmInfo(TT);
  return MAI;
}

static MCCodeGenInfo *createTOYMCCodeGenInfo(StringRef TT,
Reloc::Model RM,
                                             CodeModel::Model CM,
                                             CodeGenOpt::Level OL)
{
  MCCodeGenInfo *X = new MCCodeGenInfo();
  if (RM == Reloc::Default) {
    RM = Reloc::Static;
  }
  if (CM == CodeModel::Default) {
    CM = CodeModel::Small;
  }
  if (CM != CodeModel::Small && CM != CodeModel::Large) {
    report_fatal_error("Target only supports CodeModel
  Small or Large");
  }

  X->InitMCCodeGenInfo(RM, CM, OL);
  return X;
}

static MCInstPrinter *
createTOYMCInstPrinter(const Target &T, unsigned
SyntaxVariant,
                       const MCAsmInfo &MAI, const
  MCInstrInfo &MII,
```

```
                        const MCRegisterInfo &MRI, const
  MCSubtargetInfo &STI) {
  return new TOYInstPrinter(MAI, MII, MRI);
}

static MCStreamer *
createMCAsmStreamer(MCContext &Ctx, formatted_raw_ostream
&OS, bool isVerboseAsm, bool useDwarfDirectory,MCInstPrinter
*InstPrint, MCCodeEmitter *CE,MCAsmBackend *TAB, bool ShowInst) {
  return createAsmStreamer(Ctx, OS, isVerboseAsm,
useDwarfDirectory, InstPrint,  CE,   TAB, ShowInst);
}

static MCStreamer *createMCStreamer(const Target &T,
StringRef TT,
                                    MCContext &Ctx,
  MCAsmBackend &MAB,
                                    raw_ostream &OS,
                                    MCCodeEmitter *Emitter,
                                    const MCSubtargetInfo
  &STI,
                                    bool RelaxAll,
                                    bool NoExecStack) {
  return createELFStreamer(Ctx, MAB, OS, Emitter, false,
  NoExecStack);
}


// Force static initialization.
extern "C" void LLVMInitializeTOYTargetMC() {
  // Register the MC asm info.
  RegisterMCAsmInfoFn X(TheTOYTarget, createTOYMCAsmInfo);

  // Register the MC codegen info.
  TargetRegistry::RegisterMCCodeGenInfo(TheTOYTarget,
  createTOYMCCodeGenInfo);

  // Register the MC instruction info.
```

```
  TargetRegistry::RegisterMCInstrInfo(TheTOYTarget,
  createTOYMCInstrInfo);

  // Register the MC register info.
  TargetRegistry::RegisterMCRegInfo(TheTOYTarget,
  createTOYMCRegisterInfo);

  // Register the MC subtarget info.
  TargetRegistry::RegisterMCSubtargetInfo(TheTOYTarget,

createTOYMCSubtargetInfo);

  // Register the MCInstPrinter
  TargetRegistry::RegisterMCInstPrinter(TheTOYTarget,
  createTOYMCInstPrinter);

  // Register the ASM Backend.
  TargetRegistry::RegisterMCAsmBackend(TheTOYTarget,
  createTOYAsmBackend);

  // Register the assembly streamer.
  TargetRegistry::RegisterAsmStreamer(TheTOYTarget,
  createMCAsmStreamer);

  // Register the object streamer.
  TargetRegistry::RegisterMCObjectStreamer(TheTOYTarget,
  createMCStreamer);

  // Register the MCCodeEmitter
  TargetRegistry::RegisterMCCodeEmitter(TheTOYTarget,
  createTOYMCCodeEmitter);
}
```

15. In the same folder, create an `LLVMBuild.txt` file:

```
[component_0]
type = Library
name = TOYDesc
parent = TOY
required_libraries = MC Support TOYAsmPrinter TOYInfo
add_to_library_groups = TOY
```

16. Create a `CMakeLists.txt` file:

```
add_llvm_library(LLVMTOYDesc
  TOYMCTargetDesc.cpp)
```

How it works...

Build the enitre LLVM project, as follows:

```
$ cmake llvm_src_dir -DCMAKE_BUILD_TYPE=Release -
DLLVM_TARGETS_TO_BUILD="TOY"
$ make
```

Here, we have specified that we are building the LLVM compiler for the toy target. After the build completes, check whether the TOY target appears with the `llc` command:

```
$ llc -version
…
…
Registered Targets :
toy - TOY
```

See also

- For a more detailed description about complex targets that involve pipelining and scheduling, follow the chapters in *Tutorial: Creating an LLVM Backend for the Cpu0 Architecture* by Chen Chung-Shu and Anoushe Jamshidi

9
Using LLVM for Various Useful Projects

In this chapter, we will cover the following recipes:

- Exception handling in LLVM
- Using sanitizers
- Writing the garbage collector with LLVM
- Converting LLVM IR to JavaScript
- Using the Clang Static Analyzer
- Using bugpoint
- Using LLDB
- Using LLVM utility passes

Introduction

Until now, you have learned how to write the frontend of a compiler, write optimizations and create a backend. In this chapter, the last of this book, we will look into some other features that the LLVM infrastructure provides and how we can use them in our projects. We won't be diving very deep into the details of the topics in this chapter. The main point is to let you know about these important tools and techniques, which are hot points in LLVM.

Exception handling in LLVM

In this recipe, we will look into the exception handling infrastructure of LLVM. We will discuss how the exception handling information looks in the IR and the intrinsic functions provided by LLVM for exception handling.

Getting ready...

You must understand how exception handling works normally and the concepts of `try`, `catch` and `throw` and so on. You must also have Clang and LLVM installed in your path.

How to do it...

We will take an example to describe how exception handling works in LLVM:

1. Open a file to write down the source code, and enter the source code to test exception handling:

```
$ cat eh.cpp
class Ex1 {};
void throw_exception(int a, int b) {
  Ex1 ex1;
  if (a > b) {
    throw ex1;
  }
}

int test_try_catch() {
  try {
    throw_exception(2, 1);
  }
  catch(...) {
    return 1;
  }
  return 0;
}
```

2. Generate the bitcode file using the following command:

```
$ clang -c eh.cpp -emit-llvm -o eh.bc
```

3. To view the IR on the screen, run the following command, which will give you the output as shown:

```
$ llvm-dis eh.bc -o -
; ModuleID = 'eh.bc'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

%class.Ex1 = type { i8 }

@_ZTVN10__cxxabiv117__class_type_infoE = external global i8*
@_ZTS3Ex1 = linkonce_odr constant [5 x i8] c"3Ex1\00"
@_ZTI3Ex1 = linkonce_odr constant { i8*, i8* } { i8* bitcast
(i8** getelementptr inbounds (i8** @_ZTVN10__cxxabiv117__class_
type_infoE, i64 2) to i8*), i8*
getelementptr inbounds ([5 x i8]* @_ZTS3Ex1, i32 0, i32 0) }

; Function Attrs: uwtable
define void @_Z15throw_exceptionii(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %ex1 = alloca %class.Ex1, align 1
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = icmp sgt i32 %3, %4
  br i1 %5, label %6, label %9

; <label>:6                                       ; preds = %0
  %7 = call i8* @__cxa_allocate_exception(i64 1) #1
  %8 = bitcast i8* %7 to %class.Ex1*
  call void @__cxa_throw(i8* %7, i8* bitcast ({ i8*, i8* }*
@_ZTI3Ex1 to i8*), i8* null) #2
  unreachable

; <label>:9                                       ; preds = %0
  ret void
```

```
}

declare i8* @__cxa_allocate_exception(i64)

declare void @__cxa_throw(i8*, i8*, i8*)

; Function Attrs: uwtable
define i32 @_Z14test_try_catchv() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8*
  %3 = alloca i32
  %4 = alloca i32
  invoke void @_Z15throw_exceptionii(i32 2, i32 1)
          to label %5 unwind label %6

; <label>:5                                       ; preds = %0
  br label %13

; <label>:6                                       ; preds = %0
  %7 = landingpad { i8*, i32 } personality i8* bitcast (i32
(...)* @__gxx_personality_v0 to i8*)
          catch i8* null
  %8 = extractvalue { i8*, i32 } %7, 0
  store i8* %8, i8** %2
  %9 = extractvalue { i8*, i32 } %7, 1
  store i32 %9, i32* %3
  br label %10

; <label>:10                                      ; preds = %6
  %11 = load i8** %2
  %12 = call i8* @__cxa_begin_catch(i8* %11) #1
  store i32 1, i32* %1
  store i32 1, i32* %4
  call void @__cxa_end_catch()
  br label %14
```

```
; <label>:13                                          ; preds = %5
  store i32 0, i32* %1
  br label %14

; <label>:14                                          ; preds =
%13, %10
  %15 = load i32* %1
  ret i32 %15
}

declare i32 @__gxx_personality_v0(...)

declare i8* @__cxa_begin_catch(i8*)

declare void @__cxa_end_catch()

attributes #0 = { uwtable "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-
protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-
float"="false" }
attributes #1 = { nounwind }
attributes #2 = { noreturn }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.6.0 (220636)"}
```

How it works...

In LLVM, if an exception is thrown, the runtime tries its best to find a handler. It tries to find an exception frame corresponding to the function where the exception was thrown. This exception frame contains a reference to the exception table, which contains the implementation—how to handle the exception when a programming language supports exception handling. When the language does not support exception handling, the information on how to unwind the current activation and restore the state of the prior activation is found in this exception frame.

Let's look at the preceding example to see how to generate exception handling code with LLVM.

The `try` block is translated to invoke instruction in LLVM:

```
invoke void @_Z15throw_exceptionii(i32 2, i32 1)
          to label %5 unwind label %6
```

The preceding line tells the compiler how it should handle an exception if the `throw_exception` function throws it. If no exception is thrown, then normal execution will take place through the `%5` label. But if an exception is thrown, it will branch into the `%6` label, which is the landing pad. This corresponds roughly to the `catch` portion of a `try/catch` sequence. When execution resumes at a landing pad, it receives an exception structure and a selector value corresponding to the type of exception thrown. The selector is then used to determine which `catch` function should actually process the exception. In this case, it looks something like this:

```
%7 = landingpad { i8*, i32 } personality i8* bitcast (i32 (...)*
@__gxx_personality_v0 to i8*)
     catch i8* null
```

The `%7` in the preceding code snippet represents the information describing the exception. The `{ i8*, i32 }` part of the code describes the type of information. The `i8*` part of the code represents the exception pointer part, and `i32` is the selector value. In this case, we have only one selector value, as the `catch` function accepts all types of exception objects thrown. The `@__gxx_personality_v0` function is the `personality` function. It receives the context of the exception, an exception structure containing the exception object type and value, and a reference to the exception table for the current function. The personality function for the current compile unit is specified in a common exception frame. In our case, the `@__gxx_personality_v0` function represents the fact that we are dealing with C++ exceptions.

So, the `%8 = extractvalue { i8*, i32 } %7, 0` will represent the exception object, and `%9 = extractvalue { i8*, i32 } %7, 1` represents the selector value.

The following are some noteworthy IR functions:

- `__cxa_throw`: This is a function used to throw an exception
- `__cxa_begin_catch`: This takes an exception structure reference as an argument and returns the value of the exception object
- `__cxa_end_catch`: This locates the most recently caught exception and decrements its handler count, removing the exception from the caught state if this counter goes down to zero

See also

- To understand the exception format used by LLVM, go to `http://llvm.org/docs/ExceptionHandling.html#llvm-code-generation`.

Using sanitizers

You might have used tools such as **Valgrind** for memory debugging. LLVM also provides us with tools for memory debugging, such as the address sanitizer, memory sanitizer, and so on. These tools are very fast compared to Valgrind, even though they are not as mature as Valgrind. Most of these tools are in their experimental stage, so if you want, you can contribute to the open source development of these tools.

Getting ready

To make use of these sanitizers, we need to check out the code for `compiler-rt` from the LLVM SVN:

```
cd llvm/projects
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
```

Build LLVM as we did in *Chapter 1*, *LLVM Design and Use*. By doing so, we get the runtime libraries required.

How to do it...

Now, we will test the address sanitizer on a test code.

1. Write a test case to check the address sanitizer:

```
$ cat asan.c
int main() {
int a[5];
int index = 6;
int retval = a[index];
return retval;
}
```

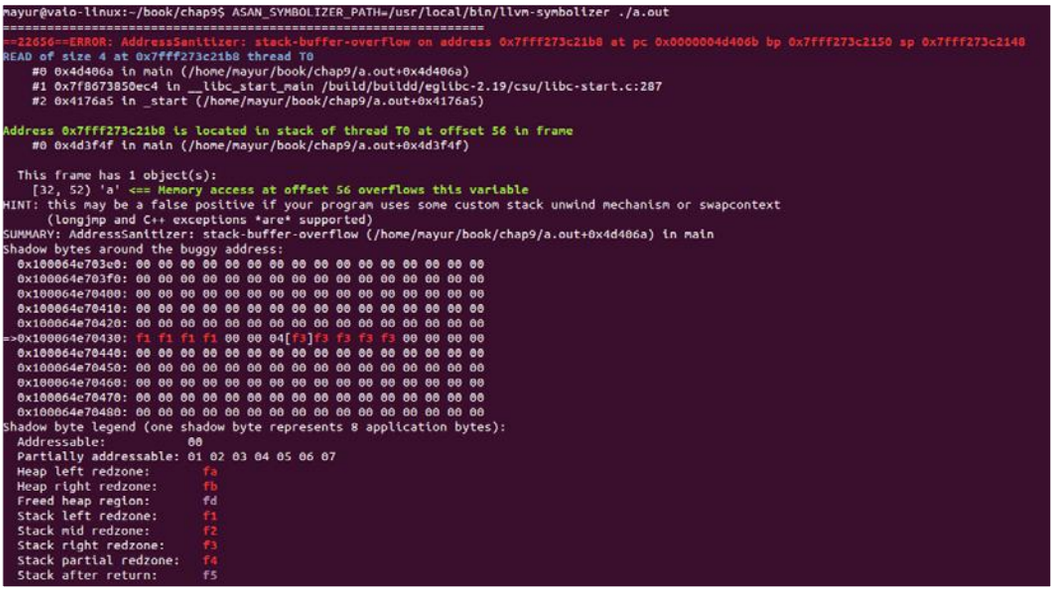
2. Compile the test code using the `fsanitize=address` `command-line argument` for using the address sanitizer:

```
$ clang -fsanitize=address asan.c
```

3. Generate the output of running the address sanitizer using the following command:

```
$ ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
```

Here's the output:



How it works...

The LLVM address sanitizer works on the principle of code instrumentation. The tool consists of a compiler instrumentation module and a runtime library. The code instrumentation part is done by the pass of LLVM, which runs on passing the `fsanitize=address` command-line argument, as is done in the preceding example. The runtime library replaces the `malloc` and `free` functions in the code with custom-made code. Before we go ahead and discuss the details of how code instrumentation is done, here we must know that the virtual address space is divided into two disjointed classes: the main application memory, which is used by the regular application code; and the shadow memory, which contains the shadow values (or metadata).

The shadow memory and the main application memory are linked to each other. Poisoning a byte in the main memory means writing a special value into the corresponding shadow memory.

Let's come back to the address sanitizer; the memory around the regions allocated by the `malloc` function is poisoned. The memory freed by the `free` function is placed in quarantine and is also poisoned. Every memory access in the program is transformed by the compiler in the following way.

At first, it is like this:

```
*address = ...;
```

After transformation, it becomes the following:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

This means that if it finds any invalid access to this memory, it reports an error.

In the preceding example, we wrote a piece of code for a buffer overrun, accessing an array that is out of bounds. Here, the instrumentation of code is done on the address just before and after the array. So, when we access the array beyond its upper bound, we try accessing the red zone. Hence, the address sanitizer gives us a stack buffer overflow report.

See also...

- You can check out the documentation page at `http://clang.llvm.org/docs/AddressSanitizer.html` for more information.
- You can also check out the other sanitizers in LLVM using the following links:

  `http://clang.llvm.org/docs/MemorySanitizer.html`

  `http://clang.llvm.org/docs/ThreadSanitizer.html`

  `https://code.google.com/p/address-sanitizer/wiki/LeakSanitizer`

Writing the garbage collector with LLVM

Garbage collection is a technique of memory management where the collector tries to reclaim the memory occupied by objects that are no longer in use. This frees the programmer from of being required to keep track of the lifetimes of heap objects.

In this recipe, we will see how to integrate LLVM into a compiler for a language that supports garbage collection. LLVM does not itself provide a garbage collector, but provides a framework for describing the garbage collector's requirements to the compiler.

Getting ready

LLVM must be built and installed.

How to do it...

We will see in the following recipe how the LLVM IR code, with garbage collection intrinsic functions, is converted to the corresponding machine assembly code:

1. Write the test code:

```
$ cat testgc.ll

declare i8* @llvm_gc_allocate(i32)
declare void @llvm_gc_initialize(i32)

declare void @llvm.gcroot(i8**, i8*)
declare void @llvm.gcwrite(i8*, i8*, i8**)

define i32 @main() gc "shadow-stack" {
entry:
  %A = alloca i8*
  %B = alloca i8**

  call void @llvm_gc_initialize(i32 1048576)  ; Start with 1MB
heap

        ;; void *A;
  call void @llvm.gcroot(i8** %A, i8* null)

        ;; A = gcalloc(10);
  %Aptr = call i8* @llvm_gc_allocate(i32 10)
  store i8* %Aptr, i8** %A

        ;; void **B;
  %tmp.1 = bitcast i8*** %B to i8**
  call void @llvm.gcroot(i8** %tmp.1, i8* null)

  ;; B = gcalloc(4);
  %B.upgrd.1 = call i8* @llvm_gc_allocate(i32 8)
  %tmp.2 = bitcast i8* %B.upgrd.1 to i8**
  store i8** %tmp.2, i8*** %B
```

```
  ;; *B = A;
  %B.1 = load i8**, i8*** %B
  %A.1 = load i8*, i8** %A
  call void @llvm.gcwrite(i8* %A.1, i8* %B.upgrd.1, i8** %B.1)

  br label %AllocLoop

AllocLoop:
  %i = phi i32 [ 0, %entry ], [ %indvar.next, %AllocLoop ]
        ;; Allocated mem: allocated memory is immediately
dead.
  call i8* @llvm_gc_allocate(i32 100)

  %indvar.next = add i32 %i, 1
  %exitcond = icmp eq i32 %indvar.next, 10000000
  br i1 %exitcond, label %Exit, label %AllocLoop

Exit:
  ret i32 0
}

declare void @__main()
```

2. Use the `llc` tool to generate the assembly code and view the assembly code using the `cat` command:

```
$ llc testgc.ll

$ cat testgc.s
  .text
  .file  "testgc.ll"
  .globl  main
  .align  16, 0x90
  .type  main,@function
main:                                   # @main
.Lfunc_begin0:
```

```
  .cfi_startproc
  .cfi_personality 3, __gcc_personality_v0
  .cfi_lsda 3, .Lexception0
# BB#0:                                 # %entry
  pushq  %rbx
.Ltmp9:
  .cfi_def_cfa_offset 16
  subq  $32, %rsp
.Ltmp10:
  .cfi_def_cfa_offset 48
.Ltmp11:
  .cfi_offset %rbx, -16
  movq  llvm_gc_root_chain(%rip), %rax
  movq  $__gc_main, 8(%rsp)
  movq  $0, 16(%rsp)
  movq  %rax, (%rsp)
  leaq  (%rsp), %rax
  movq  %rax, llvm_gc_root_chain(%rip)
  movq  $0, 24(%rsp)
.Ltmp0:
  movl  $1048576, %edi          # imm = 0x100000
  callq  llvm_gc_initialize
.Ltmp1:
# BB#1:                                 # %entry.cont3
.Ltmp2:
  movl  $10, %edi
  callq  llvm_gc_allocate
.Ltmp3:
# BB#2:                                 # %entry.cont2
  movq  %rax, 16(%rsp)
.Ltmp4:
  movl  $8, %edi
  callq  llvm_gc_allocate
.Ltmp5:
# BB#3:                                 # %entry.cont
```

```
  movq  %rax, 24(%rsp)
  movq  16(%rsp), %rcx
  movq  %rcx, (%rax)
  movl  $10000000, %ebx         # imm = 0x989680
  .align  16, 0x90
.LBB0_4:                                # %AllocLoop
                                        # =>This Inner Loop
Header: Depth=1
.Ltmp6:
  movl  $100, %edi
  callq  llvm_gc_allocate
.Ltmp7:
# BB#5:                                 # %AllocLoop.cont
                                        #   in Loop: Header=BB0_4
Depth=1
  decl  %ebx
  jne  .LBB0_4
# BB#6:                                 # %Exit
  movq  (%rsp), %rax
  movq  %rax, llvm_gc_root_chain(%rip)
  xorl  %eax, %eax
  addq  $32, %rsp
  popq  %rbx
  retq
.LBB0_7:                                # %gc_cleanup
.Ltmp8:
  movq  (%rsp), %rcx
  movq  %rcx, llvm_gc_root_chain(%rip)
  movq  %rax, %rdi
  callq  _Unwind_Resume
.Lfunc_end0:
  .size  main, .Lfunc_end0-main
  .cfi_endproc
  .section  .gcc_except_table,"a",@progbits
  .align  4
```

