

# LLVM代码生成器进一步深入，第一部分

翻译

2014年11月12日 15:59:56

标签：compiler / llvm

2681



1

原作：<http://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1/>

作者：Eli Berman

在前一篇文章中，我追踪了当从源语言编译到机器代码时，在LLVM中一条指令的各种化身。那篇文章简明地提到了LLVM中的许多层面，它们中的每一个都有趣且不简单。



这里我希望关注其中一个最重要及复杂的层面——代码生成器，特别地指令选择机制。一个简短的提醒：代码生成器的任务是高级的、几乎机器无关的LLVM IR转换为低级的、机器相关的机器语言。指令选择是这样的过程，其中IR抽象操作被映射到目标机器架构的具体指令。



本文将追寻一个简单的例子来展示起作用的指令选择机制（LLVM的说法，*ISel*）。



## 开始：简单乘法的DAG

下面是一些简单的IR：

```
define i64 @imul(i64 %a, i64 %b) nounwind readnone {
```

```
entry:
```

```
    %mul = mul nsw i64 %b, %a
```

```
    ret i64 %mul
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！



wuhui\_gdnt

原创

592

粉丝

168

喜欢

15

评论

51



等级：博客 7

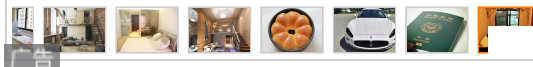
访问量：59万+

积分：1万+

排名：1061



一室一厅出租



他的最新文章

更多

登录

注册



它是在x64机器上从下面的C代码，用Clang（选项-emit-llvm）编译得到的：

```
long imul(long a, long b) {  
  
    return a * b;  
  
