学习TableGen

邢明杰

2019-12-18

目 录

TableGen语言

- DSL（Domain Specific Language）语言
- 处理领域特定的信息记录（Records）
- LLVM用它来描述：
  - 寄存器信息
  - 指令信息
  - 调用约定
  - 指令调度信息
  - Intrinsics
  - 命令行选项
  - 等等
- 也在Clang中使用

td文件

- Taget Description（目标描述）文件
- 使用TableGen语言编写
- `$ find llvm -name *.td`
  - llvm/include/llvm/IR/
  - llvm/include/llvm/Target
  - llvm/lib/Target/*/
  - llvm/tools/*/
  - 等等
- 语法高亮
  - emacs: llvm/utils/emacs
  - vim: llvm/utils/vim

foo.td

llvm-tblgen工具

- DSL编译器
  - 解析td文件（语言前端）
  - 实例化（IR/Records）
  - 交给特定后端处理（代码生成）
- $llvm-tblgen –help
  - 缺省为打印所有Record
  - 通过命令行选项指定代码生成后端

目 录

TableGen语言介绍

- 注释（C/C++风格）

```
/* This is a comment. */

// This is a comment.
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
  - 整数
  - 字符串、代码片段
  - 单个bit、多个bits
  - 列表
  - DAG（有向无环图）
  - class

类型	描述
bit	bool值，存放0或者1
int	32bit整数值
string	字符串
code	代码片段，字符串文字
bits<n>	长度为n个bit的整数
list<type>	列表
Class type	Class类
dag	有向无环图

TableGen语言介绍

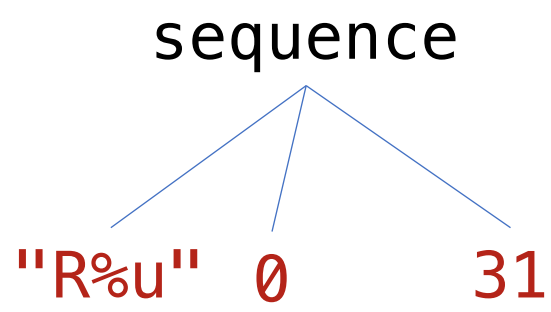
- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
  - 常量值

值	描述
?	未初始化域
0b1001011	二进制整数，size大小有给定的bit个数来确定，不会进行隐式的扩展/截取
7	十进制整数
0x7F	十六进制整数
“foo”	（单行）字符串
[{ … }]	代码片段，多行字符串文字
[X, Y, Z]<type>	列表值，<type>表示元素类型，通常为可选，当无法推断出元素类型的时候需要显式给出
(DEF a, b)	DAG值，第1个元素为def引用，是DAG图的操作符（operator）

DAG示例

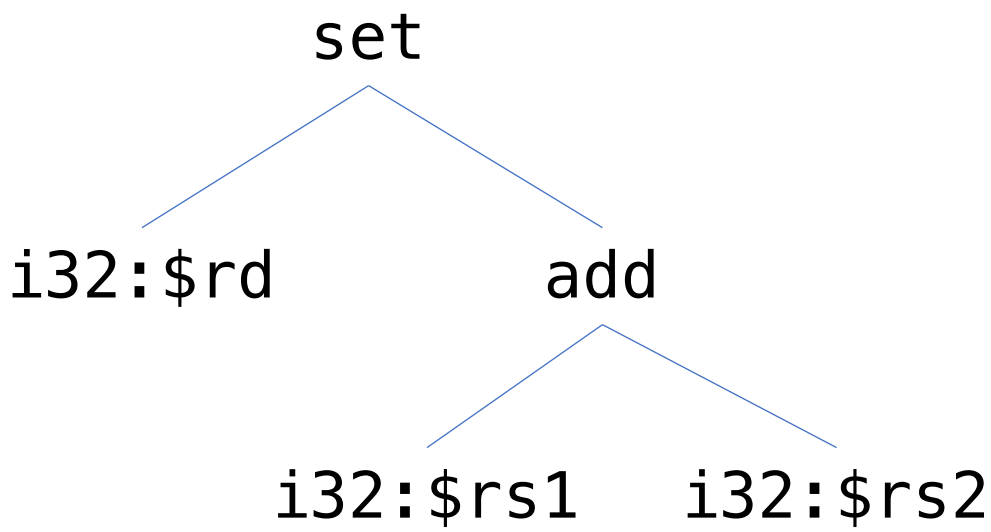
operator operand

