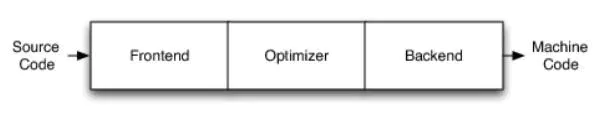
What is LLVM

1. LLVM instruction

传统的编译器通常分为三个部分，前端（frontEnd），优化器（Optimizer）和后端(backEnd)。



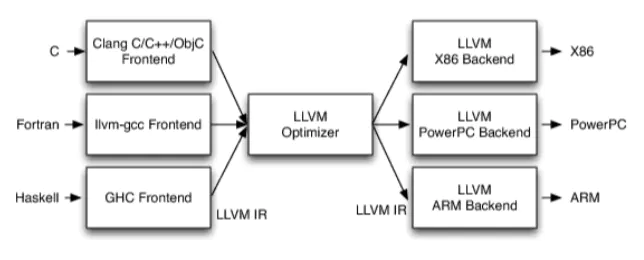
**Frontend**:**前端** 词法分析、语法分析、语义分析、生成中间代码

**Optimizer**:**优化器** 中间代码优化

**Backend**:**后端** 生成机器码

前端主要负责词法和语法分析，将源代码转化为抽象语法树；优化器则是在前端的基础上，对得到的中间代码进行优化，使代码更加高效；后端则是将已经优化的中间代码转化为针对各自平台的机器代码。(将代码转换成配硬件的机器码)。

GCC（GNU Compiler Collection，GNU 编译器套装），是一套由 GNU 开发的编程语言编译器。GCC原名为GNU C语言编译器，因为它原本只能处理 C语言。GCC 快速演进，变得可处理 C++、Fortran、Pascal、Objective-C、Java 以及 Ada 等他语言。



不同的语言采用不同的frontend，不同的硬件设备对应的机器码由对应的backend生成，LLVM IR统一frontend和backend两个过程，同时，LLVM Optimizer代码优化器针对LLVM IR进行优化。LLVM将compiler的frontend和backend进行了解耦。广义的LLVM包括前端，代码优化器，后端整个架构。狭义的LLVM表示LLVM的后端，代码优化和机器码生成。

相关文档链接如下：

LLVM: <https://llvm.org/>

LLVM 13 Documents：<https://releases.llvm.org/13.0.0/docs/index.html>

LVM Reference: <https://llvm.org/docs/Reference.html#xray>

LLVM Language Reference Manual: <https://llvm.org/docs/LangRef.html#llvm-objc-autoreleasepoolpop-intrinsic>

Getting Started with the LLVM System: <https://llvm.org/docs/GettingStarted.html>

后端LLVM详细介绍：<https://llvm.org/docs/WritingAnLLVMBackend.html>

The LLVM Linker：<https://releases.llvm.org/13.0.0/tools/lld/docs/index.html>

Building LLVM with CMake：<https://llvm.org/docs/CMake.html>

Machine IR (MIR) Format Reference Manual: <https://llvm.org/docs/MIRLangRef.html>

LLVM Bitcode File Format: <https://llvm.org/docs/BitCodeFormat.html>

https://llvm.org/docs/LangRef.html#llvm-objc-autoreleasepoolpop-intrinsic

User Guides: <https://llvm.org/docs/UserGuides.html>

[Tutorial: Creating an LLVM Backend for the Cpu0 Architecture](https://jonathan2251.github.io/lbd/): <https://jonathan2251.github.io/lbd/>

1. primary component of LLVM

2.1 The primary sub-projects of LLVM are:

1. The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.
2. Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer and clang-tidy are tools that automatically find bugs in your code, and are great examples of the sort of tools that can be built using the Clang frontend as a library to parse C/C++ code.LLVM的前端，由于语言解释。
3. The LLDB project builds on libraries provided by LLVM and Clang to provide a great native debugger. It uses the Clang ASTs and expression parser, LLVM JIT, LLVM disassembler, etc so that it provides an experience that "just works". It is also blazing fast and much more memory efficient than GDB at loading symbols.类似于GCC的GDB
4. The libc++ and libc++ ABI projects provide a standard conformant and high-performance implementation of the C++ Standard Library, including full support for C++11 and C++14.标准库的实现
5. The compiler-rt project provides highly tuned implementations of the low-level code generator support routines like "\_\_fixunsdfdi" and other calls generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It also provides implementations of run-time libraries for dynamic testing tools such as AddressSanitizer, ThreadSanitizer, MemorySanitizer, and DataFlowSanitizer.
6. The MLIR subproject is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.硬件兼容与软件优化相关
7. The OpenMP subproject provides an OpenMP runtime for use with the OpenMP implementation in Clang.跨平台的多线程实现
8. The polly project implements a suite of cache-locality optimizations as well as auto-parallelism and vectorization using a polyhedral model.缓存与并行化向量化
9. The libclc project aims to implement the OpenCL standard library.实现OpenCL standardlibrary
10. The klee project implements a "symbolic virtual machine" which uses a theorem prover to try to evaluate all dynamic paths through a program in an effort to find bugs and to prove properties of functions. A major feature of klee is that it can produce a testcase in the event that it detects a bug.错误与优先级检查
11. The LLD project is a new linker. That is a drop-in replacement for system linkers and runs much faster.关于GCC LD linker的drop-in replacement,LLVM的链接器

