LESSONS IN TABLEGEN

NICOLAI HÄHNLE

Agenda

- What is TableGen?
  - TableGen, the tool and the language
  - Uses in LLVM
- TableGen language features
  - Type system
  - Classes
  - Let-statements and late evaluation
  - Multiclasses, foreach, and defset
  - Built-ins and “functional programming”
- Example: AMDGPU image intrinsics and instructions
  - Generic searchable tables backend

What is TableGen?

TableGen, the tool

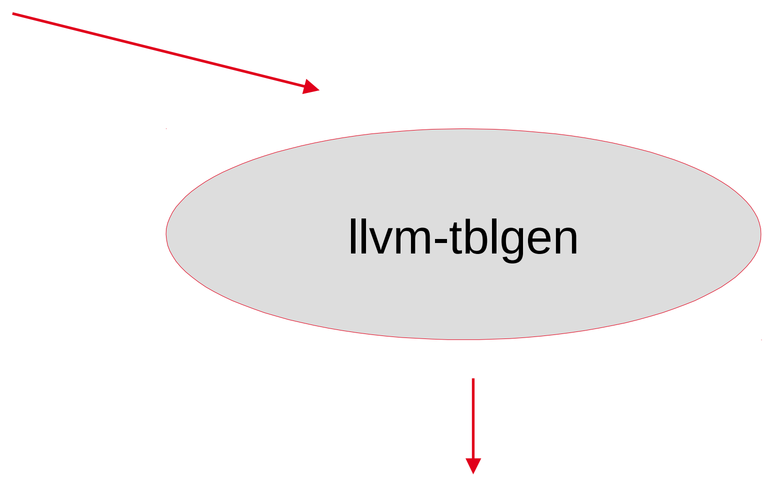
```
let Defs = [SCC] in {
let isCommutable = 1 in {
  def S_AND_B32 : SOP2_32 <"s_and_b32",
    [(set i32:$sdst, (UniformBinFrag<and> i32:$src0, i32:$src1))]
  >;

  def S_AND_B64 : SOP2_64 <"s_and_b64",
    [(set i64:$sdst, (UniformBinFrag<and> i64:$src0, i64:$src1))]
  >;

  def S_OR_B32 : SOP2_32 <"s_or_b32",
    [(set i32:$sdst, (UniformBinFrag<or> i32:$src0, i32:$src1))]
  >;

  def S_OR_B64 : SOP2_64 <"s_or_b64",
    [(set i64:$sdst, (UniformBinFrag<or> i64:$src0, i64:$src1))]
  >;

  // …
}
}
```