```
(sequence "R%u", 0, 31)
```



operand name

```
(set i32:$rd, (add i32:$rs1, i32:$rs2))
```



TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
  - 常量值
  - 各种表达式

表达式	描述
value{17}	访问值的1个bit
value{15-17}	访问值的1个有序bit序列，value{15-17}和value{17-15}所产生的序列顺序相反
X.Y	访问值的域
list[4-7,17,2-3]	列表的切片，元素可以出现多次
{a, b, 0b10}	bits<4>的初始化，1个bit来自a，1个bit来自b，2个bit来自0b10
str1#str2	字符串拼接

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
  - 常量值
  - 各种表达式

表达式	描述
!add(a,b,...)	算术加法
!and(a,b,...)	按位与
!or(a,b,...)	按位或
!shl(a,b)	左移
!srl(a,b)	逻辑右移
!sra(a,b)	算术右移
!le(a,b)	小于等于
!lt(a,b)	小于
!ge(a,b)	大于等于
!gt(a,b)	大于

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
  - 常量值
  - 各种表达式

表达式	描述
!eq(a,b)	等于，用来比较string，int和bit对象
!ne(a,b)	不等于
!strconcat(a, b, ...)	字符串拼接
!subst(a, b, c)	字符串替换
!cast<type>(a)	类型转换
!isa<type>(a)	类型检测，如果a为给定type，则返回1
!if(a,b,c)	条件表达式，如果a非零则返回b，否则返回c

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
  - 常量值
  - 各种表达式

表达式	描述
!head(a)	列表a中的第1个元素
!tail(a)	列表a中的除去第1个之外的所有元素
!empty(a)	判断列表a是否为空
!size(a)	列表a的元素个数
!listconcat(a, b, ...)	列表拼接，列表的元素类型必须相同
!con(a, b, ...)	DAG拼接，DAG图的操作符必须相同
!dag(op, children, names)	创建一个带有操作数名字的DAG
!foreach(a, b, c)	遍历DAG图或列表b，执行操作c
!foldl(start, lst, a, b, expr)	左折叠替换, f(a,b) -> f(…f(f(start, lst[0]), lst[1]), …), lst[n-1])’

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME

```
class C { bit V = 1; }
```

class C

NAME	V
?	1

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承

```
class C { bit V = 1; }
def X : C;
```

class C

NAME	V
?	1

def X

NAME	V
?	1

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承

```
class C { bit V = 1; }
def X : C;
def Y : C {
  string Greeting = "hello";
}
```

class C

NAME	V
?	1

def X

NAME	V
?	1

def Y

NAME	V	Greeting
?	1	“hello”

例子：class 和 def

- $ llvm-tblgen class.td

```
class C { bit V = 1; }
def X : C;
def Y : C {
  string Greeting = "hello";
}
```

```
------------- Classes -----------------
class C {
  bit V = 1;
  string NAME = ?;
}
------------- Defs -----------------
def X { // C
  bit V = 1;
  string NAME = ?;
}
def Y { // C
  bit V = 1;
  string Greeting = "hello";
  string NAME = ?;
}
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承
  - 支持覆写（override）父类成员的值（let表达式）

```
class C { bit V = 1; }
class D : C { let V = 0; }
def Z : D;
```

class C

NAME	V
?	1

class D

NAME	V
?	0

def Z

NAME	V
?	0

例子：let

- $ llvm-tblgen let.td

```
class C { bit V = 1; }
class D : C { let V = 0; }
def Z : D;
```

```
------------- Classes -----------------
class C {
  bit V = 1;
  string NAME = ?;
}
class D {       // C
  bit V = 0;
  string NAME = ?;
}
------------- Defs -----------------
def Z { // C D
  bit V = 0;
  string NAME = ?;
}
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承
  - 支持覆写父类成员的值（let表达式）

```
class C { bit V = 1; let NAME = "C"; }
def X : C;
def Y : C {
  string Greeting = "hello";
  let NAME = "Y";
}
```

class C

NAME	V
“C”	1

def X

NAME	V
“C”	1

def Y

NAME	V	Greeting
“Y”	1	“hello”

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承
  - 支持覆写父类成员的值
  - 支持模版参数