2.2 LLVM IR

IR(Intermediate representation)是LLVM的核心，IR是一种RISC的编程语言，对高级语言进行抽象，LLVM支持三种等价形式的IR(three equivalent forms):

1. 可读的汇编语言(a human-readable assembly format）
2. 内存列式的前端(an in-memory format suitable for frontends)
3. 序列化的凑密位代码(a dense bitcode format for serializing)

A simple ["Hello, world!" program](https://en.wikipedia.org/wiki/"Hello,_world!"_program" \o "\"Hello, world!\" program) in the IR format:

@.str = **internal** **constant** [14 **x** i8] **c**"hello, world\0A\00"

**declare** i32 @printf(i8\*, ...)

**define** i32 @main(i32 %argc, i8\*\* %argv) **nounwind** {entry:

%tmp1 = **getelementptr** [14 **x** i8], [14 **x** i8]\* @.str, i32 0, i32 0

%tmp2 = **call** i32 (i8\*, ...) @printf( i8\* %tmp1 ) **nounwind**

**ret** i32 0}

The many different conventions used and features provided by different targets mean that LLVM cannot truly produce a target-independent IR and retarget it without breaking some established rules. Examples of target dependence beyond what is explicitly mentioned in the documentation can be found in a 2011 proposal for "wordcode", a fully target-independent variant of LLVM IR intended for online distribution.[[33]](https://en.wikipedia.org/wiki/LLVM" \l "cite_note-33) A more practical example is [PNaCl](https://en.wikipedia.org/wiki/PNaCl" \o "PNaCl).[[34]](https://en.wikipedia.org/wiki/LLVM" \l "cite_note-34)

LLVMde IR模块旨在模块化解耦，做到与硬件解耦实现一套硬件无关的中间码，但是实际生成的IR还无法做到完全硬件无关，比如C/C++语言本身包含一些硬件依赖的属性。

The LLVM machine code (MC) subproject is LLVM's framework for translating machine instructions between textual forms and machine code. Formerly, LLVM relied on the system assembler, or one provided by a toolchain, to translate assembly into machine code. LLVM MC's integrated assembler supports most LLVM targets, including x86, x86-64, ARM, and ARM64. For some targets, including the various MIPS instruction sets, integrated assembly support is usable but still in the beta stage.

LLVM MC是代码汇编转换成机器码模块，实现IR到到平台特性的机器码的汇编工作。该部分是后端(backend)编译器开发的关键，也是GPU性能释放的关键。

LLVM Linker that lld subproject is an attempt to develop a built-in, platform-independent [linker](https://en.wikipedia.org/wiki/Linker_(computing)" \o "Linker (computing)) for LLVM.lld aims to remove dependence on a third-party linker. As of May 2017, lld supports [ELF](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format" \o "Executable and Linkable Format), [PE/COFF](https://en.wikipedia.org/wiki/PE/COFF" \o "PE/COFF), [Mach-O](https://en.wikipedia.org/wiki/Mach-O" \o "Mach-O), and [WebAssembly](https://en.wikipedia.org/wiki/WebAssembly" \o "WebAssembly) in descending order of completeness. lld is faster than both flavors of [GNU ld](https://en.wikipedia.org/wiki/GNU_ld" \o "GNU ld).