}
```



代码生成器完成的第一件事是把IR转换为一个selection DAG表示。这是刚开始的DAG，就在构建出来之后：



LLVM学习笔记（30）

64-ia-32架构优化手册（14）

Intel, AMD及VIA CPU的微架构（14）

LLVM学习笔记（29）

文章分类

ELF对线程局部储存的处理	7篇
GCC-3.4.6源代码学习笔记	209篇
GCC后端及汇编发布	50篇
Linux内核Modutils工具系列	16篇
Studying note of GCC-3.4.6 s...	202篇
GCC's back-end & assemble ...	17篇
展开	

文章存档

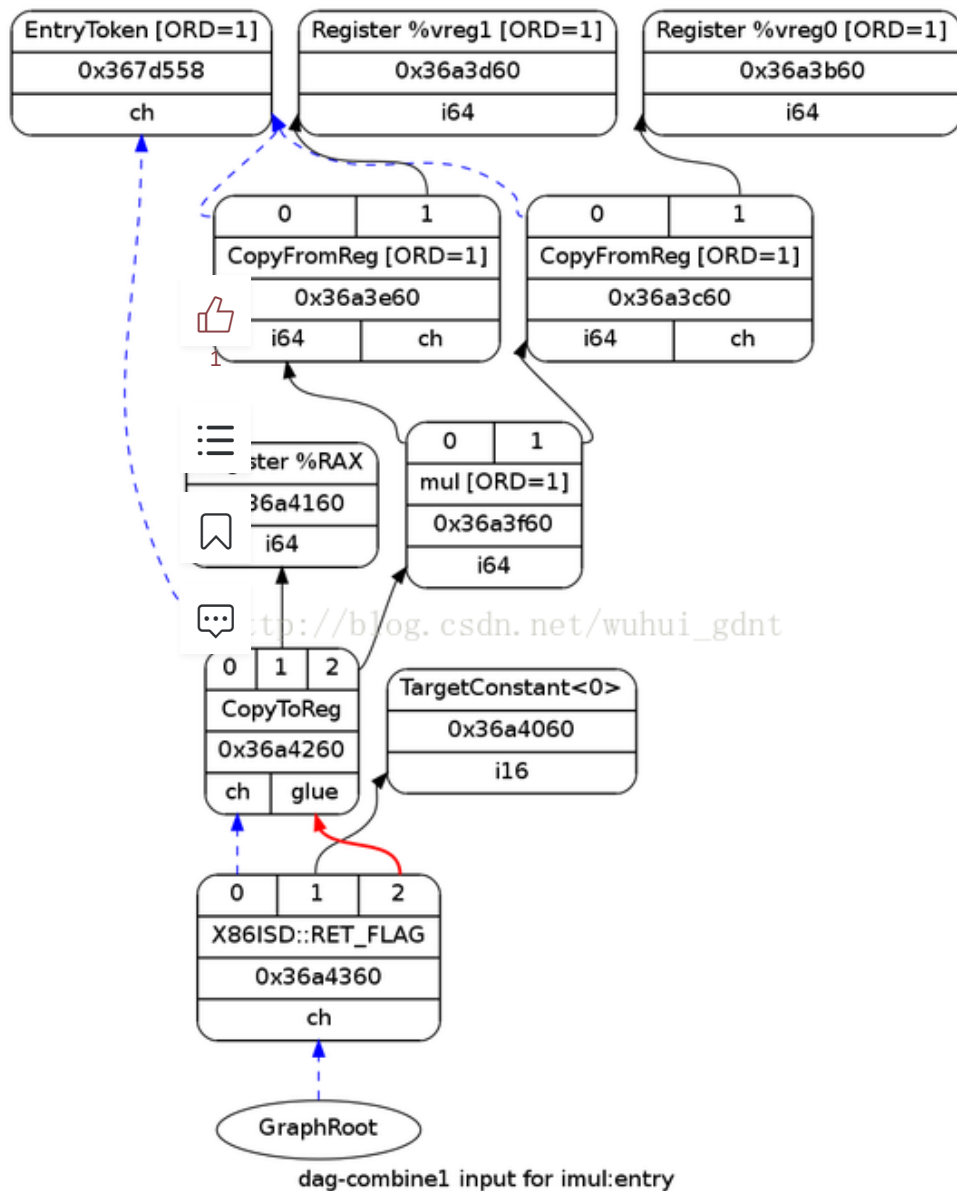
2018年3月	5篇
2018年2月	2篇
2018年1月	4篇
2017年12月	8篇
2017年11月	7篇
2017年10月	2篇
展开	

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册





这里没有什么特别有趣的东西，对于目标机器架构所有的类型都是合法的；因此，这也是到达指令选择阶段的DAG

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

## 他的热门文章

DWARF调试格式的简介

13548

LLVM

9474

GCC-3.4.6源代码学习笔记（1）

6612

DWARF调试格式的简介（续完）

4985

LLVM中指令的一生

4172

ELF对线程局部储存的处理（1）

3251

C++11线程，亲合与超线程

3247

autoconfig手册（1）

3245

每个C程序员应该知道的未定义行为#1/3

3204

位置无关代码

3169

# 指令选择的模式

指令选择是代码生成阶段最重要的部分（对此有争议）。它的任务是将一个合法的selection DAG转换为一个目标机器码形式的新DAG。换言之，抽象的、机器无关的输入必须被匹配到具体的、机器相关的输出。对此，LLVM使用了一个优雅的模式匹配算法，它包括了两大步骤。

第一步发生在（offline），在LLVM在进行自身构建时，涉及到了TableGen工具，它从指令定义产生模式匹配表。TableGen是LLVM生态系统的一个重要部分，在指令选择中它扮演了一个特别关键的角色，因此值得花几分钟来介绍它（也有从[TableGen fundamentals](#)开始的官方文档）。

TableGen的问题：它的某些应用实在太复杂（而指令选择，正如我们很快会看到的，是其中一个最恶劣的冒犯者），很容易搞错。它的核心是很简单的想法。很久之前，LLVM的开发者意识到开发每个新的目标机器需要编写大量的重复代码。以机器指令为例。一条指令被用在代码生成、汇编器、反汇编器、优化器及其他许多地方。每个这样的应用生成了一个将指令映射到某块信息的一张“表”。如果我们可以在一个集中位置定义所有的指令，收集我们需要的关于它们的信息，然后自动地产生所有的表，不是很好吗？这就是TableGen与生俱来要做的事。

让我们看一个与本文有关的指令定义（摘自lib/Target/X86/X86InstrArithmetic.td并稍作修改）：