```
class FPFormat<bits<2> val> {
  bits<2> Value = val;
}
def NotFP      : FPFormat<0>;
def ZeroArgFP  : FPFormat<1>;
def OneArgFP   : FPFormat<2>;
def TwoArgFP   : FPFormat<3>;
```

class FPFormat

NAME	Value
?	{ FPFormat:val{1}, FPFormat:val{0} }

def NotFP, ZeorArgFP

NAME	Value
?	{ 0, 0 }

NAME	Value
?	{ 0, 1 }

def OneArgFP , TwoArgFP

NAME	Value
?	{ 1, 0 }

NAME	Value
?	{ 1, 1 }

例子：template

- $ llvm-tblgen template.td

```
class FPFormat<bits<2> val> {
  bits<2> Value = val;
}
def NotFP      : FPFormat<0>;
def ZeroArgFP  : FPFormat<1>;
def OneArgFP   : FPFormat<2>;
def TwoArgFP   : FPFormat<3>;
```

```
------------- Classes -----------------
class FPFormat<bits<2> FPFormat:val = { ?, ? }> {
  bits<2> Value = { FPFormat:val{1}, FPFormat:val{0} };
  string NAME = ?;
}
------------- Defs -----------------
def NotFP {     // FPFormat
  bits<2> Value = { 0, 0 };
  string NAME = ?;
}
def OneArgFP {  // FPFormat
  bits<2> Value = { 1, 0 };
  string NAME = ?;
}
def TwoArgFP {  // FPFormat
  bits<2> Value = { 1, 1 };
  string NAME = ?;
}
def ZeroArgFP { // FPFormat
  bits<2> Value = { 0, 1 };
  string NAME = ?;
}
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
  - 类似C++语法
  - 有1个隐式的NAME
  - 支持继承
  - 支持覆写父类成员的值
  - 支持模版参数
  - 支持一次定义多个def（multiclass, defm）

```
class C<string n, bit v> { let NAME = n; bit Value = v; }
multiclass MC<string n> {
  def _v0 : C<n, 0>;
  def _v1 : C<n, 1>;
}
defm X : MC<"x">;
defm Y : MC<"y">;
```

class C

NAME	Value
C:n	C:v

def X_v0, X_v1

NAME	Value
"X"	0

NAME	Value
"X"	1

def Y_v0, Y_v1

NAME	Value
"Y"	0

NAME	Value
"Y"	1

例子：multiclass

- $ llvm-tblgen multiclass.td

```
class C<string n, bit v> {
  let NAME = n;
  bit Value = v;
}

multiclass MC<string n> {
  def _v0 : C<n, 0>;
  def _v1 : C<n, 1>;
}

defm X : MC<"x">;
defm Y : MC<"y">;
```

```
------------- Classes -----------------
class C<string C:n = ?, bit C:v = ?> {
  bit Value = C:v;
  string NAME = C:n;
}
------------- Defs -----------------
def X_v0 {      // C _v0
  bit Value = 0;
  string NAME = "X";
}
def X_v1 {      // C _v1
  bit Value = 1;
  string NAME = "X";
}
def Y_v0 {      // C _v0
  bit Value = 0;
  string NAME = "Y";
}
def Y_v1 {      // C _v1
  bit Value = 1;
  string NAME = "Y";
}
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
- include

```
include "llvm/Target/Target.td"
```

TableGen语言介绍

- 注释（C/C++风格）
- 类型（强类型）
- 值和表达式
- class和def（都属于record）
- include
- foreach