LLVM Linker是LLVM的链接器，实现与GNU ld 一样的功能。链接器用于实现变量函数的链接功能，实现函数/变量在作用域内的调用/使用，将不同的函数链接在一起，实现完整的功能。

Unlike the GNU linkers, lld has built-in support for [link-time optimization](https://en.wikipedia.org/wiki/Link-time_optimization" \o "Link-time optimization). This allows for faster code generation as it bypasses the use of a linker plugin, but on the other hand prohibits interoperability with other flavors of LTO.[[42]](https://en.wikipedia.org/wiki/LLVM" \l "cite_note-42) LLVM Linker相比较GNU GCC的ld更快一些，不支持GCC LTO文件。

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

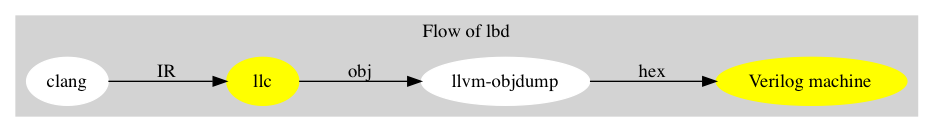
1. What is LLVM JIT

LLVM JIT represent LLVM just-in-time compilation component of dynamic translation or run-time complilation。

JIT也即动态编译/即时编译（dynamic translation/run-time compilations）在代码运行/执行时进行编译，而不是在执行前编译，在运行时完成字节码到机器码的转换。

1. Getting Started with the LLVM System

The LLVM Getting Started documentation may be out of date. The Clang Getting Started page might have more accurate information.



4.1 Toturials of LLVM Starting

This is an example workflow and configuration to get and build the LLVM source:

1. Checkout LLVM (including related subprojects like Clang):

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

Or, on windows, git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git

To save storage and speed-up the checkout time, you may want to do a shallow clone. For example, to get the latest revision of the LLVM project, use git clone --depth 1 https://github.com/llvm/llvm-project.git

1. Configure and build LLVM and Clang:

cd llvm-project

mkdir build

cd build

cmake -G <generator> [options] ../llvm

Some common build system generators are:

* Ninja — for generating Ninja build files. Most llvm developers use Ninja.
* Unix Makefiles — for generating make-compatible parallel makefiles.
* Visual Studio — for generating Visual Studio projects and solutions.
* Xcode — for generating Xcode projects.

Some Common options:

* -DLLVM\_ENABLE\_PROJECTS='...' — semicolon-separated list of the LLVM subprojects you’d like to additionally build. Can include any of: clang, clang-tools-extra, lldb, compiler-rt, lld, polly, or cross-project-tests.
* For example, to build LLVM, Clang, libcxx, and libcxxabi, use -DLLVM\_ENABLE\_PROJECTS="clang" -DLLVM\_ENABLE\_RUNTIMES="libcxx;libcxxabi".
* -DCMAKE\_INSTALL\_PREFIX=directory — Specify for directory the full pathname of where you want the LLVM tools and libraries to be installed (default /usr/local).
* -DCMAKE\_BUILD\_TYPE=type — Valid options for type are Debug, Release, RelWithDebInfo, and MinSizeRel. Default is Debug.
* -DLLVM\_ENABLE\_ASSERTIONS=On — Compile with assertion checks enabled (default is Yes for Debug builds, No for all other build types).

cmake --build . [--target <target>] or the build system specified above directly.

* The default target (i.e. cmake --build . or make) will build all of LLVM.
* The check-all target (i.e. ninja check-all) will run the regression tests to ensure everything is in working order.
* CMake will generate build targets for each tool and library, and most LLVM sub-projects generate their own check-<project> target.
* Running a serial build will be slow. To improve speed, try running a parallel build. That’s done by default in Ninja; for make, use the option -j NN, where NN is the number of parallel jobs, e.g. the number of available CPUs.

For more information see CMake

If you get an “internal compiler error (ICE)” or test failures, see below.Consult the Getting Started with LLVM section for detailed information on configuring and compiling LLVM. Go to Directory Layout to learn about the layout of the source code tree.

4.2 Host C++ Toolchain, both Compiler and Standard Library

LLVM对C++编译器有较高的要求，因为LLVM也是C++程序，在编译LLVM工具时名需要本地的C++编译器和标准库，下面是对本地库及版本的要求：

* Clang 3.5
* Apple Clang 6.0
* GCC 5.1
* Visual Studio 2017

GNU ld 2.16.X: Some 2.16.X versions of the ld linker will produce very long warning messages complaining that some “.gnu.linkonce.t.\*” symbol was defined in a discarded section. You can safely ignore these messages as they are erroneous and the linkage is correct. These messages disappear using ld 2.17.

GNU binutils 2.17: Binutils 2.17 contains [a bug](http://sourceware.org/bugzilla/show_bug.cgi?id=3111) which causes huge link times (minutes instead of seconds) when building LLVM. We recommend upgrading to a newer version (2.17.50.0.4 or later).