```
GCC_except_table0:
.Lexception0:
  .byte  255                    # @LPStart Encoding = omit
  .byte  3                      # @TType Encoding = udata4
  .asciz  "\234"                 # @TType base offset
  .byte  3                      # Call site Encoding = udata4
  .byte  26                     # Call site table length
  .long  .Ltmp0-.Lfunc_begin0   # >> Call Site 1 <<
  .long  .Ltmp7-.Ltmp0          #   Call between .Ltmp0 and
.Ltmp7
  .long  .Ltmp8-.Lfunc_begin0   #     jumps to .Ltmp8
  .byte  0                      #   On action: cleanup
  .long  .Ltmp7-.Lfunc_begin0   # >> Call Site 2 <<
  .long  .Lfunc_end0-.Ltmp7     #   Call between .Ltmp7 and
.Lfunc_end0
  .long  0                      #     has no landing pad
  .byte  0                      #   On action: cleanup
  .align  4

  .type  llvm_gc_root_chain,@object # @llvm_gc_root_chain
  .bss
  .weak  llvm_gc_root_chain
  .align  8
llvm_gc_root_chain:
  .quad  0
  .size  llvm_gc_root_chain, 8

  .type  __gc_main,@object      # @__gc_main
  .section  .rodata,"a",@progbits
  .align  8
__gc_main:
  .long  2                      # 0x2
  .long  0                      # 0x0
  .size  __gc_main, 8

  .section  ".note.GNU-stack","",@progbits
```

How it works...

In the preceding code, in the main function, we are using the built-in GC collector strategy called `shadow-stack`, which maintains a linked list of stack `roots()`:

```
define i32 @main() gc "shadow-stack"
```

It mirrors the machine stack. We can provide any other technique, if we want to, by specifying its name after the function name in this format, `gc "strategy name"`. This strategy name can either be the built-in strategy or our own custom strategy for garbage collection.

To identify the roots, that is, the references to the heap object, LLVM makes use of the intrinsic function `@llvm.gcroot` or `the .statepoint` relocation sequence. The `llvm.gcroot` intrinsic function informs LLVM that a stack variable references an object on the heap and it needs to be tracked by the collector. In the preceding code, the following line is the call to the `llvm.gcroot` function to mark the `%tmp.1` stack variable:

```
call void @llvm.gcroot(i8** %tmp.1, i8* null)
```

The `llvm.gcwrite` function is a write barrier. This means that whenever a program on which garbage collection is being done, it writes a pointer to a field of a heap object, the collector is informed about that. The `llvm.gcread` intrinsic function is also present, which informs the garbage collector when the program reads a pointer to a field of a heap object. The following line of code writes the `%A.1` value to the `%B.upgrd` heap object:

```
call void @llvm.gcwrite(i8* %A.1, i8* %B.upgrd.1, i8** %B.1)
```

Note that LLVM does not provide a garbage collector. It should be a part of the runtime library of the language. The preceding explanation deals with the infrastructure that LLVM provides for describing garbage collector requirements to the compiler.

See also

- See `http://llvm.org/docs/GarbageCollection.html` for the documentation on garbage collection.
- Also, check out `http://llvm.org/docs/Statepoints.html` for an alternative method of garbage collection.

Converting LLVM IR to JavaScript

In this recipe, we will briefly discuss how we can convert LLVM IR to JavaScript.

Getting ready

To convert IR to JavaScript, perform the following steps:

1. We will make use of the `emscripten` LLVM to JavaScript compiler. You need to download the SDK provided at `https://kripken.github.io/emscripten-site/docs/getting_started/downloads.html` . You can also build it from the source code, but just for experimenting, you can use the SDK that comes with the toolchain.
2. After downloading the SDK, extract it to a location and go to the root folder of the download.
3. Install the `default-jre`, `nodejs`, `cmake`, `build-essential`, and `git` dependencies.
4. Execute the following commands to install the SDK:

```
./emsdk update
./emsdk install latest
./emsdk activate latest
```

5. See the `~/emscripten` script to check whether it has the correct values, and if not, update it accordingly.

How to do it...

Perform the following steps:

1. Write the test code for the conversion:

```
$ cat test.c
#include<stdio.h>

int main() {
  printf("hi, user!\n");
  return 0;
}
```

2. Convert the code to the LLVM IR:

```
$ clang -S -emit-llvm test.c
```

3. Now use the `emcc` executable located in the `emsdk_portable/emscripten/master` directory to take this `.ll` file as the input and convert it into JavaScript:

```
$ ./emcc test.ll
```

4. The output file generated is the `a.out.js` file. We can execute this file using the following command:

```
$ nodejs a.out.js
hi, user!
```

See more

- To know more details, visit `https://github.com/kripken/emscripten`

Using the Clang Static Analyzer

In this recipe, you will learn about the static analysis of code, which is carried out by the **Clang Static Analyzer**. It is built on top of Clang and LLVM. The static analysis engine used by the Clang Static Analyzer is a Clang library, and it has the capability to be reused in different contexts by different clients.

We will take the example of the divide-by-zero defect and show you how the Clang Static Analyzer handles this defect.

Getting ready

You need to build and install LLVM along with Clang.

How to do it...

Perform the following steps:

1. Create a test file and write the test code in it:

```
$ cat sa.c
int func() {
int a = 0;
int b = 1/a;
return b;
}
```

2. Run the Clang Static Analyzer by passing the command-line options shown in the following command, and get the output on the screen:

```
$ clang -cc1 -analyze -analyzer-checker=core.DivideZero sa.c
sa.c:3:10: warning: Division by zero
int b = 1/a;
        ~^~
1 warning generated.
```

How it works...

The static analyzer core performs the symbolic execution of the program. The input values are represented by symbolic values. The values of the expressions are calculated by the analyzer using the input symbol and the path. The execution of the code is path-sensitive, and hence every possible path is analyzed.

While executing, the execution traces are represented by an exploded graph. Each node of this `ExplodedGraph` is called `ExplodedNode`. It consists of a `ProgramState` object, which represents the abstract state of the program; and a `ProgramPoint` object, which represents the corresponding location in the program.

For each type of bug, there is an associated checker. Each of these checkers is linked to the core in a way by which they contribute to the `ProgramState` construction. Each time the analyzer engine explores a new statement, it notifies each checker registered to listen for that statement, giving it an opportunity to either report a bug or modify the state.

Each checker registers for some events and callbacks such as `PreCall` (prior to the call of the function), `DeadSymbols` (when a symbol goes dead), and so on. They are notified in the case of the requested events, and they implement the action to be taken for such events.

In this recipe, we looked at a divide-by-zero checker, which reports when a divide-by-zero condition occurs. The checker, in this case, registers for the `PreStmt` callback, before a statement gets executed. It then checks the operator of the next statement to be executed, and if it finds a division operator, it looks for a zero value. If it finds such a possible value, it reports a bug.

See also

- For more detailed information about the static analyzer and checkers, visit `http://clang-analyzer.llvm.org/checker_dev_manual.html`

Using bugpoint

In this recipe, you will learn about a useful tool provided by LLVM infrastructure, known as bugpoint. Bugpoint allows us to narrow down the source of problems in the LLVM's tools and passes. It is helpful in debugging optimizer crashes, miscompilations by optimizers, or bad native code generation. Using this, we can get a small test case for our problem and work on that.

Getting ready

You need to build and install LLVM.

How to do it...

Perform the following steps:

1. Write the test cases using the bugpoint tool:

```
$ cat crash-narrowfunctiontest.ll
define i32 @foo() { ret i32 1 }

define i32 @test() {
  call i32 @test()
  ret i32 %1
}
define i32 @bar() { ret i32 2 }
```

2. Use bugpoint in this test case to view the results :

```
$ bugpoint -load  path-to-llvm/build/./lib/BugpointPasses.
so crash-narrowfunctiontest.ll -output-prefix crash-
narrowfunctiontest.ll.tmp -bugpoint-crashcalls -silence-passes
Read input file      : 'crash-narrowfunctiontest.ll'
*** All input ok
Running selected passes on program to test for crash: Crashed:
Aborted (core dumped)
Dumped core

*** Debugging optimizer crash!
Checking to see if these passes crash: -bugpoint-crashcalls:
Crashed: Aborted (core dumped)
```

```
Dumped core

*** Found crashing pass: -bugpoint-crashcalls
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-passes.bc'

*** You can reproduce the problem with: opt crash-
narrowfunctiontest.ll.tmp-passes.bc -load /home/mayur/LLVMSVN_REV/
llvm/llvm/rbuild/./lib/BugpointPasses.so -bugpoint-crashcalls

*** Attempting to reduce the number of functions in the testcase
Checking for crash with only these functions:  foo test bar:
Crashed: Aborted (core dumped)
Dumped core
Checking for crash with only these functions:  foo test: Crashed:
Aborted (core dumped)
Dumped core
Checking for crash with only these functions:  test: Crashed:
Aborted (core dumped)
Dumped core
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-reduced-
function.bc'

*** You can reproduce the problem with: opt crash-
narrowfunctiontest.ll.tmp-reduced-function.bc -load /home/mayur/
LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so -bugpoint-
crashcalls
Checking for crash with only these blocks: : Crashed: Aborted
(core dumped)
Dumped core
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-reduced-
blocks.bc'

*** You can reproduce the problem with: opt crash-
narrowfunctiontest.ll.tmp-reduced-blocks.bc -load /home/mayur/
LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so -bugpoint-
crashcalls
Checking for crash with only 1 instruction: Crashed: Aborted (core
dumped)
Dumped core
```

```
*** Attempting to reduce testcase by deleting instructions:
Simplification Level #1
Checking instruction:   %1 = call i32 @test()Success!


*** Attempting to reduce testcase by deleting instructions:
Simplification Level #0
Checking instruction:   %1 = call i32 @test()Success!