```
class Reg<bits<4> v, string n> {
  bits<4> Enc = v;
  let NAME = n;
}
foreach i = [0, 1, 2, 3] in {
  def R#i : Reg<i, "r"#i>;
}
```

class Reg

NAME	Enc
Reg:n	{ Reg:v{3}, Reg:v{2}, Reg:v{1}, Reg:v{0} }

def R0, R1

NAME	Enc
"r0"	{ 0, 0, 0, 0 }

NAME	Enc
"r1"	{ 0, 0, 0, 1 }

def R2, R3

NAME	Enc
"r2"	{ 0, 0, 1, 0 }

NAME	Enc
"r3"	{ 0, 0, 1, 1 }

例子：foreach

- $ llvm-tblgen foreach.td

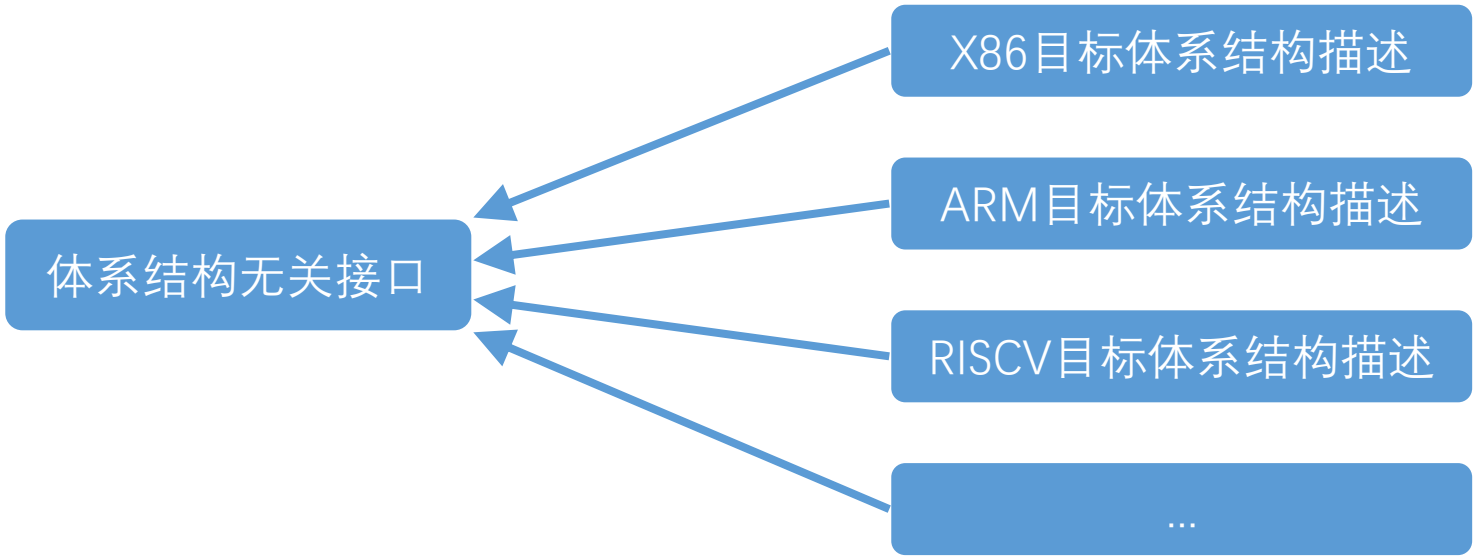
```
class Reg<bits<4> v, string n>
{
  bits<4> Enc = v;
  let NAME = n;
}


foreach i = [0, 1, 2, 3] in {
  def R#i : Reg<i, "r"#i>;
}
```

```
------------- Classes -----------------
class Reg<bits<4> Reg:v = { ?, ?, ?, ? }, string
Reg:n = ?> {
  bits<4> Enc = { Reg:v{3}, Reg:v{2}, Reg:v{1},
Reg:v{0} };
  string NAME = Reg:n;
}
------------- Defs -----------------
def R0 {          // Reg
  bits<4> Enc = { 0, 0, 0, 0 };
  string NAME = "r0";
}
def R1 {          // Reg
  bits<4> Enc = { 0, 0, 0, 1 };
  string NAME = "r1";
}
def R2 {          // Reg
  bits<4> Enc = { 0, 0, 1, 0 };
  string NAME = "r2";
}
def R3 {          // Reg
  bits<4> Enc = { 0, 0, 1, 1 };
  string NAME = "r3";
}
```

目 录

体系结构无关接口

- 体系结构无关接口：include/llvm/Target/Target.td 等等
- 目标体系结构描述：lib/Target/RISCV/RISCV.td 等等



目标体系结构描述

■ include/llvm/Target/Target.td 等等

```
class Target
class SubtargetFeature
class RegisterClass
class Register
class Operand
class Instruction
...
```

体系结构无关接口

继承关系

■ lib/Target/RISCV/RISCV.td 等等