GNU Binutils 2.19.1 Gold: This version of Gold contained [a bug](http://sourceware.org/bugzilla/show_bug.cgi?id=9836) which causes intermittent failures when building LLVM with position independent code. The symptom is an error about cyclic dependencies. We recommend upgrading to a newer version of Gold.

4.3 [Local LLVM Configuration](https://llvm.org/docs/GettingStarted.html" \l "id20)

具体详情见 <https://llvm.org/docs/GettingStarted.html>

|  |  |
| --- | --- |
| **Variable** | **Purpose** |
| CMAKE\_C\_COMPILER | Tells cmake which C compiler to use. By default, this will be /usr/bin/cc. |
| CMAKE\_CXX\_COMPILER | Tells cmake which C++ compiler to use. By default, this will be /usr/bin/c++. |
| CMAKE\_BUILD\_TYPE | Tells cmake what type of build you are trying to generate files for. Valid options are Debug, Release, RelWithDebInfo, and MinSizeRel. Default is Debug. |
| CMAKE\_INSTALL\_PREFIX | Specifies the install directory to target when running the install action of the build files. |
| PYTHON\_EXECUTABLE | Forces CMake to use a specific Python version by passing a path to a Python interpreter. By default the Python version of the interpreter in your PATH is used. |
| LLVM\_TARGETS\_TO\_BUILD | A semicolon delimited list controlling which targets will be built and linked into llvm. The default list is defined as LLVM\_ALL\_TARGETS, and can be set to include out-of-tree targets. The default value includes: AArch64, AMDGPU, ARM, AVR, BPF, Hexagon, Lanai, Mips, MSP430, NVPTX, PowerPC, RISCV, Sparc, SystemZ, WebAssembly, X86, XCore. |
| LLVM\_ENABLE\_DOXYGEN | Build doxygen-based documentation from the source code This is disabled by default because it is slow and generates a lot of output. |
| LLVM\_ENABLE\_PROJECTS | A semicolon-delimited list selecting which of the other LLVM subprojects to additionally build. (Only effective when using a side-by-side project layout e.g. via git). The default list is empty. Can include: clang, clang-tools-extra, compiler-rt, cross-project-tests, flang, libc, libclc, libcxx, libcxxabi, libunwind, lld, lldb, mlir, openmp, polly, or pstl. |
| LLVM\_ENABLE\_SPHINX Build | sphinx-based documentation from the source code. This is disabled by default because it is slow and generates a lot of output. Sphinx version 1.5 or later recommended. |
| LLVM\_BUILD\_LLVM\_DYLIB | Generate libLLVM.so. This library contains a default set of LLVM components that can be overridden with LLVM\_DYLIB\_COMPONENTS. The default contains most of LLVM and is defined in tools/llvm-shlib/CMakelists.txt. This option is not available on Windows. |
| LLVM\_OPTIMIZED\_TABLEGEN | Builds a release tablegen that gets used during the LLVM build. This can dramatically speed up debug builds. |

Variables are passed to cmake on the command line using the format -D<variable name>=<value>. The following variables are some common options used by people developing LLVM.

4.4 [Directory Layout](https://llvm.org/docs/GettingStarted.html" \l "id25) and source code(LLVMk开源代码不同模块及相应的路径)

4.4.1 [llvm/cmake](https://llvm.org/docs/GettingStarted.html" \l "id26)

Generates system build files.

* llvm/cmake/modules:

Build configuration for llvm user defined options. Checks compiler version and linker flags.

* llvm/cmake/platforms:

Toolchain configuration for Android NDK, iOS systems and non-Windows hosts to target MSVC.

4.4.2 [llvm/include](https://llvm.org/docs/GettingStarted.html" \l "id28)

Public header files exported from the LLVM library. The three main subdirectories:

* llvm/include/llvm

All LLVM-specific header files, and subdirectories for different portions of LLVM: Analysis, CodeGen, Target, Transforms, etc…

* llvm/include/llvm/Support

Generic support libraries provided with LLVM but not necessarily specific to LLVM. For example, some C++ STL utilities and a Command Line option processing library store header files here.

* llvm/include/llvm/Config

Header files configured by cmake. They wrap “standard” UNIX and C header files. Source code can include these header files which automatically take care of the conditional #includes that cmake generates.

4.4.3 [llvm/lib](https://llvm.org/docs/GettingStarted.html" \l "id29)

Most source files are here. By putting code in libraries, LLVM makes it easy to share code among the [tools](https://llvm.org/docs/GettingStarted.html" \l "tools).

* llvm/lib/IR/

Core LLVM source files that implement core classes like Instruction and BasicBlock.

