LLVM

Chris Lattner

This chapter discusses some of the design decisions that shaped LLVM[1], an umbrella project that hosts and develops a set of close-knit low-level toolchain components (e.g., assemblers, compilers, debuggers, etc.), which are designed to be compatible with existing tools typically used on Unix systems. The name "LLVM" was once an acronym, but is now just a brand for the umbrella project. While LLVM provides some unique capabilities, and is known for some of its great tools (e.g., the Clang compiler[2], a C/C++/Objective-C compiler which provides a number of benefits over the GCC compiler), the main thing that sets LLVM apart from other compilers is its internal architecture.

From its beginning in December 2000, LLVM was designed as a set of reusable libraries with well-defined interfaces [LA04]. At the time, open source programming language implementations were designed as special-purpose tools which usually had monolithic executables. For example, it was very difficult to reuse the parser from a static compiler (e.g., GCC) for doing static analysis or refactoring. While scripting languages often provided a way to embed their runtime and interpreter into larger applications, this runtime was a single monolithic lump of code that was included or excluded. There was no way to reuse pieces, and very little sharing across language implementation projects.

Beyond the composition of the compiler itself, the communities surrounding popular language implementations were usually strongly polarized: an implementation usually provided *either* a traditional static compiler like GCC, Free Pascal, and FreeBASIC, *or* it provided a runtime compiler in the form of an interpreter or Just-In-Time (JIT) compiler. It was very uncommon to see language implementation that supported both, and if they did, there was usually very little sharing of code.

Over the last ten years, LLVM has substantially altered this landscape. LLVM is now used as a common infrastructure to implement a broad variety of statically and runtime compiled languages (e.g., the family of languages supported by GCC, Java, .NET, Python, Ruby, Scheme, Haskell, D, as well as countless lesser known languages). It has also replaced a broad variety of special purpose compilers, such as the runtime specialization engine in Apple's OpenGL stack and the image processing library in Adobe's After Effects product. Finally LLVM has also been used to create a broad variety of new products, perhaps the best known of which is the OpenCL GPU programming language and runtime.

11.1. A Quick Introduction to Classical Compiler Design

The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end (Figure 11.1). The front end parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST is optionally converted to a new representation for optimization, and the optimizer and back end are run on the code.

Figure 11.1: Three Major Components of a Three-Phase Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The back end (also known as the code generator) then maps the code onto the target instruction set. In addition to making *correct* code, it is responsible for generating *good* code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and JIT compilers. The Java Virtual Machine (JVM) is also an implementation of this model, which uses Java bytecode as the interface between the front end and optimizer.

11.1.1. Implications of this Design

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Figure 11.2.

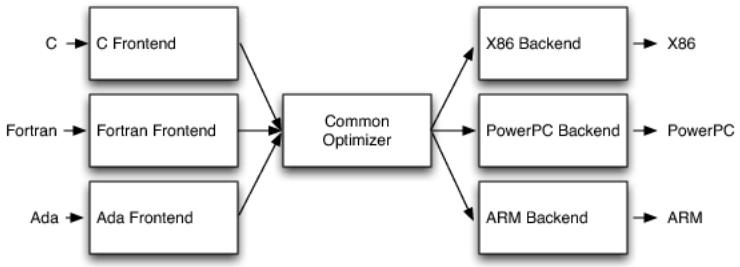

Figure 11.2: Retargetablity

With this design, porting the compiler to support a new source language (e.g., Algol or BASIC) requires implementing a new front end, but the existing optimizer and back end can be reused. If these parts weren't separated, implementing a new source language would require starting over from scratch, so supporting `N` targets and `M` source languages would need N*M compilers.

Another advantage of the three-phase design (which follows directly from retargetability) is that the compiler serves a broader set of programmers than it would if it only supported one source language and one target. For an open source project, this means that there is a larger community of potential contributors to draw from, which naturally leads to more enhancements and improvements to the compiler. This is the reason why open source compilers that serve many communities (like GCC) tend to generate better optimized machine code than narrower compilers like FreePASCAL. This isn't the case for proprietary compilers, whose quality is directly related to the project's budget. For example, the Intel ICC Compiler is widely known for the quality of code it generates, even though it serves a narrow audience.