```
def IMUL64rr : RI<0xAF, MRMSrcReg, (outs GR64:$dst),

    (ins GR64:$src1, GR64:$src2),

    "imul{q}\t{$src2, $dst|$dst, $src2}",

    [(set GR64:$dst, EFLAGS,

        (X86smul_flag GR64:$src1, GR64:$src2))],

    IIC_IMUL64_RR>
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！



同城征婚网

望京写字楼



生肖猪运势

移民加拿大条件

## 联系我们



请扫描二维码联系客服

✉ webmaster@csdn.net

☎ 400-660-0108

👤 QQ客服 🗣 客服论坛

关于 招聘 广告服务 百度

©1999-2018 CSDN版权所有

京ICP证09002463号

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心

登录

注册



如果这看起来凌乱，不用担心，这正是应有的第一印象。为了提取出公用的代码且疯狂地保持硬实（preserve DRY），TableGen发展了一些先进的特性，像多重继承，模板化的形式，等等；所有这些使得定义一开始有些难以理解。如果你希望看到IMUL64rr“裸露的”定义，你可以在LLVM源代码树的根节点运行这个命令：

```
$ llvm-tblgen lib/Target/X86/X86.td -I=include -I=lib/Target/X86
```

13.5MB仅包含`tblgen`的输出——TableGen后端可以从中获取所需的表项。IMUL64rr的def拥有大约75个域。不过我们只将<sup>1</sup>文本所需要的那些，上面所贴的扼要描述足够了。

对于我们的讨论，最重要的域是上面def中的第六个模板参数：

```
[(set GR64:$dst, EFLAGS,
  (X86smul_flag GR64:$src1, GR64:$src2))],
```

这是IMUL64rr被选中的模式。它实际上是一个描述要匹配的DAG路径的s-expression。在这里它大意为：一个带有两个64位GPR（通用寄存器）的X86ISD::SMUL节点（这被隐藏在X86smul\_flag定义之后）被调用并返回两个值——一个被赋给一个目标GPR，另一个赋给状态标记寄存器（虽然状态标记寄存器在x86里是隐含的（没有你可以操作的明确的寄存器），LLVM把它处理作明确的以辅助代码生成算法）。当自动化指令选择在DAG中看到这样一个序列，将把它匹配到上述的IMUL64rr指令。

在这里，仔细的读者会注意到我撒了点小谎。如果这个模式匹配的节点是X86ISD::SMUL，那么它如何匹配上面所示的包含一个ISD::MUL节点的DAG呢？确实，它不能。很快我会展示实际匹配这个DAG的模式，不过我觉得展示带有模式的指令定义很重要，使我后面能够讨论所有的模式如何交织在一起。

那么ISD::MUL与X86ISD::SMUL之间有什么差别呢（X86ISD::SMUL是ISD::SMULO通用节点特定于X86的降级）？在前者，我们不关心乘法实际影响的标记，而后者我们关心。在C，就乘法而言，我们通常不关心受影响的标记，因此选择了ISD::MUL。但LLVM提供了某些特殊的固有函数，比如`llvm.smul.with.overflow`，操作可以返回一个溢出标记。对于这些（连同其他可能的使用），X86ISD::SMUL节点应运而生（对此你可能有“噢！天，为什么这如此复杂？”的反应。简要的回答是“编译器很难，让我们钓鱼去吧”。一个更长的理由则是：x86指令集非常复杂。另外，LLVM是一个带有许多（相当不同）目标机器的编译器，其许多手段被设计作目标机

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册



0 KLOC左右的C++降级代码，对比包含3000页左右的Intel架构手册。就Kolmogorov复杂度来说，这不能算太坏)

这里是什么匹配ISD::MUL节点呢？这个模式来自lib/Target/X86/X86InstrCompiler.td：

```
def : Pat<(mul GR64:$src1, GR64:$src2),
  (: 4rr GR64:$src1, GR64:$src2)>;
```

