Tablegen Deep Dive

Reed Kotler
MIPS Technologies, Inc
12 April 2012

Overview

- Introduction to TableGen
- Providing additional information to available documentation
- Help improve TableGen definition
  - Grammars and language definition
  - Develop more documentation and examples
- Ability to further define compiler through TableGen

Why TableGen?

- **Describes target machine to LLVM**
  - Architecture information typically duplicated and spread throughout machine's ISA/ABI documents
- **Succinctly represents target description**
  - Description used by compiler, assemblers and disassemblers, etc.
- **Simplifies maintainability and reliability due to modular design**
- **TableGen documents (located at llvm.org)**
  - "TableGen Fundamentals"
  - "The LLVM Target-Independent Code Generator"
  - "Writing an LLVM Compiler Backend"

Overview of TableGen Processing

- **Two pass processing of description files**
  - First pass is essentially a template/macro preprocessor . Its output is an expanded set of class and record (data) definitions.
  - There are many second passes (application specific backends). These produce various .inc files used by the target dependent backends
- **Each backend processor assigns its own meaning to expanded class and record data, as for example an application that reads XML does**

Overview of TableGen Processing (continued)

- **First phase is text macro processor**
  - important exception: The information collected during record creation is available to the application specific backend using it .
- **For each expanded record, the list of classes which it was derived from is preserved for use by the application specific backends.**
  - This is an important detail and something not usually present in a macro or template preprocessor.
  - Example: all records derived from a class can be collected by tool and data can be distributed throughout definitions and later, this can be collected back together. i.e. the set of register or instruction defns.
- **All known backends for tablegen are linked into llvm-tblgen. Full set can be see by llvm-tblgen --help.**

Where to Find Additional Information

- **Test_data for TableGen tool**
  - test/tablegen/*.td
  - Any target-specific compiler .td file
- **Information available on code.google.com**
  - http://code.google.com/p/alt-llvm-tablegen/
  - This presentation
  - Other sample code
- **Google code project is also scratch pad for future ideas**
  - Experimentation
  - Code samples and clarified documentation
- **Goal is to improve llvm.org documentation with minimal disruption**

Learning More About TableGen

- **Run only the first phase of llvm-tblgen. Some options:**
  - -print-records(default)
  - -print-enums
  - -class=<classname>
  - -print-sets
- **read through the .td files in the llvm source tree**
  - Target template .td files. Lib/include/llvm/target/*.td
  - **Target-specific .td for familiar machines** Lib/target/*/*.td
  - **Study the sample code of the talk**
  - All .td files from the talk are in a tar file at the conference website
  - can be found in the alt-llvm-tablegen project at googlecode
- **Contribute to the project alt-llvm-tablegen**

Example: talk4.td

```
// pure data. anonymous type.
def rec1 {
   int i = 1;
   string h = "hello";
}
// data record of type class1
class class1 {
   int i = 1;
   string h = "hello";
}
def rec2 : class1;
// simple parameterized with default parameters
class class2 <int p1, string p2 = "hello"> {
   int i = p1;
   string h = p2;
   }
def rec3 : class2 <5>
```

Example: talk4a.out (classes)

```
------------- Classes -----------------
class class1 {
   int i = 1;
   string h = "hello";
   string NAME = ?;
}
class class2<int class2:p1 = ?, string class2:p2 = "hello"> {
   int i = class2:p1;
   string h = class2:p2;
   string NAME = ?;
}
```

Example: talk4a.out (defs)

```
------------- Defs -----------------
def rec1 {
   int i = 1;
   string h = "hello";
   string NAME = ?;
}
def rec2 { // class1
   int i = 1;
   string h = "hello";
   string NAME = ?;
}
def rec3 { // class2
   string class2:p2 = "hello";
   int i = 5;
   string h = "hello";
   string NAME = ?;
}
```

Example: talk5.td

```
class forward; // could be empty class or forward declaration

class t <list<forward> parm1> {
   list<forward> f = parm1;
}

class forward<string parm1 = ""> {
   string n = parm1;
}
def one: forward<"cat">;
def two: forward<"dog">;
```

Example: talk5.td (cont)

```
class empty;
def e1 : empty;
def e2 : empty;

// or is it?
class empty {
   int i;
}
def e3: empty;
```

Example: talk6.td

```
class node;
def na : node;
def nb : node;
def nc : node;
def prim_types {
  bit b;
  bit t = 1;
  bit f = 0;
  int i;
  int j = 0b1100; // binary representation
  int k = 0377; // octal representation
```

Example: talk6.td

```
int l = 0xface;
bits<3> i3 = 0b110;
bits<4> j4 = {1, 0, 1, 1};
string s1;
string s2 = "hello";
list<int> li = [1, 2, 3, 4];
list<string> strings = ["house", "road", "basement", "shed"];
list<node> ln = [na, nb, nc];
dag d1 = (na 1, 2 );
dag d2 = (na (nb 1, 2), nc, 4);
```

Example: talk6.td

```
 code c;
  // similar to python """
  code c_hello_world = [{
#include <stdio.h>
int main() {
  printf("hello world \n");
}
  }] ;

string s_code = [{
#include <stdio.h>
int main() {
  printf("hello world \n");
}
  }] ;
}
```

Example: talk7.td

```
// examples of let
class c1 {
  int i = 1;
}

def rec1 : c1 {
  let i = 2;
}

class c {
  int i = 1;
  int j = 2;
  int k = 3;
}
```

Example: talk7.td

```
def rec1: c;
let i = 2 in
  def rec2: c;

let i = 3, j=6, k=9 in {
  def rec3: c;
  def rec4: c;
}

let i=100 in {
  def rec5: c;
  let j=200 in {
    def rec6: c;
    let k=300 in
      def rec7: c;
  }
}
```