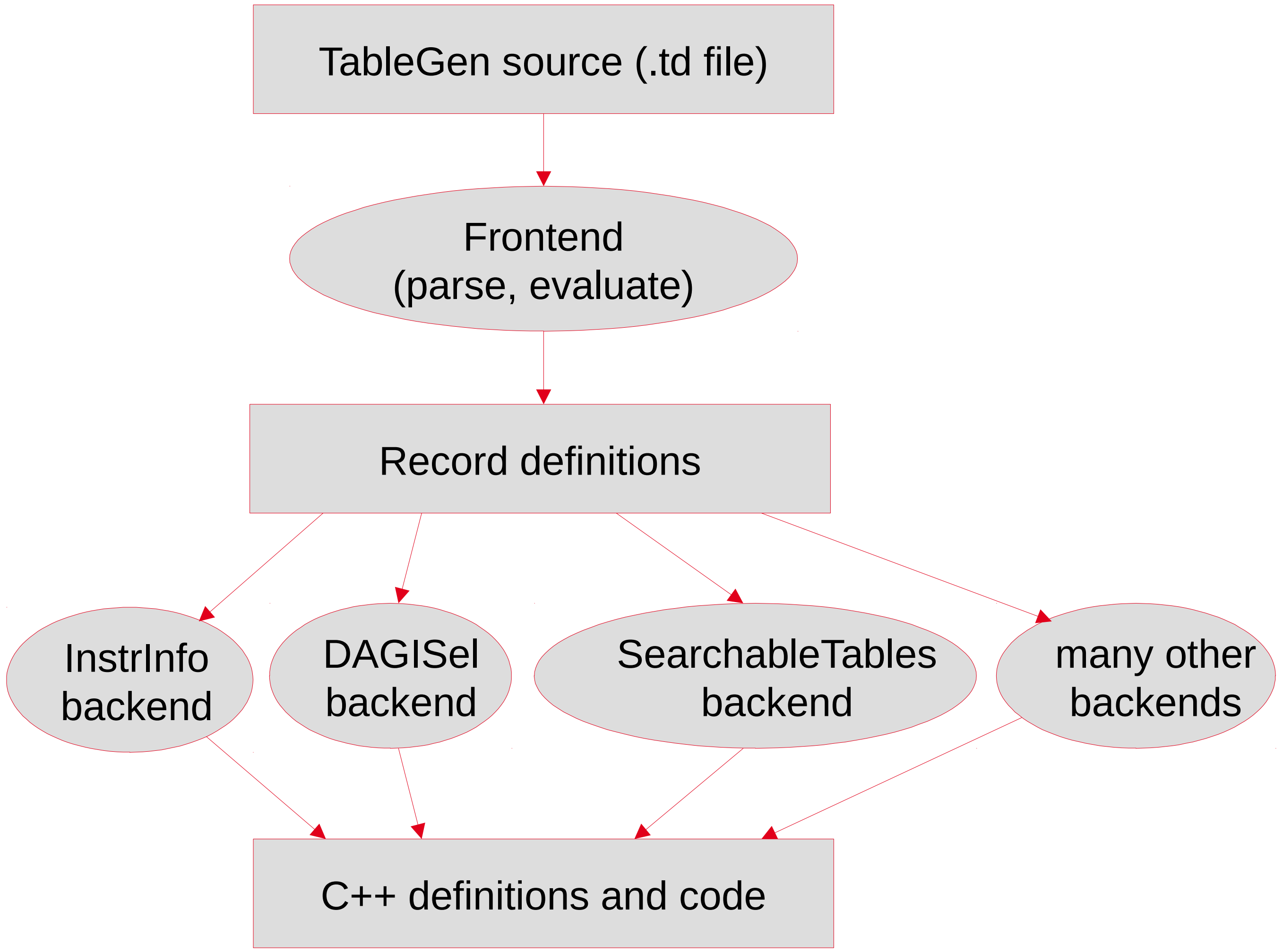


- Generated files in ${builddir}/lib/Target/${target}/
  - MCInstrDesc
  - Instruction selection
  - Assembly parser
  - Disassembler
  - …

Architecture

- llvm-tblgen and clang-tblgen tools
  - Same frontend (library in lib/TableGen)
  - Different backends (utils/TableGen)
- Specify the desired backend on the command-line
  - Default: dump all record definitions
- CMake integration
  - LLVM_OPTIMIZED_TABLEGEN=ON in debug builds!

  -
    ```
    set(LLVM_TARGET_DEFINITIONS AMDGPU.td)
    tablegen(LLVM AMDGPUGenAsmMatcher.inc
                             -gen-asm-matcher)
    ```



Records

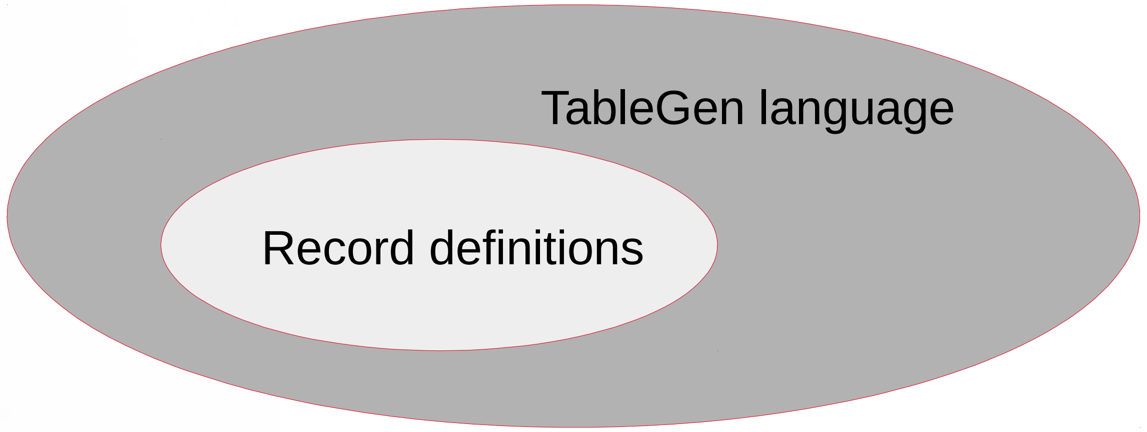
- Key-value dictionaries
- Meaning defined by backends
- Records can be derived from classes
  - Often used for filtering
- Records can be named
  - Sometimes required (e.g. intrinsics, machine instructions)
  - Sometimes optional (e.g. ISel patterns)
  - Helpful for debugging