这是一个匿名的TableGen记录，它定义了一个脱离了任何特定指令的“模式”。这个模式只是一个从DAG输入到DAG输出的映射，后者包含一条选中的指令。我们不关心这个映射叫什么，因此TableGen让我们定义匿名的实例。在这个情形里，该模式应该是相当简单的。下面是来自include/llvm/Target/TargetSelectionDAG.td的一段有趣的片段，其定义了类Pattern（连同它的特化Pat）：

```
// Selection Pattern Support.

//

// Patterns are what are actually matched against by the target-flavored

// instruction selection DAG. Instructions defined by the target implicitly

// define patterns in most cases, but patterns can also be explicitly added when

// an operation is defined by a sequence of instructions (e.g. loading a large

// immediate value on RISC targets that do not support immediates as large as

// their GPRs).

//
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册



```

class Pattern<dag patternToMatch, list<dag> resultInstrs> {

    dag          PatternToMatch  = patternToMatch;

    list<dag>     ResultInstrs    = resultInstrs;

    list<Predicate> Predicates    = []; // See class Instruction in Target.td.

    int           AddedComplexity = 0;   // See class Instruction in Target.td.

}

// Pat - A simple (but common) form of a pattern, which produces a simple result
// not needing a full list.

```

```

class Pat<dag pattern, dag result> : Pattern<pattern, [result]>;

```

片段开头的大段注释是有帮助的，但它描述了与我们在IMUL64rr观察到的实际相反的情形。在我们的情形里，定义在指令里的模式实际上更为复杂，而基本的模式以Pattern定义在外面。

## 模式匹配机制

TableGen目标机器指令描述支持多种模式类型。我们已经研究了指令定义中隐含定义的模式，以及单独显式定义的模式。另外，还有指定要调用的C++函数的“复杂”模式，以及包含任意C++代码片段执行定制匹配的“模式片段（pattern fragments）”。如果你有兴趣，这些模式类型在include/llvm/Target/TargetSelectionDAG.td的注释中做了稍许描述。

在TableGen中混合C++代码可行，是因为TableGen（与特定DAG ISel后端）运行的最终结果是一个嵌入到一个

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册





更具体些，这个序列是：

- 通用的SelectionDAGISel::DoInstructionSelection方法对每个DAG节点调用Select。
- Select是一个抽象方法，由目标机器实现。例如，X86DAGToDAGISel::Select。
- 后者拦截某些节点进行手工匹配，但向X86DAGToDAGISel::SelectCode委托了大量的工作。
- X86DAGToDAGISel::SelectCode由TableGen自动生成（它生成在一个由lib/Target/X86/X86ISelDAGToDAGISel.cpp包含的文件<BUILD\_DIR>/lib/Target/X86/X86GenDAGISel.inc中），包含匹配表（matcher table），随后将该表作为参数调用通用的SelectionDAGISel::SelectCodeCommon。

那么匹配表是什么呢？本质上，它是一个针对指令选择，以某种“字节码”写成的程序。为了实现灵活的模式匹配同时保持效率，TableGen把所有的模式合并起来产生一个程序，给定一个DAG样式（mode），找出它匹配哪个模式（pattern）。SelectionDAGISel::SelectCodeCommon作为这个字节码的解析器。

不幸的是，用模式匹配的字节码语言没有文档记录。为了理解它如何工作，除了查看解析器的代码及为某个后端产生的字节码外，别无他法（如果你希望理解这个字节码如何从TableGen的模式定义产生，你还需要看TableGenDAGISel后端）。

## 例子：匹配我们的案例DAG节点

让我们研究一下我们案例DAG中的ISD::MUL是如何匹配的。出于这个目的，向llc传递-debug选项是有用的，它使llc在代码生成过程中转储详细的调试信息。特别的，可以追踪每个DAG节点的选择过程。下面是我们ISD::MUL节点的相关部分：

```
Selecting: 0x38c4ee0: i64 = mul 0x38c4de0, 0x38c4be0 [ORD=1] [ID=7]
```

```
ISEL: Starting pattern match on root node: 0x38c4ee0: i64 = mul 0x38c4de0, 0x38c4be0 [ORD=1]
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册