* llvm/lib/AsmParser/

Source code for the LLVM assembly language parser library.

* llvm/lib/Bitcode/

Code for reading and writing bitcode.

* llvm/lib/Analysis/

A variety of program analyses, such as Call Graphs, Induction Variables, Natural Loop Identification, etc.

* llvm/lib/Transforms/

IR-to-IR program transformations, such as Aggressive Dead Code Elimination, Sparse Conditional Constant Propagation, Inlining, Loop Invariant Code Motion, Dead Global Elimination, and many others.

* llvm/lib/Target/

Files describing target architectures for code generation. For example, llvm/lib/Target/X86 holds the X86 machine description.

* llvm/lib/CodeGen/

The major parts of the code generator: Instruction Selector, Instruction Scheduling, and Register Allocation.

* llvm/lib/MC/

The libraries represent and process code at machine code level. Handles assembly and object-file emission.

* llvm/lib/ExecutionEngine/

Libraries for directly executing bitcode at runtime in interpreted and JIT-compiled scenarios.

* llvm/lib/Support/

Source code that corresponding to the header files in llvm/include/ADT/ and llvm/include/Support/.

4.4.4 [llvm/bindings](https://llvm.org/docs/GettingStarted.html" \l "id30)

Contains bindings for the LLVM compiler infrastructure to allow programs written in languages other than C or C++ to take advantage of the LLVM infrastructure. LLVM project provides language bindings for Go, OCaml and Python.

4.4.5 [llvm/tools](https://llvm.org/docs/GettingStarted.html" \l "id34)

Executables built out of the libraries above, which form the main part of the user interface. You can always get help for a tool by typing tool\_name -help. The following is a brief introduction to the most important tools. More detailed information is in the [Command Guide](https://llvm.org/docs/CommandGuide/index.html).

* bugpoint

bugpoint is used to debug optimization passes or code generation backends by narrowing down the given test case to the minimum number of passes and/or instructions that still cause a problem, whether it is a crash or miscompilation. See [HowToSubmitABug.html](https://llvm.org/docs/HowToSubmitABug.html) for more information on using bugpoint.

* llvm-ar

The archiver produces an archive containing the given LLVM bitcode files, optionally with an index for faster lookup.

* llvm-as

The assembler transforms the human readable LLVM assembly to LLVM bitcode.

* llvm-dis

The disassembler transforms the LLVM bitcode to human readable LLVM assembly.

* llvm-link

llvm-link, not surprisingly, links multiple LLVM modules into a single program.

* lli

lli is the LLVM interpreter, which can directly execute LLVM bitcode (although very slowly…). For architectures that support it (currently x86, Sparc, and PowerPC), by default, lli will function as a Just-In-Time compiler (if the functionality was compiled in), and will execute the code much faster than the interpreter.

* llc

llc is the LLVM backend compiler, which translates LLVM bitcode to a native code assembly file.

* opt

opt reads LLVM bitcode, applies a series of LLVM to LLVM transformations (which are specified on the command line), and outputs the resultant bitcode. ‘opt -help’ is a good way to get a list of the program transformations available in LLVM.

opt can also run a specific analysis on an input LLVM bitcode file and print the results. Primarily useful for debugging analyses, or familiarizing yourself with what an analysis does.

* [llvm/utils](https://llvm.org/docs/GettingStarted.html" \l "id35)

Utilities for working with LLVM source code; some are part of the build process because they are code generators for parts of the infrastructure.

* codegen-diff

codegen-diff finds differences between code that LLC generates and code that LLI generates. This is useful if you are debugging one of them, assuming that the other generates correct output. For the full user manual, run `perldoc codegen-diff'.

* getsrcs.sh

Finds and outputs all non-generated source files, useful if one wishes to do a lot of development across directories and does not want to find each file. One way to use it is to run, for example: xemacs `utils/getsources.sh` from the top of the LLVM source tree.

* llvmgrep

Performs an egrep -H -n on each source file in LLVM and passes to it a regular expression provided on llvmgrep’s command line. This is an efficient way of searching the source base for a particular regular expression.

* TableGen/ (指令集和寄存器相关)

Contains the tool used to generate register descriptions, instruction set descriptions, and even assemblers from common TableGen description files.