```
let Defs = [SCC] in {
let isCommutable = 1 in {
  def S_AND_B32 : SOP2_32 <"s_and_b32",
    [(set i32:$sdst, (UniformBinFrag<and> i32:$src0, i32:$src1))]
  >;
```

```
def S_AND_B32 { // Instruction AMDGPUInst PredicateControl
                // GCNPredicateControl InstSI SIMCInstr SOP_Pseudo
                // SOP2_Pseudo SOP2_32
  dag OutOperandList = (outs SReg_32:$sdst);
  dag InOperandList = (ins SSrc_b32:$src0, SSrc_b32:$src1);
  string Mnemonic = "s_and_b32";
  string AsmOperands = "$sdst, $src0, $src1";
  list<dag> Pattern = [(set i32:$sdst, (anonymous_1822 i32:$src0, i32:$src1))];
  list<Register> Uses = [];
  list<Register> Defs = [SCC];
  bit isReturn = 0;
  bit isBranch = 0;
  bit isIndirectBranch = 0;
  bit mayLoad = 0;
  bit mayStore = 0;
  bit isConvertibleToThreeAddress = 0;
  bit isCommutable = 1;
  bit isTerminator = 0;
  bit isReMaterializable = 0;
  bit isPredicable = 0;
  InstrItinClass Itinerary = NullALU;
  list<SchedReadWrite> SchedRW = [WriteSALU];
  Predicate AssemblerPredicate = TruePredicate;
  ...
}
```

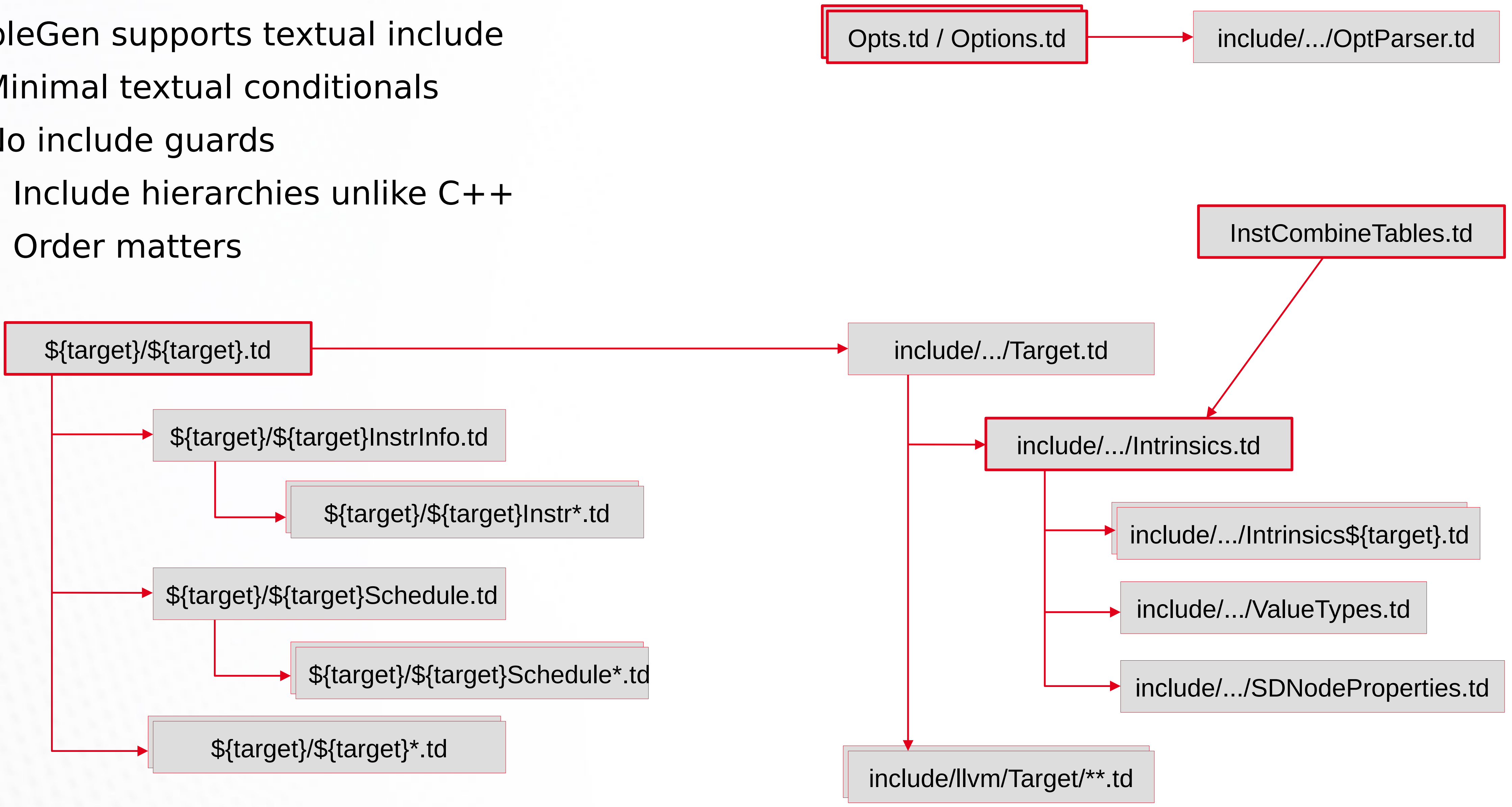
TableGen, the language

- Record definitions
  - Consumed by backends
  - Printed by default backend
  - No classes, built-in ops
- TableGen language
  - Written by developers
  - Superset of record definitions
  - Tools for generating many records with regularities



.td sources in LLVM

- TableGen supports textual include
  - Minimal textual conditionals
  - No include guards
    - Include hierarchies unlike C++