Example: talk8.td

```
// bits. very useful for encoding.
def basic_bits {
  bits<5> x = 0b11001;
  bits<5> y = {1, 0, 1, 0, 0};
  bits<3> z = y{0-2};
  bits<10> zz;
  let zz{0-4} = x;
  let zz{5-9} = y;
  bits<10> zzz;
  let zzz{4-0} = x;
  let zzz{9-5} = y;
  bit b = 1;
  bits<5> zzzz;
  let zzzz{0} = b;
  let zzzz{4-1} = 5;
  bit b1 = zzzz{0};
}
```

Example: talk9.td

```
// Example of multiple inheritance
class c1 {
  int i1 = 1;
}

class c2 {
  int i2 = 2;
}

class c3: c1, c2;

def d1: c1, c2;
def d2: c3;
```

Example: talk10.td

```
// struct c1;
// struct c1 {
// struct c1 *list;
// };

// struct c2 {
// struct c2 *list;
// };

// used for subregisters
class c1;
class c1 <list<c1> param = []> {
  list<c1> l = param;
}

class c2<list<c2> param = []> {
  list<c2> l = param;
}
```

Example: talk11.td

```
//
//
class Instruction<string n> {
  string variant = n;
}

multiclass basic_r {
  def rr: Instruction< "rr">;
  def rm: Instruction<"rm">;
}

defm ADD : basic_r;
defm SUB : basic_r;
```

Example: talk11.td (output)

```
------------- Classes -----------------
class Instruction<string Instruction:n = ?> {
  string variant = Instruction:n;
  string NAME = ?;
}
------------- Defs -----------------
def ADDrm { // Instruction rm
  string variant = "rm";
  string NAME = ?;
}
def ADDrr { // Instruction rr
  string variant = "rr";
  string NAME = ?;
}
```

Example: talk11.td (output)

```
def SUBrm { // Instruction rm
  string variant = "rm";
  string NAME = ?;
}
def SUBrr { // Instruction rr
  string variant = "rr";
  string NAME = ?;
}
```

Example: talk12.td

```
class Instruction<string n> {
  string variant = n;
}
multiclass basic_r {
  def rr: Instruction< "rr">;
  def rm: Instruction<"rm">;
}
multiclass basic_s {
  defm SS : basic_r;
  defm SD : basic_r;
}
defm ADD2 : basic_s;
```

Example: talk12.td (output)

```
------------- Classes -----------------
class Instruction<string Instruction:n = ?> {
  string variant = Instruction:n;
  string NAME = ?;
}
------------- Defs -----------------
def ADD2SDrm {  // Instruction rm SDrm
  string variant = "rm";
  string NAME = ?;
}
```

Example: talk12.td (output)

```
def ADD2SDrr { // Instruction rr SDrr
  string variant = "rr";
  string NAME = ?;
}
def ADD2SSrm { // Instruction rm SSrm
  string variant = "rm";
  string NAME = ?;
}
def ADD2SSrr { // Instruction rr SSrr
  string variant = "rr";
  string NAME = ?;
}
```

Understanding Register Definitions

- *** Examples of Register Classes**
  - Intel 64 bit integer, 32 bit integer, 16 bit integer, 8 bit integer
  - MIPS 64 bit integer, 32 bit integer, 32 bit float, 64 bit float
- **In some cases, registers are aliased.**
  - Intel AX register consists of AH and AL
  - MIPS 64 bit float contains a 32 bit sp even and 32 bit sp odd register
- **Want to be able to refer to the various register pieces. - Introduce notion of "subreg index".**
  - Subreg index is way to refer to the pieces.
    - Intel 16 bit registers contain two 8 bit integer registers, designated as Lo and Hi, this Ax containt AH and AL.
    - MIPS D0 contains registers F0, F1

What About sub-registers?

Given an x86 16-bit register, one identifies that AH and AL are subregisters and will be in a list of subregisters .But how do you find the 8 bit low and 8 bit hi?

Well there is a Parallel list of subregister indices to the list of subregisters.

for example, in the AX register definition there will be.

```
    [AL, AH]
    [8bit_lo, 8bit]
```

This could also be written

```
    [AH, AL]
    [8 bit, 8bit_lo]
```

What About sub-registers? (cont.)

```
for MIPS D0, the definition for subregisters would contain:
     [F0, F1]
     [FP_EVEN, FP_ODD]
This could also be written
     [F1, F0]
     [FP_ODD, FP_EVEN]

So then to find the 8bit_lo subregister of AX, you search
for AL in the subreg iindex list and take that index number
and then find the parallel member of
the subregister list.
```

Example: from talk15.td

```
// From Target.td
// SubRegIndex - Use instances of SubRegIndex to identify
   subregisters.
class SubRegIndex<list<SubRegIndex> comps = []> {
  string Namespace = "";
  // ComposedOf - A list of two SubRegIndex instances, [A, B].
  // This indicates that this SubRegIndex is the result of
   composing A and B.
  list<SubRegIndex> ComposedOf = comps;
}
// abbreviated RegisterClass definition
//
class Register<string n> {
  string AsmName = n;
  list<Register> SubRegs = [];
  list<SubRegIndex> SubRegIndices = [];
}
```

Example: from talk15.td

```
class RegisterWithSubRegs<string n, list<Register> subregs> :
   Register<n> {
  let SubRegs = subregs;
}
// from X86RegisterInfo.td
//
def sub_8bit : SubRegIndex; // no subregisters
def sub_8bit_hi: SubRegIndex;
def AL : Register<"al">;
def AH : Register<"ah">;

   let SubRegIndices = [sub_8bit, sub_8bit_hi] in {
   def AX : RegisterWithSubRegs<"ax", [AL,AH]>;
   }
```

Thank You!

At the core of the user experience®