Match failed at index 57922

Continuing at 58133

Match failed at index 58137

Continuing at 58246



Match failed at index 58249

Continuing at 58335



TypeSwitch [ ] from 58337 to 58380



MatchAddress: 6ISelAddressMode 0x7fff447ca040



Base\_Reg nul Base.FrameIndex 0

Scale1

IndexReg nul Disp 0

GV nul CP nul

ES nul JT-1 Align0

Match failed at index 58380

Continuing at 58396

Match failed at index 58407

Match failed at index 58517

Continuing at 58531

Match failed at index 58532

Continuing at 58544



Match failed at index 58545

Continuing at 58557



Morphed node 0x38c4ee0: i64,i32 = IMUL64rr 0x38c4de0, 0x38c4be0 [ORD=1]



ISEL: Match complete!

=> 0x38c4ee0: i64,i32 = IMUL64rr 0x38c4de0, 0x38c4be0 [ORD=1]

这里提到的索引援引匹配表。在生成文件X86GenDAGISel.inc每行开头的注释里，你可以看到它们。下面是这个表的开头（注意表中的值与我为这个例子所构建的LLVM的版本（r174056）是相关的。X86模式定义的变化可能导致不同的编号，但原理是相同的）：

```
// The main instruction selector code.
```

```
SDNode *SelectCode(SDNode *N) {
```

```
    // Some target values are emitted as 2 bytes, TARGET_VAL handles
```

```
    // this.
```

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册



```
static const unsigned char MatcherTable[] = {

/*0*/      OPC_SwitchOpcode /*221 cases */, 73|128,103/*13257*/,  TARGET_VAL(ISD::STORE),// ->

/*5*/      OPC_RecordMemRef,

/*6*/      RecordNode,    // #0 = 'st' chained node

/*7*/      OPC_Scope, 5|128,2/*261*/, /*->271*/ // 7 children in Scope
```

在位置0我们有一个 OPC\_SwitchOpcode 操作，它是相当于节点操作码的一个大的switch表。它跟有一组case。每个case以其开始（这样匹配器知道如果这个case匹配失败要去哪里），然后是操作码。例如，正如在上面你会看到的，表中第一个case用于操作码ISD::STORE，其大小是13257（这个尺寸以一个特殊的变长编码方式编码，因为它是4字节组织的）。

看一下调试输出，我们MUL节点的匹配始于偏移57917。下面是表中相关的部分：

```
/*SwitchOpcode*/ 53|128,8/*1077*/,  TARGET_VAL(ISD::MUL),// ->58994

/*57917*/      OPC_Scope, 85|128,1/*213*/, /*->58133*/ // 7 children in Scope
```

正如期待的，这是以ISD::MUL作为操作码的switch case。这个case的匹配始于OPC\_Scope，它是让解析器压入当前状态的一条指令。如果在这个域里某个case失败了，可以恢复当前状态，进而匹配下面的case。在上面的片段里，如果该域中的匹配失败，将前进到偏移58133。

在调试输出里你可以看到发生了这些：

```
Initial Opcode index to 57917
```

```
Match failed at index 57922
```

在57922，解析器尝试匹配该节点的孩子到一个ISD::LOAD（意思是——与内存内参数相乘），失败，按域指示跳到58133。类似的，余下的匹配过程可以被追踪——跟随调试输出并以匹配表为参考。在偏移58337发生了一些有趣的事情。下面是表相关的部分：

```

/*58337*/      OPC_SwitchType /*2 cases */, 38, MVT::i32, // ->58378

/*58340*/      OPC_Scope, 17, /*->58359*/ // 2 children in Scope
/*58342*/      1 OPC_CheckPatternPredicate, 4, // (!Subtarget->is64Bit())
/*58344*/      :: OPC_CheckComplexPat, /*CP*/3, /*#*/0, // SelectLEAAddr:$src #1 #2 #3 #4 #5
/*58347*/      OPC_MorphNodeTo, TARGET_VAL(X86::LEA32r), 0,
                1/*#VTs*/, MVT::i32, 5/*#Ops*/, 1, 2, 3, 4, 5,
                // Src: lea32addr:i32:$src - Complexity = 18
                // Dst: (LEA32r:i32 lea32addr:i32:$src)