    - Order matters


Opts.td / Options.td
include/.../OptParser.td
InstCombineTables.td
${target}/${target}.td
include/.../Target.td
${target}/${target}InstrInfo.td
include/.../Intrinsics.td
${target}/${target}Instr*.td
include/.../Intrinsics${target}.td
${target}/${target}Schedule.td
include/.../ValueTypes.td
${target}/${target}Schedule*.td
include/.../SDNodeProperties.td
${target}/${target}*.td
include/llvm/Target/**.td

TableGen language features

Agenda

- What is TableGen?
  - TableGen, the tool and the language
  - Uses in LLVM
- TableGen language features
  - Type system
  - Classes
  - Let-statements and late evaluation
  - Multiclasses, foreach, and defset
  - Built-ins and “functional programming”
- Example: AMDGPU image intrinsics and instructions
  - Generic searchable tables backend

Type system at a glance

- bit, bits<N>, int
  - Implicit conversions from and to *int* (with range checks)
  - Bit sequence: *{ 1, 0, 1, 0, 1, 0 }*
  - Slicing a bit sequence: *foo{4}*, *foo{6-8}* – literal constant indices only!
- string, code
  - *“this is a string”* vs. *[{ this_is_code(); }]*
  - Often implicit conversion
- list<T>
  - *[“this”, “is”, “a”, “list<string>”]*
  - *[“explicitly”, “typed”, “list”, “literal”]<string>*
  - Indexing a list: *foo[4]* – literal constant indices only!
- unset (not actually a proper type)
- dag
- class/record types

Unset values and instruction encodings

- ? is-a T for all types T
- Values should usually be defined somehow
- Exception: instruction encodings

