TableGen

Introduction

TableGen's purpose is to help a human develop and maintain records of domain-specific information. Because there may be a large number of these records, it is specifically designed to allow writing flexible descriptions and for common features of these records to be factored out. This reduces the amount of duplication in the description, reduces the chance of error, and makes it easier to structure domain specific information.

The core part of TableGen parses a file, instantiates the declarations, and hands the result off to a domain-specific backend for processing.

The current major users of TableGen are The LLVM Target-Independent Code Generator and the Clang diagnostics and attributes.

Note that if you work on TableGen much, and use emacs or vim, that you can find an emacs "TableGen mode" and a vim language file in the llvm/utils/emacs and llvm/utils/vim directories of your LLVM distribution, respectively.

The TableGen program

TableGen files are interpreted by the TableGen program: *llvm-tblgen* available on your build directory under *bin*. It is not installed in the system (or where your sysroot is set to), since it has no use beyond LLVM's build process.

Running TableGen

TableGen runs just like any other LLVM tool. The first (optional) argument specifies the file to read. If a filename is not specified, llvm-tblgen reads from standard input.

To be useful, one of the backends must be used. These backends are selectable on the command line (type 'llvm-tblgen -help' for a list). For example, to get a list of all of the definitions that subclass a particular type (which can be useful for building up an enum list of these records), use the -print-enums option:

```
$ llvm-tblgen X86.td -print-enums -class=Register
AH, AL, AX, BH, BL, BP, BPL, BX, CH, CL, CX, DH, DI, DIL, DL, DX, EAX, EBP, EBX,
ECX, EDI, EDX, EFLAGS, EIP, ESI, ESP, FP0, FP1, FP2, FP3, FP4, FP5, FP6, IP,
MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7, R10, R10B, R10D, R10W, R11, R11B, R11D,
R11W, R12, R12B, R12D, R12W, R13, R13B, R13D, R13W, R14, R14B, R14D, R14W, R15,
R15B, R15D, R15W, R8, R8B, R8D, R8W, R9, R9B, R9D, R9W, RAX, RBP, RBX, RCX, RDI,
RDX, RIP, RSI, RSP, SI, SIL, SP, SPL, ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7,
XMM0, XMM1, XMM10, XMM11, XMM12, XMM13, XMM14, XMM15, XMM2, XMM3, XMM4, XMM5,
XMM6, XMM7, XMM8, XMM9,

$ llvm-tblgen X86.td -print-enums -class=Instruction
ABS_F, ABS_Fp32, ABS_Fp64, ABS_Fp80, ADC32mi, ADC32mi8, ADC32mr, ADC32ri,
ADC32ri8, ADC32rm, ADC32rr, ADC64mi32, ADC64mi8, ADC64mr, ADC64ri32, ADC64ri8,
ADC64rm, ADC64rr, ADD16mi, ADD16mi8, ADD16mr, ADD16ri, ADD16ri8, ADD16rm,
ADD16rr, ADD32mi, ADD32mi8, ADD32mr, ADD32ri, ADD32ri8, ADD32rm, ADD32rr,
ADD64mi32, ADD64mi8, ADD64mr, ADD64ri32, ...
```

The default backend prints out all of the records.

If you plan to use TableGen, you will most likely have to write a backend that extracts the information specific to what you need and formats it in the appropriate way.