```

这是上面描述的一个复杂模式的结果。SelectLEAAddr是一个C++方法（由X86后端的ISel实现），它被尝试调用来匹配节点操作数到一个LEA（某些乘法可以被优化为使用更快的LEA指令）。跟在其后的调试输出来自该方法，并正如我们看到的，最终失败。

最后，解析器到达偏移58557，匹配成功。下面是表相关的部分：

```

/*58557*/      /*Scope*/ 12, /*->58570*/

/*58558*/      OPC_CheckType, MVT::i64,


/*58560*/      OPC_MorphNodeTo, TARGET_VAL(X86::IMUL64rr), 0,

```

```
// Src: (mul:i64 GR64:i64:$src1, GR64:i64:$src2) - Complexity = 3
```

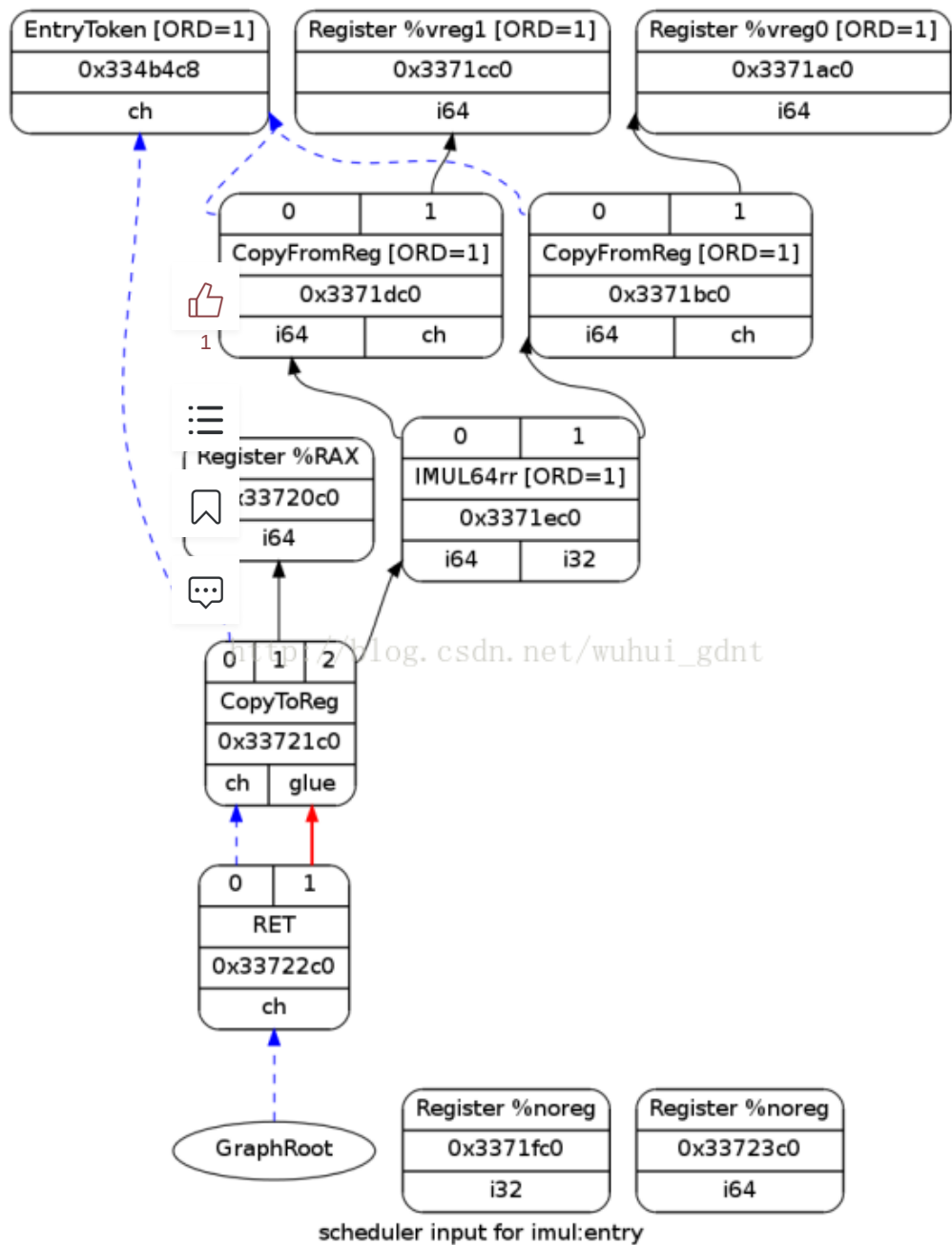
```
// Dst: (IMUL64rr:i64:i32 GR64:i64:$src1, GR64:i64:$src2)
```