```
def Foo {
  bit Unset;
  bit AlsoUnset = ?;
}
```

```
class Enc32 {
  field bits<32> Inst;
  int Size = 4;
}

class VINTRPe <bits<2> op> : Enc32 {
  bits<8> vdst;
  bits<8> vsrc;
  bits<2> attrchan;
  bits<6> attr;

  let Inst{7-0} = vsrc;
  let Inst{9-8} = attrchan;
  let Inst{15-10} = attr;
  let Inst{17-16} = op;
  let Inst{25-18} = vdst;
  let Inst{31-26} = 0x32; // encoding
}
```

```
def V_INTERP_P1_F32_si { // Instruction AMDGPUInst PredicateControl
                         // GCNPredicateControl InstSI VINTRPCommon
                         // Enc32 VINTRPe SIMCInstr VINTRP_Real_si
  field bits<32> Inst = {
    1, 1, 0, 0, 1, 0, vdst{7}, vdst{6},
    vdst{5}, vdst{4}, vdst{3}, vdst{2}, vdst{1}, vdst{0}, 0, 0,
    attr{5}, attr{4}, attr{3}, attr{2}, attr{1}, attr{0}, attrchan{1}, attrchan{0},
    vsrc{7}, vsrc{6}, vsrc{5}, vsrc{4}, vsrc{3}, vsrc{2}, vsrc{1}, vsrc{0} };
  dag OutOperandList = (outs VINTRPDst:$vdst);
  dag InOperandList = (ins VGPR_32:$vsrc, Attr:$attr, AttrChan:$attrchan);
  int Size = 4;
  bits<8> vdst = { ?, ?, ?, ?, ?, ?, ?, ? };
  bits<8> vsrc = { ?, ?, ?, ?, ?, ?, ?, ? };
  bits<2> attrchan = { ?, ? };
  bits<6> attr = { ?, ?, ?, ?, ?, ? };

  ...
}
```

dag type: S-expressions (kind of)

- *(op arg0:$name0, arg1:$name1, ...)*
  - No type restrictions
  - Names are just identifiers (internally of *string* type)
  - arg / name are optional, e.g. *(op $dst, 0, $src)* is equivalent to *(op ?:$dst, 0, ?:$src)*
- Most prominent use is for ISel patterns
  - *op* typically corresponds to SDNode or machine instruction

```
def : GCNPat <
  (f32 (f16_to_fp (xor_oneuse i32:$src0, 0x8000))),
  (V_CVT_F32_F16_e64 SRCMODS.NEG, $src0, DSTCLAMP.NONE, DSTOMOD.NONE)
>;
```

Classes and record types

- Classes are *templates* for records
  - Same syntax, inheritance written like C++
  - Separate namespace
- Type of records derived from a class
  - Also: type of records derived from a set of classes
    - Printed as *{}*, *{A, B}*, …
    - Not expressible in the source language
- Implicit template argument *NAME*
  - Equal to the final name of instantiated record
- Default template arguments are supported

```
class A {
  string Name = NAME;
}

class B<int x> {
  int X = x;
}

def MyRecord : A, B<5> {
  int Y = 3;
}

def Other {
  A a = MyRecord;
  B b = B<3>;
}
```

```
def MyRecord {  // A B
  string Name = "MyRecord";
  int X = 5;
  int Y = 3;
}
def Other {
  A a = MyRecord;
  B b = anonymous_0;
}
def anonymous_0 {  // B
  int X = 3;
}
```

Let-statements and late evaluation

- Override values in records
  - Globally or in class/record bodies
    - Also multiclass “bodies”
  - “Innermost” let wins
- Very powerful due to late evaluation
  - Expressions are evaluated as late as possible
- Consider using let instead of template arguments!