Example

With no other arguments, *llvm-tblgen* parses the specified file and prints out all of the classes, then all of the definitions. This is a good way to see what the various definitions expand to fully. Running this on the X86.td file prints this (at the time of this writing):

```
...
def ADD32rr {   // Instruction X86Inst I
  string Namespace = "X86";
  dag OutOperandList = (outs GR32:$dst);
  dag InOperandList = (ins GR32:$src1, GR32:$src2);
  string AsmString = "add{l}\t{$src2, $dst|$dst, $src2}";
  list<dag> Pattern = [(set GR32:$dst, (add GR32:$src1, GR32:$src2))];
  list<Register> Uses = [];
  list<Register> Defs = [EFLAGS];
  list<Predicate> Predicates = [];
  int CodeSize = 3;
  int AddedComplexity = 0;
  bit isReturn = 0;
  bit isBranch = 0;
  bit isIndirectBranch = 0;
  bit isBarrier = 0;
  bit isCall = 0;
  bit canFoldAsLoad = 0;
  bit mayLoad = 0;
  bit mayStore = 0;
  bit isImplicitDef = 0;
  bit isConvertibleToThreeAddress = 1;
  bit isCommutable = 1;
  bit isTerminator = 0;
  bit isReMaterializable = 0;
  bit isPredicable = 0;
  bit hasDelaySlot = 0;
  bit usesCustomInserter = 0;
  bit hasCtrlDep = 0;
  bit isNotDuplicable = 0;
  bit hasSideEffects = 0;
  InstrItinClass Itinerary = NoItinerary;
  string Constraints = "";
  string DisableEncoding = "";
  bits<8> Opcode = { 0, 0, 0, 0, 0, 0, 0, 1 };
  Format Form = MRMDestReg;
  bits<6> FormBits = { 0, 0, 0, 0, 1, 1 };
  ImmType ImmT = NoImm;
  bits<3> ImmTypeBits = { 0, 0, 0 };
  bit hasOpSizePrefix = 0;
  bit hasAdSizePrefix = 0;
  bits<4> Prefix = { 0, 0, 0, 0 };
  bit hasREX_WPrefix = 0;
  FPFormat FPForm = ?;
  bits<3> FPFormBits = { 0, 0, 0 };
}
...
```

This definition corresponds to the 32-bit register-register add instruction of the x86 architecture. def ADD32rr defines a record named ADD32rr, and the comment at the end of the line indicates the superclasses of the definition. The body of the record contains all of the data that TableGen assembled for the record, indicating that the instruction is part of the “X86” namespace, the pattern indicating how the instruction is selected by the code generator, that it is a two-address instruction, has a particular encoding, etc. The contents and semantics of the information in the record are specific to the needs of the X86 backend, and are only shown as an example.

As you can see, a lot of information is needed for every instruction supported by the code generator, and specifying it all manually would be unmaintainable, prone to bugs, and tiring to do in the first place. Because we are using TableGen, all of the information was derived from the following definition:

```
let Defs = [EFLAGS],
    isCommutable = 1,                // X = ADD Y,Z --> X = ADD Z,Y
    isConvertibleToThreeAddress = 1 in // Can transform into LEA.
def ADD32rr  : I<0x01, MRMDestReg, (outs GR32:$dst),
                                   (ins GR32:$src1, GR32:$src2),
                "add{l}\t{$src2, $dst|$dst, $src2}",
                [(set GR32:$dst, (add GR32:$src1, GR32:$src2))]>;
```

This definition makes use of the custom class I (extended from the custom class X86Inst), which is defined in the X86-specific TableGen file, to factor out the common features that instructions of its class share. A key feature of TableGen is that it allows the end-user to define the abstractions they prefer to use when describing their information.

Each def record has a special entry called “NAME”. This is the name of the record (“ADD32rr” above). In the general case def names can be formed from various kinds of string processing expressions and NAME resolves to the final value obtained after resolving all of those expressions. The user may refer to NAME anywhere she desires to use the ultimate name of the def. NAME should not be defined anywhere else in user code to avoid conflicts.

Syntax

TableGen has a syntax that is loosely based on C++ templates, with built-in types and specification. In addition, TableGen’s syntax introduces some automation concepts like multiclass, foreach, let, etc.

Basic concepts

TableGen files consist of two key parts: ‘classes’ and ‘definitions’, both of which are considered ‘records’.

TableGen records have a unique name, a list of values, and a list of superclasses. The list of values is the main data that TableGen builds for each record; it is this that holds the domain specific information for the application. The interpretation of this data is left to a specific backend, but the structure and format rules are taken care of and are fixed by TableGen.