4.5 Building LLVM with CMake

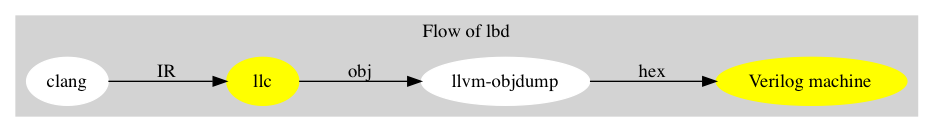
1. LLVM的API详情：

LLVM 相关的详细API文档，包括各种函数，类，结构体等。

<https://llvm.org/doxygen/index.html>

<https://llvm.org/doxygen/>

1. Instruction of LLVM compiler process



LLVM 是一种应用广泛的开源编译器架构，该架构采用模块化设计，能够轻松地新增对编程语言和处理器架构的支持。 CUDA 编译器可支持 C、C++ 以及 Fortran 语言，能够为运用大规模并行 NVIDIA GPU 的应用程序加速。 NVIDIA 携手 LLVM 开发者，共同提供针对 LLVM 内核的 CUDA 编译器源代码变化以及并行线程执行后端。如此一来，程序员便能够利用更广泛的编程语言来针对 GPU 加速器开发应用程序，从而令 GPU 计算比以往任何时候都更加唾手可得、更加普遍。

LLVM 支持各种各样的编程语言和前端，其中包括 C/C++、Objective-C、Fortran、Ada、Haskell、Java bytecode、Python、Ruby、ActionScript、GLSL 以及 Rust。它也是 NVIDIA 在其 CUDA C/C++ 架构中所采用的编译器架构，长期以来一直被苹果、AMD 以及 Adobe 等领先企业广泛采用。

Double Negative 公司研究员兼 LLVM 项目贡献者 Dan Bailey 表示：“Double Negative 已经将其流体动力学解算软件移植到 Jet 这一特定领域的语言上来，该语言就是基于 LLVM 的。 除了支持的现有架构以外，NVIDIA (英伟达™) 的全新开源 LLVM 编译器让程序员能够毫不费力地编译那些针对 NVIDIA GPU 架构而大力优化的代码，从而可大幅加速电影视觉特效中用到的模拟计算。”

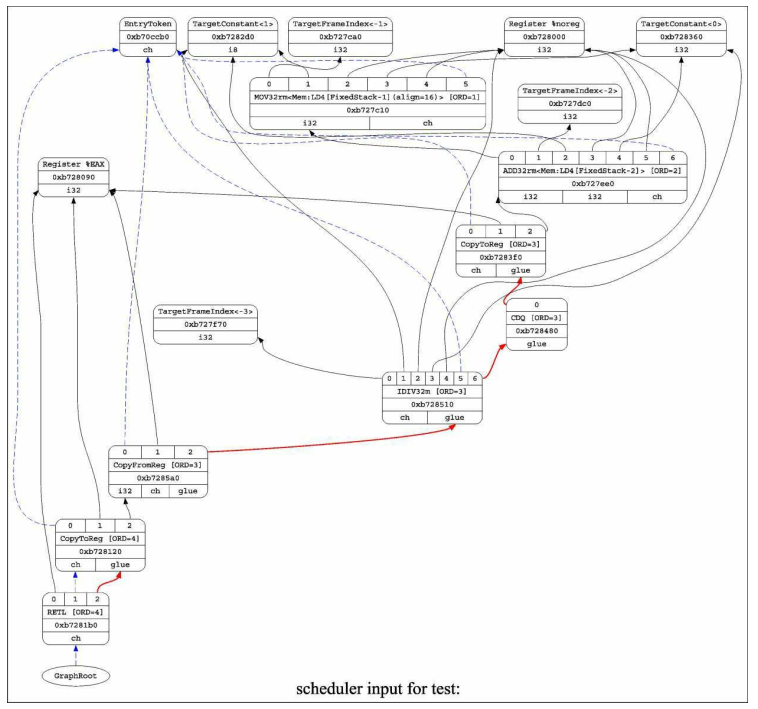
LLVM采用模块化设计，不同的模块之间进行解耦，它的设计就是基于库的。它允许你选择各个Pass(趟)的执行顺序,也能够选择执行哪些优化Pass。LLVM的代码有3种表示形式：内存编译器中的IR、存于磁盘的bitcode，以及用户可读的汇编码。LLVM IR是基于静态单赋值(Static Single Assignment——SSA)的，并且提供了类型安全性、底层操作性、灵活性，因此能够清楚表达绝大多数高级语言。这种表示形式贯穿LLVM编译的各个阶段。事实上，LLVM IR致力于成为一种足够底层的通用IR，只有这样，高级语言的诸多特性才能够得以实现。