```
class A<int p> {
  int x = p;
  int y = x;
}

let x = 12 in {
  def A1 : A<1>;
  def A2 : A<2> {
    let x = 17;
  }
}

def A3 : A<3> {
  let x = 10;
  let y = 11;
}
```

```
def A1 {        // A
  int x = 12;
  int y = 12;
}
def A2 {        // A
  int x = 17;
  int y = 17;
}
def A3 {        // A
  int x = 10;
  int y = 11;
}
```

- This is not the let you know from functional programming style
  - Cannot define new variables, only override existing ones

```
def Foo {
  let x = 5 in
  int y = x;
}
```

ERROR!

Multiclasses

- Multiclasses are templates for a *set* of records
  - Template arguments like classes
  - Instantiated via *defm*
- Record names are concatenated by default
  - Unless implicit NAME template argument is used

```
class AMDGPUReadPreloadRegisterIntrinsicNamed<string name>
  : Intrinsic<[llvm_i32_ty], [], [IntrNoMem, IntrSpeculatable]>, GCCBuiltin<name>;

multiclass AMDGPUReadPreloadRegisterIntrinsic_xyz_named<string prefix> {
  def _x : AMDGPUReadPreloadRegisterIntrinsicNamed<!strconcat(prefix, "_x")>;
  def _y : AMDGPUReadPreloadRegisterIntrinsicNamed<!strconcat(prefix, "_y")>;
  def _z : AMDGPUReadPreloadRegisterIntrinsicNamed<!strconcat(prefix, "_z")>;
}

defm int_amdgcn_workgroup_id : AMDGPUReadPreloadRegisterIntrinsic_xyz_named
                               <"__builtin_amdgcn_workgroup_id">;
```

```
multiclass MC {
  def Rec1;
  def NAME#Rec2;
  def Rec3#NAME;
}
defm Base : MC;
```

```
def BaseRec1 {
}
def BaseRec2 {
}
def Rec3Base {
}
```

Multiclasses and defm – corner cases

- Multiclasses can inherit from other multiclasses
  - Equivalent to nesting with *defm*

```
multiclass MC : Base {
  ...
}
```

```
multiclass MC {
  defm : Base;
  ...
}
```

    - Almost: behavior with let-statements is inconsistent
    - Is this a bug? Maybe remove inheritance entirely?
- *defm* can “inherit” from classes
  - All instantiated records inherit
  - Useful for tagging records with additional data
    - e.g. InstrMapping
    - Only when instantiated records are homogenous
- *defm* cannot have a body
  - Use let-statements instead

foreach

- Iterate over a list or range of integers

```
foreach Index = 0-15 in {
  def TTMP#Index#_vi   : SIReg<"ttmp"#Index, !add(112, Index)>;
  def TTMP#Index#_gfx9 : SIReg<"ttmp"#Index, !add(108, Index)>;
  def TTMP#Index       : SIReg<"", 0>;
}
```

- Can be used as an if-statement

```
class BoolToList<bit Value> {
  list<int> ret = !if(Value, [1]<int>, []<int>);
}

multiclass VOP1Inst <string opName, VOPProfile P,
                                    SDPatternOperator node = null_frag> {
  def _e32 : VOP1_Pseudo <opName, P>;
  def _e64 : VOP3_Pseudo <opName, P, getVOP1Pat64<node, P>.ret>;
  def _sdwa : VOP1_SDWA_Pseudo <opName, P>;
  foreach _ = BoolToList<P.HasExtDPP>.ret in
    def _dpp : VOP1_DPP_Pseudo <opName, P>;
}

defm V_MOV_B32 : VOP1Inst <"v_mov_b32", VOP_I32_I32>;
```

foreach vs. multiclass

- Both allow instantiating regular sets of records
- multiclass
  - Idiomatic for TableGen
  - Reusable
- foreach
  - Programmable
- Combine the best of both worlds!

defset

- Capture instantiated records
  - Captured records must be homogenous
  - Feed list into foreach
  - Generate derived heterogenous records

```
defset list<AMDGPUImageDimIntrinsic> AMDGPUImageDimAtomicIntrinsics = {
  defm int_amdgcn_image_atomic_swap : AMDGPUImageDimAtomic<"ATOMIC_SWAP">;
  defm int_amdgcn_image_atomic_add : AMDGPUImageDimAtomic<"ATOMIC_ADD">;
  defm int_amdgcn_image_atomic_sub : AMDGPUImageDimAtomic<"ATOMIC_SUB">;
  ...
}
```