TableGen definitions are the concrete form of 'records'. These generally do not have any undefined values, and are marked with the 'def' keyword.

```
def FeatureFPARMv8 : SubtargetFeature<"fp-armv8", "HasFPARMv8", "true",
                                      "Enable ARMv8 FP">;
```

In this example, FeatureFPARMv8 is SubtargetFeature record initialised with some values. The names of the classes are defined via the keyword *class* either on the same file or some other included. Most target TableGen files include the generic ones in include/llvm/Target.

TableGen classes are abstract records that are used to build and describe other records. These classes allow the end-user to build abstractions for either the domain they are targeting (such as "Register", "RegisterClass", and "Instruction" in the LLVM code generator) or for the implementor to help factor out common properties of records (such as "FPInst", which is used to represent floating point instructions in the X86 backend). TableGen keeps track of all of the classes that are used to build up a definition, so the backend can find all definitions of a particular class, such as "Instruction".

```
class ProcNoItin<string Name, list<SubtargetFeature> Features>
    : Processor<Name, NoItineraries, Features>;
```

Here, the class ProcNoItin, receiving parameters *Name* of type *string* and a list of target features is specializing the class Processor by passing the arguments down as well as hard-coding NoItineraries.

TableGen multiclasses are groups of abstract records that are instantiated all at once. Each instantiation can result in multiple TableGen definitions. If a multiclass inherits from another multiclass, the definitions in the sub-multiclass become part of the current multiclass, as if they were declared in the current multiclass.

```
multiclass ro_signed_pats<string T, string Rm, dag Base, dag Offset, dag Extend,
                          dag address, ValueType sty> {
def : Pat<(i32 (!cast<SDNode>("sextload" # sty) address)),
          (!cast<Instruction>("LDRS" # T # "w_" # Rm # "_RegOffset")
            Base, Offset, Extend)>;

def : Pat<(i64 (!cast<SDNode>("sextload" # sty) address)),
          (!cast<Instruction>("LDRS" # T # "x_" # Rm # "_RegOffset")
            Base, Offset, Extend)>;
}
```

```
defm : ro_signed_pats<"B", Rm, Base, Offset, Extend,
                      !foreach(decls.pattern, address,
                               !subst(SHIFT, imm_eq0, decls.pattern)),
                      i8>;
```

See the TableGen Language Introduction for more generic information on the usage of the language, and the TableGen Language Reference for more in-depth description of the formal language specification.

TableGen backends

TableGen files have no real meaning without a back-end. The default operation of running llvm-tblgen is to print the information in a textual format, but that's only useful for debugging of the TableGen files themselves. The power in TableGen is, however, to interpret the source files into an internal representation that can be generated into anything you want.

Current usage of TableGen is to create huge include files with tables that you can either include directly (if the output is in the language you're coding), or be used in pre-processing via macros surrounding the include of the file.

Direct output can be used if the back-end already prints a table in C format or if the output is just a list of strings (for error and warning messages). Pre-processed output should be used if the same information needs to be used in different contexts (like Instruction names), so your back-end should print a meta-information list that can be shaped into different compile-time formats.

See the TableGen BackEnds for more information.

TableGen Deficiencies

Despite being very generic, TableGen has some deficiencies that have been pointed out numerous times. The common theme is that, while TableGen allows you to build Domain-Specific-Languages, the final languages that you create lack the power of other DSLs, which in turn increase considerably the size and complexity of TableGen files.

At the same time, TableGen allows you to create virtually any meaning of the basic concepts via custom-made back-ends, which can pervert the original design and make it very hard for newcomers to understand the evil TableGen file.

There are some in favour of extending the semantics even more, but making sure back-ends adhere to strict rules. Others are suggesting we should move to less, more powerful DSLs designed with specific purposes, or even re-using existing DSLs.

Either way, this is a discussion that will likely span across several years, if not decades. You can read more in the TableGen Deficiencies document.