```
def RISCV : Target
def FeatureStdExtM : SubtargetFeature
def GPR : RegisterClass
class RISCVReg : Register
def uimm5 : Operand
class RVInst : Instruction
...
```

目标体系结构描述

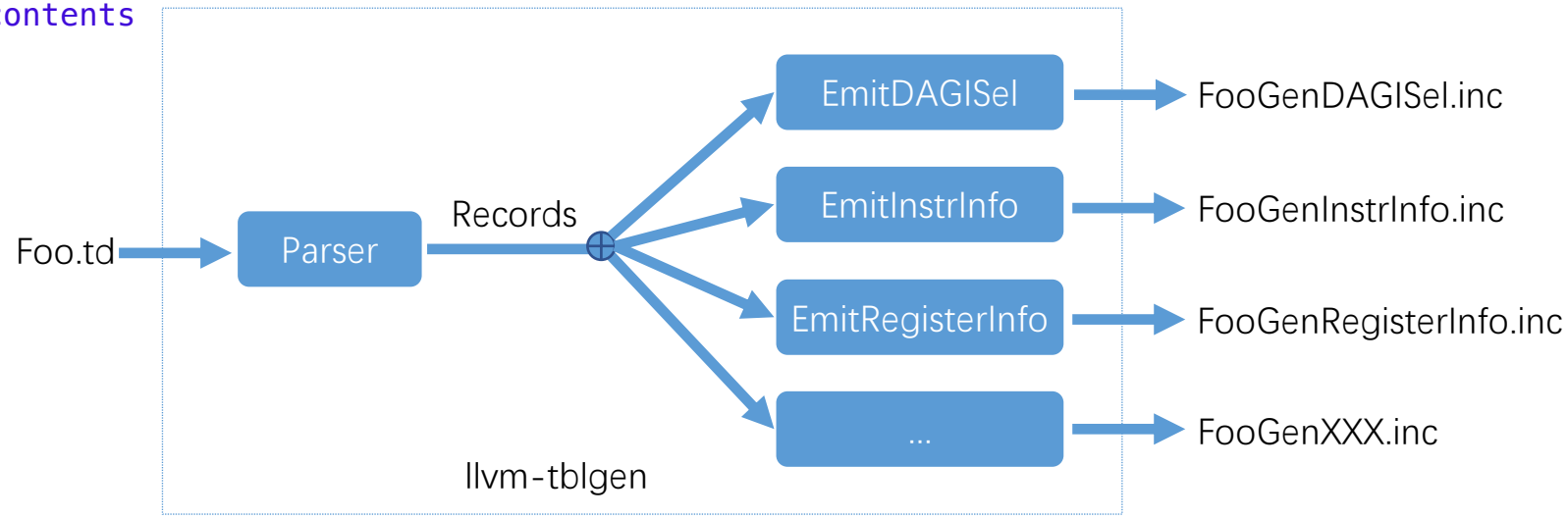
代码自动生成机制

- llvm-tblgen的代码实现
  - Parser：lib/TableGen和include/llvm/TableGen/
  - CodeGen：utils/TableGen/

```
bool LLVMTableGenMain(raw_ostream &OS, RecordKeeper &Records) {
  switch (Action) {
  case PrintRecords:
    OS << Records; // No argument, dump all contents
    break;
  case DumpJSON:
    EmitJSON(Records, OS);
    break;
  case GenEmitter:
    EmitCodeEmitter(Records, OS);
    break;
  case GenRegisterInfo:
    EmitRegisterInfo(Records, OS);
    break;
  case GenInstrInfo:
    EmitInstrInfo(Records, OS);
    break;
...
```

utils/TableGen/TableGen.cpp

- 生成的inc文件
  - 位于build/lib/Target/相应目录下
  - 在llvm的其它C++源文件中用include包含进来



例子：Foo.td

```
//===----------------------------------------------------------------------===//
// Target-independent interfaces which we are implementing
//===----------------------------------------------------------------------===//
include "llvm/Target/Target.td"

//===----------------------------------------------------------------------===//
// Register File Description
//===----------------------------------------------------------------------===//
include "FooRegisterInfo.td"

//===----------------------------------------------------------------------===//
// Instruction Description
//===----------------------------------------------------------------------===//
include "FooInstrInfo.td"
def FooInstrInfo : InstrInfo;

...