A final major win of the three-phase design is that the skills required to implement a front end are different than those required for the optimizer and back end. Separating these makes it easier for a "front-end person" to enhance and maintain their part of the compiler. While this is a social issue, not a technical one, it matters a lot in practice, particularly for open source projects that want to reduce the barrier to contributing as much as possible.

11.2. Existing Language Implementations

While the benefits of a three-phase design are compelling and well-documented in compiler textbooks, in practice it is almost never fully realized. Looking across open source language implementations (back when LLVM was started), you'd find that the implementations of Perl, Python, Ruby and Java share no code. Further, projects like the Glasgow Haskell Compiler (GHC) and FreeBASIC are retargetable to multiple different CPUs, but their implementations are very specific to the one source language they support. There is

also a broad variety of special purpose compiler technology deployed to implement JIT compilers for image processing, regular expressions, graphics card drivers, and other subdomains that require CPU intensive work.

That said, there are three major success stories for this model, the first of which are the Java and .NET virtual machines. These systems provide a JIT compiler, runtime support, and a very well defined bytecode format. This means that any language that can compile to the bytecode format (and there are dozens of them[3]) can take advantage of the effort put into the optimizer and JIT as well as the runtime. The tradeoff is that these implementations provide little flexibility in the choice of runtime: they both effectively force JIT compilation, garbage collection, and the use of a very particular object model. This leads to suboptimal performance when compiling languages that don't match this model closely, such as C (e.g., with the LLJVM project).

A second success story is perhaps the most unfortunate, but also most popular way to reuse compiler technology: translate the input source to C code (or some other language) and send it through existing C compilers. This allows reuse of the optimizer and code generator, gives good flexibility, control over the runtime, and is really easy for front-end implementers to understand, implement, and maintain. Unfortunately, doing this prevents efficient implementation of exception handling, provides a poor debugging experience, slows down compilation, and can be problematic for languages that require guaranteed tail calls (or other features not supported by C).

A final successful implementation of this model is GCC[4]. GCC supports many front ends and back ends, and has an active and broad community of contributors. GCC has a long history of being a C compiler that supports multiple targets with hacky support for a few other languages bolted onto it. As the years go by, the GCC community is slowly evolving a cleaner design. As of GCC 4.4, it has a new representation for the optimizer (known as "GIMPLE Tuples") which is closer to being separate from the front-end representation than before. Also, its Fortran and Ada front ends use a clean AST.

While very successful, these three approaches have strong limitations to what they can be used for, because they are designed as monolithic applications. As one example, it is not realistically possible to embed GCC into other applications, to use GCC as a runtime/JIT compiler, or extract and reuse pieces of GCC without pulling in most of the compiler. People who have wanted to use GCC's C++ front end for documentation generation, code indexing, refactoring, and static analysis tools have had to use GCC as a monolithic application that emits interesting information as XML, or write plugins to inject foreign code into the GCC process.

There are multiple reasons why pieces of GCC cannot be reused as libraries, including rampant use of global variables, weakly enforced invariants, poorly-designed data structures, sprawling code base, and the use of macros that prevent the codebase from being compiled to support more than one front-end/target pair at a time. The hardest problems to fix, though, are the inherent architectural problems that stem from its early design and age. Specifically, GCC suffers from layering problems and leaky abstractions: the back end walks front-end ASTs to generate debug info, the front ends generate back-end data structures, and the entire compiler depends on global data structures set up by the command line interface.

11.3. LLVM's Code Representation: LLVM IR

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

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

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

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

done:
  ret i32 %b
}
```

This LLVM IR corresponds to this C code, which provides two different ways to add integers:

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

As you can see from this example, LLVM IR is a low-level RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions like add, subtract, compare, and branch. These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register.[5] LLVM IR supports labels and generally looks like a weird form of assembly language.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple type system (e.g., `i32` is a 32-bit integer, `i32**` is a pointer to pointer to 32-bit integer) and some details of the machine are abstracted away. For example, the calling convention is abstracted through `call` and `ret` instructions and explicit arguments. Another significant difference from machine code is that the LLVM IR doesn't use a fixed set of named registers, it uses an infinite set of temporaries named with a % character.

Beyond being implemented as a language, LLVM IR is actually defined in three isomorphic forms: the textual format above, an in-memory data structure inspected and modified by optimizations themselves, and an efficient and dense on-disk binary "bitcode" format. The LLVM Project also provides tools to convert the on-disk format from text to binary: `llvm-as` assembles the textual `.ll` file into a `.bc` file containing the bitcode goop and `llvm-dis` turns a `.bc` file into a `.ll` file.

The intermediate representation of a compiler is interesting because it can be a "perfect world" for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

11.3.1. Writing an LLVM IR Optimization

To give some intuition for how optimizations work, it is useful to walk through some examples. There are lots of different kinds of compiler optimizations, so it is hard to provide a recipe for how to solve an arbitrary problem. That said, most optimizations follow a simple three-part structure:

- Look for a pattern to be transformed.

- Verify that the transformation is safe/correct for the matched instance.
- Do the transformation, updating the code.

The most trivial optimization is pattern matching on arithmetic identities, such as: for any integer `X`, `X-X` is 0, `X-0` is `X`, `(X*2)-X` is `X`. The first question is what these look like in LLVM IR. Some examples are:

```
:   :   :
%example1 = sub i32 %a, %a
:   :   :
%example2 = sub i32 %b, 0
:   :   :
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
:   :   :
```

For these sorts of "peephole" transformations, LLVM provides an instruction simplification interface that is used as utilities by various other higher level transformations. These particular transformations are in the `SimplifySubInst` function and look like this:

```
// X - 0 -> X
if (match(Op1, m_Zero()))
  return Op0;

// X - X -> 0
if (Op0 == Op1)
  return Constant::getNullValue(Op0->getType());

// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
  return Op1;

…

return 0;  // Nothing matched, return null to indicate no transformation.
```

In this code, Op0 and Op1 are bound to the left and right operands of an integer subtract instruction (importantly, these identities don't necessarily hold for IEEE floating point!). LLVM is implemented in C++, which isn't well known for its pattern matching capabilities (compared to functional languages like Objective Caml), but it does offer a very general template system that allows us to implement something similar. The `match` function and the `m_` functions allow us to perform declarative pattern matching operations on LLVM IR code. For example, the `m_Specific` predicate only matches if the left hand side of the multiplication is the same as Op1.

Together, these three cases are all pattern matched and the function returns the replacement if it can, or a null pointer if no replacement is possible. The caller of this function (`SimplifyInstruction`) is a dispatcher that does a switch on the instruction opcode, dispatching to the per-opcode helper functions. It is called from various optimizations. A simple driver looks like this:

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
  if (Value *V = SimplifyInstruction(I))
    I->replaceAllUsesWith(V);
```

This code simply loops over each instruction in a block, checking to see if any of them simplify. If so (because `SimplifyInstruction` returns non-null), it uses the `replaceAllUsesWith` method to update anything in the code using the simplifiable operation with the simpler form.

11.4. LLVM's Implementation of Three-Phase Design

In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code, as shown in Figure 11.3. This is a very straightforward implementation of the three-phase design, but this simple description glosses over some of the power and flexibility that the LLVM architecture derives from LLVM IR.

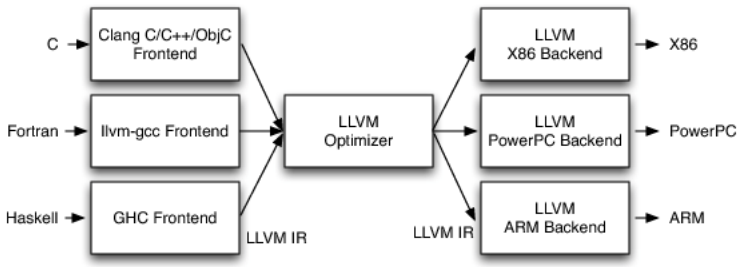

Figure 11.3: LLVM's Implementation of the Three-Phase Design

11.4.1. LLVM IR is a Complete Code Representation

In particular, LLVM IR is both well specified and the *only* interface to the optimizer. This property means that all you need to know to write a front end for LLVM is what LLVM IR is, how it works, and the invariants it expects. Since LLVM IR has a first-class textual form, it is both possible and reasonable to build a front end that outputs LLVM IR as text, then uses Unix pipes to send it through the optimizer sequence and code generator of your choice.