*** Attempting to perform final cleanups: Crashed: Aborted (core
dumped)
Dumped core
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-reduced-
simplified.bc'


*** You can reproduce the problem with: opt crash-
narrowfunctiontest.ll.tmp-reduced-simplified.bc -load
/home/mayur/LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so
-bugpoint-crashcalls
```

3. Now, to see the reduced test case, use the `llvm-dis` command to convert the `crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc` file to the `.ll` form. Then, view the reduced test case:

```
$ llvm-dis crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc
$ cat $ cat crash-narrowfunctiontest.ll.tmp-reduced-simplified.ll
define void @test() {
  call void @test()
  ret void
}
```

How it works...

The bugpoint tool runs all the passes specified in the command line on the test program. If any of these passes crash, bugpoint starts the crash debugger. The crash debugger tries to reduce the list of passes that cause this crash. Then it tries to removes unnecessary functions. Once able to reduce the test program to a single function, it tries to deletes the edges of the control flow graph to reduce the size of the function. After this, it proceeds to remove the individual LLVM instructions whose absence does not impact the failure. In the end, bugpoint gives the output showing which pass is causing the crash and a simplified reduced test case.

If the `-output` option wasn't specified, then bugpoint runs the program on a `"safe"` backend and generated reference output. It then compares the output generated by the selected code generator. If there is a crash, it runs the crash debugger as explained in the previous paragraph. Other than this, if the output generated by the code generator differs from the reference output, it starts the code generator debugger, which reduces the test case through techniques similar to those of the crash debugger.

Finally, if the output generated by the code generator and the reference output are the same, then bugpoint runs all the LLVM passes and checks the output against the reference output. If there is any mismatch, then it runs the miscompilation debugger. The miscompilation debugger works by splitting the test program into two pieces. It runs the optimizations as specified on one piece, then links the two pieces back together, and finally executes the result. It tries to narrow down to the pass that is causing miscompilation from the list of passes, and then pinpoints the portion of the test program that is being miscompiled. It outputs the reduced case that is causing the miscompilation.

In the preceding test case, bugpoint checks for the crash in all functions, and ends up knowing that the problem lies in the test function. It also tries to reduce the instructions within the function. The output for every stage is displayed on the terminal, which is self-explanatory. In the end, it produces a simplified reduced test case in the bitcode format, which we can convert to the LLVM IR and get the reduced test case.

See also

- To read more on bugpoint, go to `http://llvm.org/docs/Bugpoint.html`

Using LLDB

In this recipe, you will learn how to use the debugger known as `LLDB`, provided by LLVM. LLDB is a next-generation, high-performance debugger. It is essentially built as a set of reusable components that have advantages over the existing libraries in the larger LLVM project. You might find it quite similar to the `gdb` debugging tool.

Getting ready

We will need the following before working with LLDB:

1. To use LLDB, we need to check out the LLDB source code in the `llvm/tools` folder:
   `svn co http://llvm.org/svn/llvm-project/lldb/trunk lldb`
2. Build and install LLVM, which will also build LLDB simultaneously.

How to do it...

Perform the following steps:

1. Write a test case for a simple example using LLDB:

```
$ cat lldbexample.c
#include<stdio.h>
int globalvar = 0;

int func2(int a, int b) {
globalvar++;
return a*b;
}

int func1(int a, int b) {
globalvar++;
int d = a + b;
int e = a - b;
int f = func2(d, e);
return f;
}

int main() {
globalvar++;
int a = 5;
int b = 3;

int c = func1(a,b);
printf("%d", c);
return c;
}
```

2. Compile the code using Clang with the `-g` flag to generate the debug information:

```
$ clang -g lldbexample.c
```

3. Debug the output file generated in the previous file with LLDB. To load the output file, we need to pass its name to LLDB:

```
$ lldb a.out
(lldb) target create "a.out"
Current executable set to 'a.out' (x86_64).
```

4. Set a breakpoint in the main function:

```
(lldb) breakpoint set --name main
Breakpoint 1: where = a.out'main + 15 at lldbexample.c:20,
address = 0x00000000004005bf
```

5. To look at the list of breakpoints set, use the following command:

```
(lldb) breakpoint list
Current breakpoints:
1: name = 'main', locations = 1
  1.1: where = a.out'main + 15 at lldbexample.c:20, address =
a.out[0x00000000004005bf], unresolved, hit count = 0
```

6. Add a command to be executed when a breakpoint is hit. Here, let's add the back trace `bt` command when the breakpoint on the main function is hit:

```
(lldb) breakpoint command add 1.1
Enter your debugger command(s).  Type 'DONE' to end.
> bt
> DONE
```

7. Run the executable using the following command. This will hit the breakpoint on the `main` function and execute the back trace(`bt`) command, as set in the earlier step:

```
(lldb) process launch
Process 2999 launched: '/home/mayur/book/chap9/a.out' (x86_64)
Process 2999 stopped
* thread #1: tid = 2999, 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20, name = 'a.out', stop reason = breakpoint 1.1
    frame #0: 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20
   17
   18
   19    int main() {
-> 20    globalvar++;
   21    int a = 5;
   22    int b = 3;
```

```
   23
(lldb)  bt
* thread #1: tid = 2999, 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20, name = 'a.out', stop reason = breakpoint 1.1
  * frame #0: 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20
    frame #1: 0x00007ffff7a35ec5 libc.so.6'__libc_start_
main(main=0x00000000004005b0, argc=1,
argv=0x00007fffffffda18, init=<unavailable>, fini=<unavailable>,
rtld_fini=<unavailable>, stack_end=0x00007fffffffda08) + 245 at
libc-start.c:287
    frame #2: 0x0000000000400469 a.out
```

8. To set `watchpoint` on the global variable, use the following command:

```
(lldb) watch set var globalvar
Watchpoint created: Watchpoint 1: addr = 0x00601044 size = 4
state = enabled type = w
    declare @ '/home/mayur/book/chap9/lldbexample.c:2'
    watchpoint spec = 'globalvar'
    new value: 0
```

9. To stop the execution when the value of `globalvar` becomes `3`, use the `watch` command:

```
(lldb) watch modify -c '(globalvar==3)'
To view list of all watch points:
(lldb) watch list
Number of supported hardware watchpoints: 4
Current watchpoints:
Watchpoint 1: addr = 0x00601044 size = 4 state = enabled type
= w
    declare @ '/home/mayur/book/chap9/lldbexample.c:2'
    watchpoint spec = 'globalvar'
    new value: 0
    condition = '(globalvar==3)'
```

10. To continue execution after the main function, use the following command. The executable will stop when the value of `globalvar` becomes `3`, inside the `func2` function:

```
(lldb) thread step-over
(lldb) Process 2999 stopped
```

```
* thread #1: tid = 2999, 0x000000000040054b a.out'func2(a=8,
b=2) + 27 at lldbexample.c:6, name = 'a.out', stop reason =
watchpoint 1
    frame #0: 0x000000000040054b a.out'func2(a=8, b=2) + 27 at
lldbexample.c:6
   3
   4     int func2(int a, int b) {
   5     globalvar++;
-> 6     return a*b;
   7     }
   8
   9


Watchpoint 1 hit:
old value: 0
new value: 3
(lldb) bt
* thread #1: tid = 2999, 0x000000000040054b a.out'func2(a=8,
b=2) + 27 at lldbexample.c:6, name = 'a.out', stop reason =
watchpoint 1
  * frame #0: 0x000000000040054b a.out'func2(a=8, b=2) + 27 at
lldbexample.c:6
    frame #1: 0x000000000040059c a.out'func1(a=5, b=3) + 60 at
lldbexample.c:14
    frame #2: 0x00000000004005e9 a.out'main + 57 at
lldbexample.c:24
    frame #3: 0x00007ffff7a35ec5 libc.so.6'__libc_start_
main(main=0x00000000004005b0, argc=1,
argv=0x00007fffffffda18, init=<unavailable>,
fini=<unavailable>, rtld_fini=<unavailable>,
stack_end=0x00007fffffffda08) + 245 at libc-start.c:287
    frame #4: 0x0000000000400469 a.out
```

11. To continue the execution of the executable use the `thread continue` command, which will execute till the end as no other breakpoints are met:

```
(lldb) thread continue
Resuming thread 0x0bb7 in process 2999
Process 2999 resuming
Process 2999 exited with status = 16 (0x00000010)
```

12. To exit LLDB, use the following command:

```
(lldb)  exit
```

See also

- Check out `http://lldb.llvm.org/tutorial.html` for an exhaustive list of LLDB commands.

Using LLVM utility passes

In this recipe, you will learn about LLVM's utility passes. As the name signifies, they are of much utility to users who want to understand certain things about LLVM that are not easy to understand by going through code. We will look into two utility passes that represent the CFG of a program.

Getting ready

You need to build and install LLVM, and install the `graphviz` tool. You can download `graphviz` from `http://www.graphviz.org/Download.php`, or install it from your machine's package manager, if it is in the list of available packages.

How to do it...

Perform the following steps:

1. Write the test code required for running the utility passes. This test code consists of `if` blocks, it will create a new edge in the CFG:

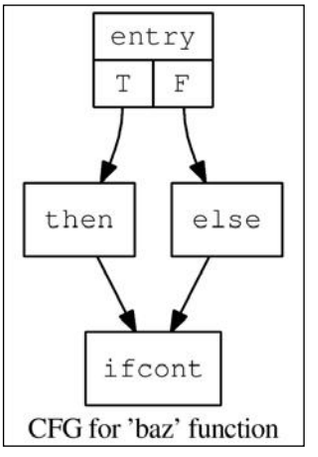
```
$ cat utility.ll
declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  %calltmp = call double @foo()
```

```
  br label %ifcont

else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else
]
  ret double %iftmp
}
```

2. Run the `view-cfg-only` pass to view the CFG of a function without the function body:

```
$ opt -view-cfg-only utility.ll
```

3. Now, view the `dot` file formed using the `graphviz` tool:

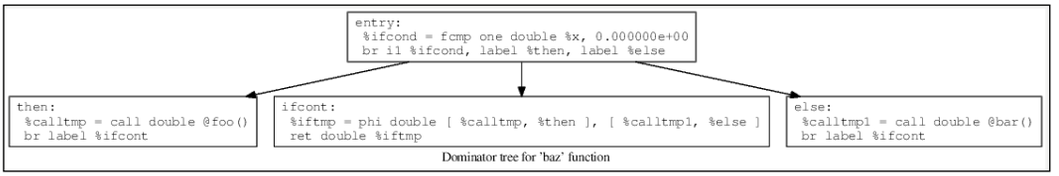


CFG for 'baz' function

4. Run the `view-dom` pass to view the **Dominator tree** of a function:

```
$ opt -view-dom utility.ll
```

5. View the `dot` file formed using the `graphviz` tool:



Dominator tree for 'baz' function

See also

- A list of the other utility passes is available at `http://llvm.org/docs/Passes.html#utility-passes`