//===----------------------------------------------------------------------===//
// Define the Foo target.
//===----------------------------------------------------------------------===//
def Foo : Target {
  let InstructionSet = FooInstrInfo;
}
```

一个简化的代码框架

寄存器信息

- 寄存器名、备用名
- 寄存器编码
- 寄存器数据类型、对齐方式
- 寄存器类别（用于寄存器分配）
- sub-register和super-register
- ...

class Register

字符串列表，缺省值为空

```
class Register<string n, list<string> altNames = []> {
  string Namespace = "";
  string AsmName = n;
  list<string> AltNames = altNames;
  ...
  // HWEncoding - The target specific hardware encoding for this register.
  bits<16> HWEncoding = 0;
  ...
}
```

寄存器备用名，比如：
RISCV的x2（architectural name）对应的sp（ABI name）

include/llvm/Target/Target.td

class RegisterClass

列表元素类型为 ValueType

```
class RegisterClass<string namespace, list<ValueType> regTypes, int alignment,
                    dag regList, RegAltNameIndex idx = NoRegAltName>
```

常用的dag：
(add ...)
(sequence ...)

在内存中的对齐方式

注： class ValueType在include/llvm/CodeGen/ValueTypes.td中定义。

dag操作，参见lib/TableGen/SetTheory.cpp。

FooRegisterInfo.td

```
class FooReg<bits<5> Enc, string n> : Register<n> {
  let HWEncoding{4-0} = Enc;
}

foreach i = 0-31 in
  def R#i  : FooReg<i, "r"#i>;

def GPR : RegisterClass<"Foo", [i32], 32, (sequence "R%u", 0, 31)>;
```

设置硬件编码

定义32个寄存器

定义1个寄存器类别

命名空间

数据类型

对齐方式

寄存器序列

注：i32在include/llvm/CodeGen/ValueTypes.td中定义

FooGenRegisterInfo.inc

- $ llvm-tblgen -gen-register-info –o FooGenRegisterInfo.inc ...

生成代码	描述
枚举值	物理寄存器枚举值 寄存器ClassID，等等
MC寄存器信息	寄存器差分表 寄存器字符串压缩表 寄存器类别字符串压缩表 寄存器编码表 InitFooMCRegisterInfo函数，等等
头文件代码片段	FooGenRegisterInfo（继承自TargetRegisterInfo） GPRRegClass的声明，等等
Target描述信息	GPRRegClass的定义 寄存器压力信息，等等

FooGenRegisterInfo.inc

- 通过宏来封装相应的代码片段

```
#ifdef GET_REGINFO_HEADER
#undef GET_REGINFO_HEADER
...
#endif // GET_REGINFO_HEADER
```

FooGenRegisterInfo.inc

```
#define GET_REGINFO_ENUM
#include ”FooGenRegisterInfo.inc"
```

FooMCTargetDesc.h

FooGenRegisterInfo.inc

```
enum {
  NoRegister,
  R0 = 1,
  R1 = 2,
  ...
  R31 = 32,
  NUM_TARGET_REGS        // 33
};
```

物理寄存器枚举值

```
enum {
  GPRRegClassID = 0,
};
```

寄存器类别枚举值

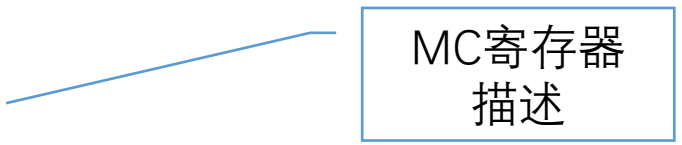
FooGenRegisterInfo.inc

```
extern const char FooRegStrings[] = {
  /* 0 */ 'R', '1', '0', 0,
  /* 4 */ 'R', '2', '0', 0,
  /* 8 */ 'R', '3', '0', 0,
  /* 12 */ 'R', '0', 0,
  /* 15 */ 'R', '1', '1', 0,
  /* 19 */ 'R', '2', '1', 0,
  /* 23 */ 'R', '3', '1', 0,
  /* 27 */ 'R', '1', 0,
  /* 30 */ 'R', '1', '2', 0,
  /* 34 */ 'R', '2', '2', 0,
  ...
};
```

字符串压缩表

FooGenRegisterInfo.inc

```
extern const MCRegisterDesc FooRegDesc[] =
{ // Descriptors
  { 3, 0, 0, 0, 0, 0 },
  { 12, 1, 1, 0, 1, 0 },
  { 27, 1, 1, 0, 1, 0 },
  { 38, 1, 1, 0, 1, 0 },
...
```



MCRegisterInfo.h

```
struct MCRegisterDesc {
  uint32_t Name;       // Printable name for the reg (for debugging)
  uint32_t SubRegs;    // Sub-register set, described above
  uint32_t SuperRegs; // Super-register set, described above