It might be surprising, but this is actually a pretty novel property to LLVM and one of the major reasons for its success in a broad range of different applications. Even the widely successful and relatively well-architected GCC compiler does not have this property: its GIMPLE mid-level representation is not a self-contained representation. As a simple example, when the GCC code generator goes to emit DWARF debug information, it reaches back and walks the source level "tree" form. GIMPLE itself uses a "tuple" representation for the operations in the code, but (at least as of GCC 4.5) still represents operands as references back to the source level tree form.

The implications of this are that front-end authors need to know and produce GCC's tree data structures as well as GIMPLE to write a GCC front end. The GCC back end has similar problems, so they also need to know bits and pieces of how the RTL back end works as well. Finally, GCC doesn't have a way to dump out "everything representing my code", or a way to read and write GIMPLE (and the related data structures that form the representation of the code) in text form. The result is that it is relatively hard to experiment with GCC, and therefore it has relatively few front ends.

11.4.2. LLVM is a Collection of Libraries

After the design of LLVM IR, the next most important aspect of LLVM is that it is designed as a set of libraries, rather than as a monolithic command line compiler like GCC or an opaque virtual machine like the JVM or .NET virtual machines. LLVM is an infrastructure, a collection of useful compiler technology that can be brought to bear on specific problems (like building a C compiler, or an optimizer in a special effects pipeline). While one of its most powerful features, it is also one of its least understood design points.

Let's look at the design of the optimizer as an example: it reads LLVM IR in, chews on it a bit, then emits LLVM IR which hopefully will execute faster. In LLVM (as in many other compilers) the optimizer is organized as a pipeline of distinct optimization passes each of which is run on the input and has a chance to do something. Common examples of passes are the inliner (which substitutes the body of a function into call sites), expression reassociation, loop invariant code motion, etc. Depending on the optimization level, different passes are run: for example at -O0 (no optimization) the Clang compiler runs no passes, at -O3 it runs a series of 67 passes in its optimizer (as of LLVM 2.8).