6.1 源码编译成LLVM IR

LLVM frontend模块用作编译源码Text Code为LLVM IR，如C/C++语言的前端Clang/Clang++，指令如下：

clang -emit-llvm -S test.c -o test.ll

LLVM开源了多种前端语言支持，包括C,C++,Objective-C,FORTRAN。



6.2 LLVM IR编译成bitcode

LLVM bitcode(也称为字节码——bytecode)由两部分组成:位流（bitstream,可类比字节流),以及将LLVM IR编码成位流的编码格式，LLVM详细文档https://llvm.org/docs/BitCodeFormat.html，需要安装llvm-as工具。指令如下：

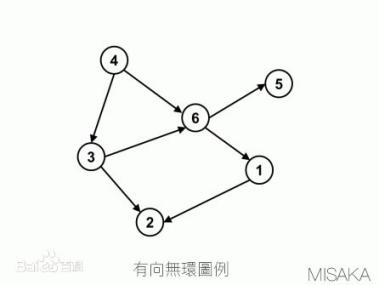
llvm-as test.ll –o test.bc

在得到优化过的LLVM IR之后，下一个阶段就是把它转为目标平台的指令了。LLVM通过SelectionDAG来将IR转为机器指令。在此过程中，指令通过DAG的节点来表示，最后线性的IR便被转为了SelectionDAG。在此之后，SelectionDAG还要经历以下几个阶段。

* 由LLVM IR创建SelectionDAG。
* SelectionDAG节点合法化。
* DAG合并优化。
* 针对目标指令的指令选择。
* 调度并发射机器指令。
* 寄存器分配——SSA解构、 寄存器赋值、 寄存器溢出。
* 发射机器码。

**6.2.1 LLVM IR转为SelectionDAG(**directed acyclic graph)

在IR转换和优化之后，LLVM IR指令会转换为SelectionDAG节点。Selection DAG节点由SelectionDAGBuilder类创建，SelectionDAGISel类调用SelectionDAGBuilder::visit()函数，遍历每个IR指令来创建SDAGNode节点。SelectionDAGBuilder::visitSDiv方法用于处理SDiv指令，它会依据ISD::SDIV操作码来向DAG请求一个新的SDNode节点，再创建DAG中的节点。



**6.2.2 合法化SelectionDAG（平台支持适配）**

创建的SelectionDAG节点未必会被目标架构全部支持，因此还需要对DAG节点做出一点修改以适应目标平台，这一过程叫作合法化（legalization）。在Selection DAG的初始阶段， 这些不被支持的节点被认为是不合法的。 在SelectionDAG机制真正为DAG节点进行机器指令发射之前， 这些不合法的节点都会做一些转换以支持目标平台。

SDNode合法化包括数据类型和操作两个方面。目标平台的相关信息通过一个叫作TargetLowering的接口传递给平台无关的算法，这个接口由目标平台实现，描述了LLVM IR如何lowering到合法的SelectionDAG操作。例如，x86平台的lowering由X86TargetLowering接口实现。在lowering的过程中，由setOperationAction()函数来指定ISD节点是否需要被操作合法化扩展或改变。在此之后，SelectionDAGLegalize::LegalizeOp如果发现扩展标记，就在setOperationAction()调用中用特定参数替换SDNode节点。

**6.2.3 从目标平台无关DAG转换为机器DAG**

在完成指令合法化之后， SDNode需要被转为MachineSDNode， 也就是转为目标平台的机器指令。 机器指令由一个通用的基于表的.td文件描述， 之后这些文件通过tablegen工具转为.inc文件， 在.inc文件中用枚举类型描述了目标平台的寄存器、 指令集等信息， 并且可以直接被C++代码调用。 指令选择的过程可以由SelectCode自动选择器完成， 或者通过编写自定义的SelectionDAGISel::Select函数自己定制。 在这一步中创建的DAG节点是MachineSDNode节点， 它是SDNode的子类， 持有用来构建真实机器指令的必要信息， 但仍是DAG节点形式。

**6.2.4 指令调度(DAG形式转换为机器三地址)**

机器执行线性指令集， 而现在我们得到的机器指令仍是DAG形式的， 所以还需要把DAG转为线性指令集， 这个过程可以通过对DAG进行一次拓扑排序完成。 尽管很轻松地就能得到线性指令集， 但它可能不是最优化的结果， 例如由于指令依赖、 寄存器压力、 流水线阻塞等问题， 都会造成执行延迟。 因此还需要对线性指令集进行顺序上的优化，这个过程叫作指令调度。 由于目标平台有自己的寄存器集和自定义的指令流水线， 所以它们也提供了自己的调度接口和一些启发式算法来优化、 加速代码。 在计算了代码的最佳执行顺序之后， 调度器就会发射机器基本块中的机器指令， 最终解构DAG。