简而言之，在一大串优化及特殊情形的匹配失败后，匹配器最终使用匹配到IMUL64rr 机器指令的64位寄存器间的通用整数乘法。

如果追踪记录  指令选择器卖力查找一条合适的指令，那是真的。要产生好的代码，在妥协到通用指令序列之前，必须进行一些工作来尝试匹配各种优化的指令序列。在文本的下一部分，我将说明带有优化的指令选择的某些先进的

## 最终代码

下面是指令选择器DAG的外观：



因为入口DAG是相当基本的，这个非常类似；主要的区别是对实际指令选中乘法与返回节点。

如果你记得[LLVM中指令的一生](#)这篇文章，在被指令选择器匹配后，这条指令经历了几个另外的化身。最终流出的代码是：

```
imul:                                # @imul
```

```
    imulq    si, %rdi
```

```
    movq     %rdi, %rax
```

```
    ret
```

imulq是X86-64的汇编表示（GAS形式）。它把函数的参数相乘（根据AMD64 ABI，头两个整数进入%rsi与%rdi）；结果被移到返回寄存器——%rax。

## 结论

本文提供了指令选择过程——LLVM代码生成器的一个关键部分的一个深入的窥探。尽管它使用了一个相对简单的例子，它应该包含足够的信息获得所涉及机制的一些初步认识。本文的下一部分，我将调查另外几个例子，通过它们，代码生成过程的其他方面将更为清晰。

👤 目前您尚未登录，请 [登录](#) 或 [注册](#) 后进行评论

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)

[注册](#)





## 编译器架构的王者LLVM——（3）用代码生成代码



sun\_xiaofan

2015年11月06日 20:56

4016

LLVM的开发思路很简单，就是用C++代码去不断生成llvm字节码

## LLVM中TableGen工具的使用



u010902721

2014年10月18日 20:42

2471

在描述处理器结构时 需要



1

## 紧跟新技术，程序员随我来！

技术提升不再难！



## LLVM Essentials-Packt 2016（读书笔记）：TableGen讲解并不透彻，另外我还想知道...

LLVM Essentials 目录 [隐藏] 1 Playing with LLVM2 Building LLVM IR3 高级IR4 基本IR...



cteng

2016年02月18日 15:35

1372

## LLVM（三）：Tablegen简介



windiwen

2014年03月04日 10:03

1303

上一篇介绍实现llvm后端需要做的一些工作，有很大一部分工作是描述目标体系结构的特征，包括指令集，寄存器等信息。Tablegen就是用于记录这些信息的描述性语言。目标体系结构目录下的\*.td文件都是用...

## LLVM TableGen介绍



dreammeard

2014年02月19日 17:04

1947

1. TableGen简介 TableGen是用来帮助程序员开发和维护某一领域想逛的信息记录。它定义了一套描述这些信息记录的数据结构和语法规则。编写代码时，如果使用TableGen描述信息记录，我们可...

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)[注册](#)

领红包，享5折，新购满1000再返券，最高可返6000元



## llvm生成rdrand指令



pang241 2017年11月18日 09:59 138

问题在做项目的时候需要使用llvm的pass对函数进行插桩，在每一个函数头之前插入一条指令 `rdrand %rax`，在寻找llvm基本指令之后发现并没有生成随机数的指令，这时就想到了ll...