Each LLVM pass is written as a C++ class that derives (indirectly) from the `Pass` class. Most passes are written in a single `.cpp` file, and their subclass of the `Pass` class is defined in an anonymous namespace (which makes it completely private to the defining file). In order for the pass to be useful, code outside the file has to be able to get it, so a single function (to create the pass) is exported from the file. Here is a slightly simplified example of a pass to make things concrete.[6]

```
namespace {
  class Hello : public FunctionPass {
  public:
    // Print out the names of functions in the LLVM IR being optimized.
    virtual bool runOnFunction(Function &F) {
      cerr << "Hello: " << F.getName() << "\n";
      return false;
    }
  };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

As mentioned, the LLVM optimizer provides dozens of different passes, each of which are written in a similar style. These passes are compiled into one or more `.o` files, which are then built into a series of archive libraries (`.a` files on Unix systems). These libraries provide all sorts of analysis and transformation capabilities, and the passes are as loosely coupled as possible: they are expected to stand on their own, or explicitly declare their dependencies among other passes if they depend on some other analysis to do their job. When given a series of passes to run, the LLVM PassManager uses the explicit dependency information to satisfy these dependencies and optimize the execution of passes.

Libraries and abstract capabilities are great, but they don't actually solve problems. The interesting bit comes when someone wants to build a new tool that can benefit from compiler technology, perhaps a JIT compiler for an image processing language. The implementer of this JIT compiler has a set of constraints in mind: for example, perhaps the image processing language is highly sensitive to compile-time latency and has some idiomatic language properties that are important to optimize away for performance reasons.

The library-based design of the LLVM optimizer allows our implementer to pick and choose both the order in which passes execute, and which ones make sense for the image processing domain: if everything is defined as a single big function, it doesn't make sense to waste time on inlining. If there are few pointers, alias analysis and memory optimization aren't worth bothering about. However, despite our best efforts, LLVM doesn't magically solve all optimization problems! Since the pass subsystem is modularized and the PassManager itself doesn't know anything about the internals of the passes, the implementer is free to implement their own language-specific passes to cover for deficiencies in the LLVM optimizer or to explicit language-specific optimization opportunities. Figure 11.4 shows a simple example for our hypothetical XYZ image processing system:

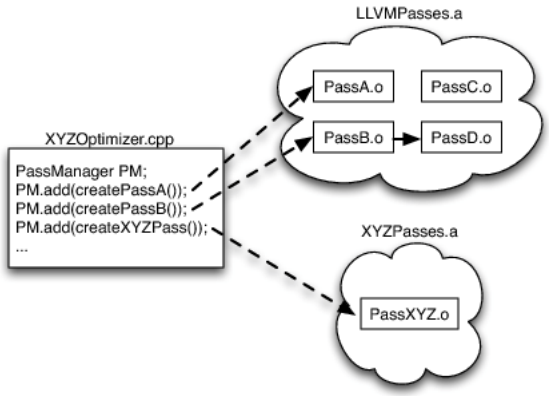

Figure 11.4: Hypothetical XYZ System using LLVM

Once the set of optimizations is chosen (and similar decisions are made for the code generator) the image processing compiler is built into an executable or dynamic library. Since the only reference to the LLVM optimization passes is the simple `create` function defined in each `.o` file, and since the optimizers live in `.a` archive libraries, only the optimization passes *that are actually used* are linked into the end application, not the entire LLVM optimizer. In our example above, since there is a reference to PassA and PassB, they will get linked in. Since PassB uses PassD to do some analysis, PassD gets linked in. However, since PassC (and dozens of other optimizations) aren't used, its code isn't linked into the image processing application.

This is where the power of the library-based design of LLVM comes into play. This straightforward design approach allows LLVM to provide a vast amount of capability, some of which may only be useful to specific audiences, without punishing clients of the libraries that just want to do simple things. In contrast, traditional compiler optimizers are built as a tightly interconnected mass of code, which is much more difficult to subset, reason about, and come up to speed on. With LLVM you can understand individual optimizers without knowing how the whole system fits together.

This library-based design is also the reason why so many people misunderstand what LLVM is all about: the LLVM libraries have many capabilities, but they don't actually *do* anything by themselves. It is up to the designer of the client of the libraries (e.g., the Clang C compiler) to decide how to put the pieces to best use. This careful layering, factoring, and focus on subset-ability is also why the LLVM optimizer can be used for such a broad range of different applications in different contexts. Also, just because LLVM provides JIT compilation capabilities, it doesn't mean that every client uses it.

11.5. Design of the Retargetable LLVM Code Generator

The LLVM code generator is responsible for transforming LLVM IR into target specific machine code. On the one hand, it is the code generator's job to produce the best possible machine code for any given target. Ideally, each code generator should be completely custom code for the target, but on the other hand, the code generators for each target need to solve very similar problems. For example, each target needs to assign values to registers, and though each target has different register files, the algorithms used should be shared wherever possible.

Similar to the approach in the optimizer, LLVM's code generator splits the code generation problem into individual passes—instruction selection, register allocation, scheduling, code layout optimization, and assembly emission—and provides many builtin passes that are run by default. The target author is then given the opportunity to choose among the default passes, override the defaults and implement completely custom target-specific passes as required. For example, the x86 back end uses a register-pressure-reducing scheduler since it has very few registers, but the PowerPC back end uses a latency optimizing scheduler since it has many of them. The x86 back end uses a custom pass to handle the x87 floating point stack, and the ARM back end uses a custom pass to place constant pool islands inside functions where needed. This flexibility allows target authors to produce great code without having to write an entire code generator from scratch for their target.

11.5.1. LLVM Target Description Files

The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets. This brings up another challenge: each shared component needs to be able to reason about target specific properties in a generic way. For example, a shared register allocator needs

to know the register file of each target and the constraints that exist between instructions and their register operands. LLVM's solution to this is for each target to provide a target description in a declarative domain-specific language (a set of `.td` files) processed by the tblgen tool. The (simplified) build process for the x86 target is shown in Figure 11.5.

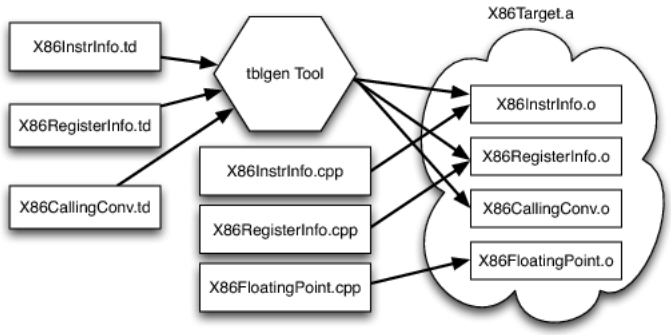

Figure 11.5: Simplified x86 Target Definition

The different subsystems supported by the `.td` files allow target authors to build up the different pieces of their target. For example, the x86 back end defines a register class that holds all of its 32-bit registers named "GR32" (in the `.td` files, target specific definitions are all caps) like this:

```
def GR32 : RegisterClass<[i32], 32,
  [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { … }
```

This definition says that registers in this class can hold 32-bit integer values ("i32"), prefer to be 32-bit aligned, have the specified 16 registers (which are defined elsewhere in the `.td` files) and have some more information to specify preferred allocation order and other things. Given this definition, specific instructions can refer to this, using it as an operand. For example, the "complement a 32-bit register" instruction is defined as:

```
let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
               (outs GR32:$dst), (ins GR32:$src),
               "not{l}\t$dst",
               [(set GR32:$dst, (not GR32:$src))]>;
```

This definition says that NOT32r is an instruction (it uses the `I` tblgen class), specifies encoding information (`0xF7, MRM2r`), specifies that it defines an "output" 32-bit register `$dst` and has a 32-bit register "input" named `$src` (the `GR32` register class defined above defines which registers are valid for the operand), specifies the assembly syntax for the instruction (using the `{}` syntax to handle both AT&T and Intel syntax), specifies the effect of the instruction and provides the pattern that it should match on the last line. The "let" constraint on the first line tells the register allocator that the input and output register must be allocated to the same physical register.

This definition is a very dense description of the instruction, and the common LLVM code can do a lot with information derived from it (by the `tblgen` tool). This one definition is enough for instruction selection to form this instruction by pattern matching on the input IR code for the compiler. It also tells the register allocator how to process it, is enough to encode and decode the instruction to machine code bytes, and is enough to parse and print the instruction in a textual form. These capabilities allow the x86 target to support generating a stand-alone x86 assembler (which is a drop-in replacement for the "gas" GNU assembler) and disassemblers from the target description as well as handle encoding the instruction for the JIT.

In addition to providing useful functionality, having multiple pieces of information generated from the same "truth" is good for other reasons. This approach makes it almost infeasible for the assembler and disassembler to disagree with each other in either assembly syntax or in the binary encoding. It also makes the target description easily testable: instruction encodings can be unit tested without having to involve the entire code generator.

While we aim to get as much target information as possible into the `.td` files in a nice declarative form, we still don't have everything. Instead, we require target authors to write some C++ code for various support routines and to implement any target specific passes they might need (like `X86FloatingPoint.cpp`, which handles the x87 floating point stack). As LLVM continues to grow new targets, it becomes more and more important to increase the amount of the target that can be expressed in the `.td` file, and we continue to increase the expressiveness of the `.td` files to handle this. A great benefit is that it gets easier and easier write targets in LLVM as time goes on.

11.6. Interesting Capabilities Provided by a Modular Design

Besides being a generally elegant design, modularity provides clients of the LLVM libraries with several interesting capabilities. These capabilities stem from the fact that LLVM provides functionality, but lets the client decide most of the *policies* on how to use it.

11.6.1. Choosing When and Where Each Phase Runs