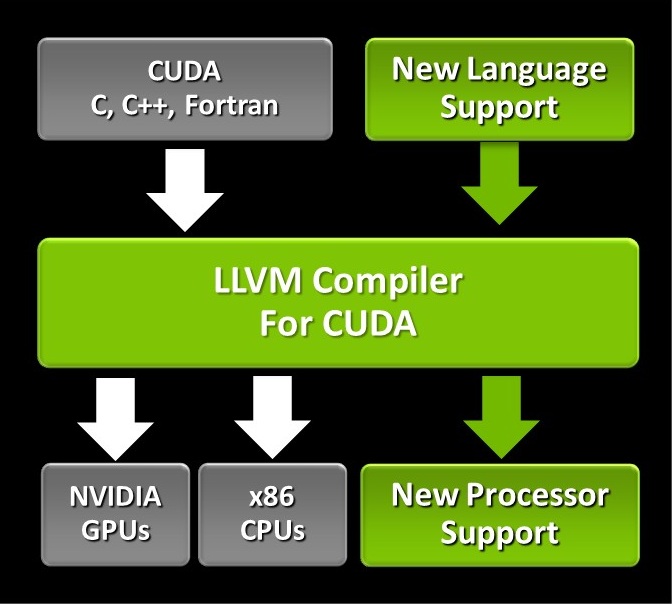
**6.2.5 寄存器分配**

**6.2.6 代码发射**

1. CUDA LLVM Compiler

NVIDIA's CUDA Compiler(NVCC) is based on the widely used [LLVM](http://llvm.org/) open source compiler infrastructure. Developers can create or extend programming languages with support for GPU acceleration using the [NVIDIA Compiler SDK](https://developer.nvidia.com/cuda-llvm-compiler" \l "cudacompilersdk).You can add support for GPU acceleration to a new or existing language by creating a language-specific frontend that compiles your language to the internal representation(IR) used by LLVM. （编写新的前端支持新语言）

The IR generated by your front end is then optimized for execution on your target device. You can add support for a new device by developing a processor-specific backend which will perform the final compilation on the optimized LLVM IR.(对于新平台需要自己写后端来解释编译LLVM IR，生成平台相关的可执行机器码)



The following components of the NVIDIA Compiler SDK are shipped as part of the latest [CUDA Toolkit Installer](https://developer.nvidia.com/cuda-downloads), github: https://github.com/nvidia-compiler-sdk:

* An optimizing compiler library (libnvvm.so, nvvm.dll/nvvm.lib, libnvvm.dylib) and its header file nvvm.h are provided for compiler developers who want to generate PTX from a program written in NVVM IR, which is a compiler internal representation based on LLVM.(PTX, Parallel Thread execution,并行库,目的就是在编译代码时可以将一些并行计算的代码通过调用CUDA代码调用GPU进行计算。SDK里实现了并行计算库)
* A set of libraries, libdevice.\*.bc, that implement the common math functions for devices in the LLVM bitcode format.(在源码中实现函数库,编译器将CUDA的函数库编译成兼容GPU的LLVM IR bitcode代码,然后转换成机器码。)
* A set of samples that illustrate the use of the compiler SDK.
* Documents for the Compiler SDK (including the specification for LLVM IR, an API document for libnvvm, and an API document for libdevice), can be found under the doc sub-directory, or [online](http://docs.nvidia.com/cuda/index.html" \l "compiler-sdk).
* The optimizing compiler libraries, the lidevice libraries and samples can be found under the nvvm sub-directory, seen after the CUDA Toolkit Install.

CUDA compilation works as follows: the input program is preprocessed for device compilation and is compiled to CUDA binary (cubin) and/or PTX intermediate code, which are placed in a fatbinary. The input program is preprocessed once again for host compilation and is synthesized to embed the fatbinary and transform CUDA specific C++ extensions into standard C++ constructs. Then the C++ host compiler compiles the synthesized host code with the embedded fatbinary into a host object.

NVCC是NVIDIA基于LLVM开发的一套编译工具，NVCC即可以对.cu的CUDA源码文件进行编译，将.cu源码编译成cu.cpp.ii/cudafe1.cpp的中间代码，也可以直接对.cpp源码进行编译，另外也可以对.cu和.c,.cpp源码文件一起编译，目前我司采用Clang替代NVIDIA的NVCC编译器，同时在编译源码时需要我们先将.cu的源码进行编译，然后链接.cu文件的中间产物或者库，再编译.cpp的源码。