  // Offset into MCRI::SubRegIndices of a list of sub-register indices for each
  // sub-register in SubRegs.
  uint32_t SubRegIndices;

  // RegUnits - Points to the list of register units. The low 4 bits holds the
  // Scale, the high bits hold an offset into DiffLists. See MCRegUnitIterator.
  uint32_t RegUnits;

  /// Index into list with lane mask sequences. The sequence contains a lanemask
  /// for every register unit.
  uint16_t RegUnitLaneMasks;
};
```

include/llvm/MC/MCRegisterInfo.h

指令信息

- 指令编码
- 操作数
- 汇编代码
- 指令DAG模式（用于指令选择）
- 各种属性/Flag
- ...

class Instruction

```
class Instruction : InstructionEncoding {
  string Namespace = "";

  dag OutOperandList;       // An dag containing the MI def operand list.
  dag InOperandList;        // An dag containing the MI use operand list.
  string AsmString = "";    // The .s format to print the instruction with.
  // Pattern - Set to the DAG pattern for this instruction, if we know of one,
  // otherwise, uninitialized.
  list<dag> Pattern;
  ...
}
```

指令操作数

汇编代码

DAG模式

include/llvm/Target/Target.td

FooInstrInfo.td

```
class FooInst<dag outs, dag ins, bits<7> opc, string opstr,
              string argstr, list<dag> pattern> : Instruction {
  bits<32> Inst;
  let Namespace = "Foo";
  let Inst{6-0} = opc;
  let OutOperandList = outs;
  let InOperandList = ins;
  let AsmString = opstr # "\t" # argstr;
  let Pattern = pattern;
}
```

固定的编码域

设置指令操作数

设置DAG模式

FooInstrInfo.td

```
class FooInst_RR<dag outs, dag ins, bits<7> opc, string opstr,
                 string argstr, list<dag> pattern>
    : FooInst<outs, ins, opc, opstr, argstr, pattern> {
  bits<5> rd;
  bits<5> rs1;
  bits<5> rs2;
  let Inst{21-17} = rd;
  let Inst{16-12} = rs1;
  let Inst{11-7} = rs2;
}

def ADD : FooInst_RR<(outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
                     0b1000000, "add", "$rd, $rs1, $rs2",
                     [(set i32:$rd, (add i32:$rs1, i32:$rs2))]>;
def SUB : FooInst_RR<(outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
                     0b1000001, "sub", "$rd, $rs1, $rs2",
                     [(set i32:$rd, (sub i32:$rs1, i32:$rs2))]>;
```

设置指令编码

操作数名

定义2条指令

FooGenInstrInfo.inc

- $ llvm-tblgen -gen-instr-info ...

```
namespace Foo {
  enum {
    ...
    ADD = 118,
    SUB = 119,
    INSTRUCTION_LIST_END = 120
  };
```

指令枚举值

FooGenInstrInfo.inc

MC指令描述

```
extern const MCInstrDesc FooInsts[] = {
  ...
  { 118,        3,       1,       0,       0,       0, 0x0ULL, nullptr, nullptr,
OperandInfo30, -1 ,nullptr },  // Inst #118 = ADD
  { 119,        3,       1,       0,       0,       0, 0x0ULL, nullptr, nullptr,
OperandInfo30, -1 ,nullptr },  // Inst #119 = SUB
};
```

注： MCInstrDesc在include/llvm/MC/MCInstrInfo.h中定义

MCInstrDesc

```
class MCInstrDesc {
public:
  unsigned short Opcode;          // The opcode number
  unsigned short NumOperands;     // Num of args (may be more if variable_ops)
  unsigned char NumDefs;          // Num of args that are definitions
  unsigned char Size;             // Number of bytes in encoding.
  unsigned short SchedClass;      // enum identifying instr sched class
  uint64_t Flags;                 // Flags identifying machine instr class
  uint64_t TSFlags;               // Target Specific Flag values
  const MCPhysReg *ImplicitUses; // Registers implicitly read by this instr
  const MCPhysReg *ImplicitDefs; // Registers implicitly defined by this instr
  const MCOperandInfo *OpInfo;   // 'NumOperands' entries about operands
  // Subtarget feature that this is deprecated on, if any
  // -1 implies this is not deprecated by any single feature. It may still be
  // deprecated due to a "complex" reason, below.
  int64_t DeprecatedFeature;
```

include/llvm/MC/MCInstrInfo.h

FooGenMCCodeEmitter.inc