As mentioned earlier, LLVM IR can be efficiently (de)serialized to/from a binary format known as LLVM bitcode. Since LLVM IR is self-contained, and serialization is a lossless process, we can do part of compilation, save our progress to disk, then continue work at some point in the future. This feature provides a number of interesting capabilities including support for link-time and install-time optimization, both of which delay code generation from "compile time".

Link-Time Optimization (LTO) addresses the problem where the compiler traditionally only sees one translation unit (e.g., a `.c` file with all its headers) at a time and therefore cannot do optimizations (like inlining) across file boundaries. LLVM compilers like Clang support this with the `-flto` or `-O4` command line option. This option instructs the compiler to emit LLVM bitcode to the `.o` file instead of writing out a native object file, and delays code generation to link time, shown in Figure 11.6.

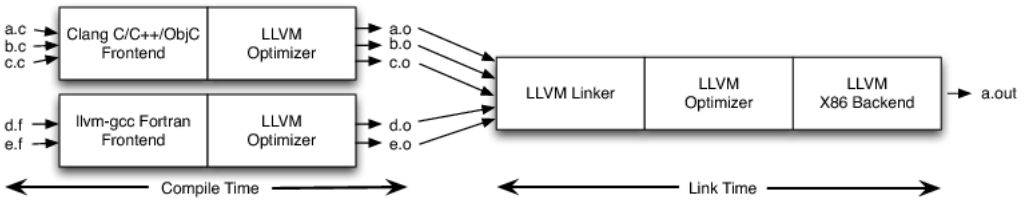

Figure 11.6: Link-Time Optimization

Details differ depending on which operating system you're on, but the important bit is that the linker detects that it has LLVM bitcode in the `.o` files instead of native object files. When it sees this, it reads all the bitcode files into memory, links them together, then runs the LLVM optimizer over the aggregate. Since the optimizer can now see across a much larger portion of the code, it can inline, propagate constants, do more aggressive dead code elimination, and more across file boundaries. While many modern compilers support LTO, most of them (e.g., GCC, Open64, the Intel compiler, etc.) do so by having an expensive and slow serialization process. In LLVM, LTO falls out naturally from the design of the system, and works across different source languages (unlike many other compilers) because the IR is truly source language neutral.

Install-time optimization is the idea of delaying code generation even later than link time, all the way to install time, as shown in Figure 11.7. Install time is a very interesting time (in cases when software is shipped in a box, downloaded, uploaded to a mobile device, etc.), because this is when you find out the specifics of the device you're targeting. In the x86 family for example, there are broad variety of chips and characteristics. By delaying instruction choice, scheduling, and other aspects of code generation, you can pick the best answers for the specific hardware an application ends up running on.

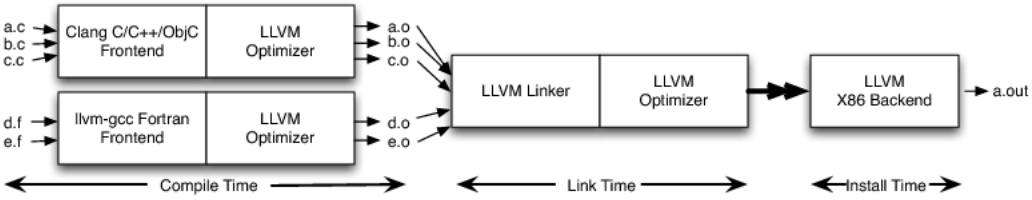

Figure 11.7: Install-Time Optimization

11.6.2. Unit Testing the Optimizer

Compilers are very complicated, and quality is important, therefore testing is critical. For example, after fixing a bug that caused a crash in an optimizer, a regression test should be added to make sure it doesn't happen again. The traditional approach to testing this is to write a `.c` file (for example) that is run through the compiler, and to have a test harness that verifies that the compiler doesn't crash. This is the approach used by the GCC test suite, for example.

The problem with this approach is that the compiler consists of many different subsystems and even many different passes in the optimizer, all of which have the opportunity to change what the input code looks like by the time it gets to the previously buggy code in question. If something changes in the front end or an earlier optimizer, a test case can easily fail to test what it is supposed to be testing.

By using the textual form of LLVM IR with the modular optimizer, the LLVM test suite has highly focused regression tests that can load LLVM IR from disk, run it through exactly one optimization pass, and verify the expected behavior. Beyond crashing, a more complicated behavioral test wants to verify that an optimization is actually performed. Here is a simple test case that checks to see that the constant propagation pass is working with add instructions:

```
; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
  %A = add i32 4, 5
  ret i32 %A
  ; CHECK: @test()
  ; CHECK: ret i32 9
}
```

The `RUN` line specifies the command to execute: in this case, the `opt` and `FileCheck` command line tools. The `opt` program is a simple wrapper around the LLVM pass manager, which links in all the standard passes (and can dynamically load plugins containing other passes) and exposes them through to the command line. The `FileCheck` tool verifies that its standard input matches a series of `CHECK` directives. In this case, this simple test is verifying that the `constprop` pass is folding the `add` of 4 and 5 into 9.

While this might seem like a really trivial example, this is very difficult to test by writing .c files: front ends often do constant folding as they parse, so it is very difficult and fragile to write code that makes its way downstream to a constant folding optimization pass. Because we can load LLVM IR as text and send it through the specific optimization pass we're interested in, then dump out the result as another text file, it is really straightforward to test exactly what we want, both for regression and feature tests.

11.6.3. Automatic Test Case Reduction with BugPoint

When a bug is found in a compiler or other client of the LLVM libraries, the first step to fixing it is to get a test case that reproduces the problem. Once you have a test case, it is best to minimize it to the smallest example that reproduces the problem, and also narrow it down to the part of LLVM where the problem happens, such as the optimization pass at fault. While you eventually learn how to do this, the process is tedious, manual, and particularly painful for cases where the compiler generates incorrect code but does not crash.

The LLVM BugPoint tool[7] uses the IR serialization and modular design of LLVM to automate this process. For example, given an input `.ll` or `.bc` file along with a list of optimization passes that causes an optimizer crash, BugPoint reduces the input to a small test case and determines which optimizer is at fault. It then outputs the reduced test case and the `opt` command used to reproduce the failure. It finds this by using techniques similar to "delta debugging" to reduce the input and the optimizer pass list. Because it knows the structure of LLVM IR, BugPoint does not waste time generating invalid IR to input to the optimizer, unlike the standard "delta" command line tool.

In the more complex case of a miscompilation, you can specify the input, code generator information, the command line to pass to the executable, and a reference output. BugPoint will first determine if the problem is due to an optimizer or a code generator, and will then repeatedly partition the test case into two pieces: one that is sent into the "known good" component and one that is sent into the "known buggy" component. By iteratively moving more and more code out of the partition that is sent into the known buggy code generator, it reduces the test case.

BugPoint is a very simple tool and has saved countless hours of test case reduction throughout the life of LLVM. No other open source compiler has a similarly powerful tool, because it relies on a well-defined intermediate representation. That said, BugPoint isn't perfect, and would benefit from a rewrite. It dates back to 2002, and is typically only improved when someone has a really tricky bug to track down that the existing tool doesn't handle well. It has grown over time, accreting new features (such as JIT debugging) without a consistent design or owner.

11.7. Retrospective and Future Directions

LLVM's modularity wasn't originally designed to directly achieve any of the goals described here. It was a self-defense mechanism: it was obvious that we wouldn't get everything right on the first try. The modular pass pipeline, for example, exists to make it easier to isolate passes so that they can be discarded after being replaced by better implementations[8].

Another major aspect of LLVM remaining nimble (and a controversial topic with clients of the libraries) is our willingness to reconsider previous decisions and make widespread changes to APIs without worrying about backwards compatibility. Invasive changes to LLVM IR itself, for example, require updating all of the optimization passes and cause substantial churn to the C++ APIs. We've done this on several occasions, and though it causes pain for clients, it is the right thing to do to maintain rapid forward progress. To make life easier for external clients (and to support bindings for other languages), we provide C wrappers for many popular APIs (which are intended to be extremely stable) and new versions of LLVM aim to continue reading old `.ll` and `.bc` files.

Looking forward, we would like to continue making LLVM more modular and easier to subset. For example, the code generator is still too monolithic: it isn't currently possible to subset LLVM based on features. For example, if you'd like to use the JIT, but have no need for inline assembly, exception handling, or debug information generation, it should be possible to build the code generator without linking in support for these features. We are also continuously improving the quality of code generated by the optimizer and code generator, adding IR features to better support new language and target constructs, and adding better support for performing high-level language-specific optimizations in LLVM.

The LLVM project continues to grow and improve in numerous ways. It is really exciting to see the number of different ways that LLVM is being used in other projects and how it keeps turning up in surprising new contexts that its designers never even thought about. The new LLDB debugger is a great example of this: it uses the C/C++/Objective-C parsers from Clang to parse expressions, uses the LLVM JIT to translate these into target code, uses the LLVM disassemblers, and uses LLVM targets to handle calling conventions among other things. Being able to reuse this existing code allows people developing debuggers to focus on writing the debugger logic, instead of reimplementing yet another (marginally correct) C++ parser.

Despite its success so far, there is still a lot left to be done, as well as the ever-present risk that LLVM will become less nimble and more calcified as it ages. While there is no magic answer to this problem, I hope that the continued exposure to new problem domains, a willingness to reevaluate previous decisions, and to redesign and throw away code will help. After all, the goal isn't to be perfect, it is to keep getting better over time.

Footnotes

1. `http://llvm.org`
2. `http://clang.llvm.org`
3. `http://en.wikipedia.org/wiki/List_of_JVM_languages`
4. A backronym that now stands for "GNU Compiler Collection".