## LLVM后端开发



jinweifu 2017年01月06日 08:20 1885

LLVM后端 介绍过 当描述了编写编译器后端的技术，将llvm IR转化为定制的机器代码或者其他语言。意图生成的特定机器码可以是汇编形式或者二进制的形式（能够被JIT编译器使用）。LLVM的后端有一个...

## LLVM中指令的一生



wuhui\_gdnt 2014年11月10日 11:28 4178

原作：<http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm/> 作者：Eli Bendersky LLVM是...

## LLVM代码生成器进一步深入，第一部分



wuhui\_gdnt 2014年11月12日 15:59 2681

原作：<http://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1/> 作者：El...

## LLVM每日谈之一 LLVM是什么



sns1984 2012年10月02日 00:03 47191

作者：sns1984 写在前面的话：最近接触llvm比较多，在这个上面花了不少的时间。感觉llvm要完全理解透是个很不容易的事情，需要在学习过程中好好的整理下自己的思路。刚好又阅读了开源项目Sto...

## LLVM学习笔记（8）



wuhui\_gdnt 2017年04月12日 11:47 523

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册



## 恭喜：一个公式教你秒懂天下英语

老司机教你一个数学公式秒懂天下英语



## 编译器架构师 LLVM——（11）深入理解GetElementPtr

LLVM平台，短短几年间，改变了众多编程语言的走向，也催生了一大批具有特色的编程语言的出现，不愧为编译器架构的王者，也荣获2012年ACM系统奖——题记 LLVM平台，和C语言极为类似，强类...

 sun\_xiaofan 2015年12月27日 20:15 10725



## LLVM Programmer's Manual---阅读笔记

 sns1984 2012年10月17日 23:15 3393

文档地址：<http://llvm.org/docs/ProgrammersManual.html> 该文档的主要目的：该文档主要介绍了LLVM源码的一些重要的类和接口，并不打算解...

## LLVM学习笔记——目录

 wuhui\_gdnt 2017年03月10日 11:14 697

前言2011年前后，GCC后端代码的阅读陷入了举步维艰的境地。GCC-3.4.6后端代码的可读性不友好（当前版本没看过，不予评价。不过据说4.0进行的重构，应该会好些），充斥着动辄数千行的函数，包含着...

## LLVM学习笔记（16）

 wuhui\_gdnt 2017年08月18日 12:05 220

3.4.2.4. PatFrag的处理 3.4.2.4.1. 模式树的构建 PatFrag是一个可重用的构件，TableGen会在PatFrag出现的地方展开其定义，有点像C/C++中...

## Clang以及LLVM研究

 skylin19840101 2017年05月23日 20:39 287

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

登录

注册



## 《深入理解LLVM》第一章 LLVM简介



sns1984

2015年07月24日 11:54

19457

第一章 LLVM简介作者：史宁宁1.1 LLVM是什么LLVM是什么？这是一个虽然基础，但是也曾让很多新入门的人迷惑的一个问题。从字面上来讲，LLVM(Low Level Virtual Mach...

## 美国杰出人才绿卡

八种中国人获得绿卡的方式

1

百度广告



## 从今天起，写一本关于LLVM的书----《深入理解LLVM》



一直想写一本关于深入学习LLVM的书，这个想法有了很久了，但是一直没有机会动手。现在虽然很忙，但是依然觉的有必要马上动手去做这个事情。可事情都是一点一点积累起来的，如果一直不动手，什么都做不成。还有...



sns1984

2015年03月07日 22:56

7699



## 使用 LLVM 框架创建有效的编译器，第 2 部分



novelly

2013年11月26日 23:36

791

使用 clang 预处理 C/C++ 代码 无论您使用哪一种编程语言，LLVM 编译器基础架构都会提供一种强大的方法来优化您的应用程序。在这个两部分系列的第二篇文章中，了解在 L...

## start.S进一步、更详细的、深入的解释和分析 2013.04.26更新（四）

转自：<http://zqwt.012.blog.163.com/blog/static/12044684201332662650711/> | 下面这几行代码的作用是定义了一些宏，给寄存器赋值（ ...



njuitf

2014年03月04日 15:26

1189

加入CSDN，享受更精准的内容推荐，与500万程序员共同成长！

[登录](#)[注册](#)