```
uint64_t FooMCCodeEmitter::getBinaryCodeForInstr(const MCInst &MI,
    SmallVectorImpl<MCFixup> &Fixups, const MCSubtargetInfo &STI) const {
  static const uint64_t InstBits[] = {...
    UINT64_C(64),        // ADD
    UINT64_C(65),        // SUB
    UINT64_C(0)
  };
  const unsigned opcode = MI.getOpcode();
  uint64_t Value = InstBits[opcode];
  uint64_t op = 0;
  (void)op;  // suppress warning
  switch (opcode) {
    case Foo::ADD:
    case Foo::SUB: {
      // op: rd
      op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
      Value |= (op & UINT64_C(31)) << 17;
      // op: rs1
      op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);
      Value |= (op & UINT64_C(31)) << 12;
      // op: rs2
      op = getMachineOpValue(MI, MI.getOperand(2), Fixups, STI);
      Value |= (op & UINT64_C(31)) << 7;
      break;
    }
```

指令编码
0b1000001

指令编码 |=
寄存器操作数编码

FooGenDAGISel.inc

```
void DAGISEL_CLASS_COLONCOLON SelectCode(SDNode *N)
{
  // Some target values are emitted as 2 bytes, TARGET_VAL handles this.
  #define TARGET_VAL(X) X & 255, unsigned(X) >> 8
  static const unsigned char MatcherTable[] = {
/*     0*/  OPC_SwitchOpcode /*2 cases */, 10, TARGET_VAL(ISD::ADD),// ->14
/*     4*/    OPC_RecordChild0, // #0 = $rs1
/*     5*/    OPC_RecordChild1, // #1 = $rs2
/*     6*/    OPC_MorphNodeTo1, TARGET_VAL(Foo::ADD), 0,
                   MVT::i32, 2/*#Ops*/, 0, 1,
          // Src: (add:{ *:[i32] } i32:{ *:[i32] }:$rs1, i32:{ *:[i32] }:$rs2) - Complexity = 3
          // Dst: (ADD:{ *:[i32] } i32:{ *:[i32] }:$rs1, i32:{ *:[i32] }:$rs2)
/*    14*/  /*SwitchOpcode*/ 10, TARGET_VAL(ISD::SUB),// ->27
/*    17*/    OPC_RecordChild0, // #0 = $rs1
/*    18*/    OPC_RecordChild1, // #1 = $rs2
/*    19*/    OPC_MorphNodeTo1, TARGET_VAL(Foo::SUB), 0,
                   MVT::i32, 2/*#Ops*/, 0, 1,
          // Src: (sub:{ *:[i32] } i32:{ *:[i32] }:$rs1, i32:{ *:[i32] }:$rs2) - Complexity = 3
          // Dst: (SUB:{ *:[i32] } i32:{ *:[i32] }:$rs1, i32:{ *:[i32] }:$rs2)
/*    27*/  0, // EndSwitchOpcode
    0
  }; // Total Array size is 29 bytes
```

指令选择
函数

指令选择
模式匹配表

目 录

参考资料

- TableGen语言

  http://releases.llvm.org/8.0.1/docs/TableGen

  http://releases.llvm.org/8.0.1/docs/TableGen/LangIntro.html

  http://releases.llvm.org/8.0.1/docs/TableGen/LangRef.html

- llvm-tblgen工具

  http://releases.llvm.org/8.0.1/docs/CommandGuide/tblgen.html

- TableGen例子

  https://github.com/isrc-cas/tablegen-example

TableGen例子

■ 欢迎试用

TableGen Example

TableGen simple examples written by xmj (mingjie@iscas.ac.cn).

- Records: shows the basic concepts of TableGen language.
- Foo: shows a simple target description.

Prerequisites

The `llvm-tblgen` tool is needed to generate outputs, which is available under the llvm build directory.

The llvm source files is also needed by the Foo target description file.

Generate outputs

Modify the Makefile, change the values of `LLVM_INCLUDE` and `LLVM_TBLGEN` to the correct ones in your host environment. Then run:

```
make
```

Or specify the values in the command:

```
make LLVM_TBLGEN=/path/to/the/llvm_tblgen LLVM_INCLUDE=/path/to/the/llvm/include
```

Cleanup outputs

```
make clean
```

谢 谢

欢迎交流合作