```
class RsrcIntrinsic<AMDGPURsrcIntrinsic intr> {
  Intrinsic Intr = !cast<Intrinsic>(intr);
  bits<8> RsrcArg = intr.RsrcArg;
  bit IsImage = intr.IsImage;
}

def RsrcIntrinsics : GenericTable {
  let FilterClass = "RsrcIntrinsic";
  let Fields = ["Intr", "RsrcArg", "IsImage"];

  let PrimaryKey = ["Intr"];
  let PrimaryKeyName = "lookupRsrcIntrinsic";
}

foreach intr = !listconcat(AMDGPUBufferIntrinsics,
                           AMDGPUImageDimIntrinsics,
                           AMDGPUImageDimAtomicIntrinsics) in {
  def : RsrcIntrinsic<!cast<AMDGPURsrcIntrinsic>(intr)>;
}
```

- defset vs. heterogenous multiclass?
  - multiclass usually more idiomatic
  - defset can help isolate parts of .td files
    - Ex: intrinsics vs. backend definitions

Built-in functions

- Built-in functions prefixed with !
  - !eq   !ne   !le   !lt   !ge   !gt
  - !add   !shl   !sra   !srl   !and   !or
  - !if
  - !head   !tail   !listconcat   !size   !empty
  - !foreach   !foldl
  - !con   !dag
  - !strconcat   !subst
  - !isa   !cast
    - Allows casting between records and strings (by name)

Agenda

- What is TableGen?
  - TableGen, the tool and the language
  - Uses in LLVM
- TableGen language features
  - Type system
  - Classes
  - Let-statements and late evaluation
  - Multiclasses, foreach, and defset
  - Built-ins and “functional programming”
- Example: AMDGPU image intrinsics and instructions
  - Generic searchable tables backend

Example: AMDGPU image intrinsics

Why image operations are challenging

- Number of address operands ranges from 1 to 12
  - Different image dimensions (1D, 2D, 3D, MSAA, …)
  - Different operand types (float vs. half)
- Number of data operands ranges from 1 to 4
- Many different instructions with a partially regular structure

IMAGE_ATOMIC_ADD
IMAGE_ATOMIC_AND
IMAGE_ATOMIC_CMPSWAP
IMAGE_ATOMIC_DEC
IMAGE_ATOMIC_INC
IMAGE_ATOMIC_OR
IMAGE_ATOMIC_SMAX
IMAGE_ATOMIC_SMIN
IMAGE_ATOMIC_SUB
IMAGE_ATOMIC_SWAP
IMAGE_ATOMIC_UMAX
IMAGE_ATOMIC_UMIN
IMAGE_ATOMIC_XOR
IMAGE_GATHER4
IMAGE_GATHER4_B
IMAGE_GATHER4_B_CL
IMAGE_GATHER4_B_CL_O
IMAGE_GATHER4_B_O
IMAGE_GATHER4_C
IMAGE_GATHER4_CL
IMAGE_GATHER4_CL_O
IMAGE_GATHER4_C_B
IMAGE_GATHER4_C_B_CL
IMAGE_GATHER4_C_B_CL_O
IMAGE_GATHER4_C_B_O
IMAGE_GATHER4_C_CL
IMAGE_GATHER4_C_CL_O
IMAGE_GATHER4_C_L
IMAGE_GATHER4_C_LZ
IMAGE_GATHER4_C_LZ_O
IMAGE_GATHER4_C_L_O
IMAGE_GATHER4_C_O
IMAGE_GATHER4_L
IMAGE_GATHER4_LZ
IMAGE_GATHER4_LZ_O
IMAGE_GATHER4_L_O
IMAGE_GATHER4_O
IMAGE_GET_LOD
IMAGE_GET_RESINFO
IMAGE_LOAD
IMAGE_LOAD_MIP
IMAGE_LOAD_MIP_PCK
IMAGE_LOAD_MIP_PCK_SGN
IMAGE_LOAD_PCK
IMAGE_LOAD_PCK_SGN
IMAGE_SAMPLE
IMAGE_SAMPLE_B
IMAGE_SAMPLE_B_CL
IMAGE_SAMPLE_B_CL_O
IMAGE_SAMPLE_B_O
IMAGE_SAMPLE_C
IMAGE_SAMPLE_CD
IMAGE_SAMPLE_CD_CL
IMAGE_SAMPLE_CD_CL_O
IMAGE_SAMPLE_CD_O
IMAGE_SAMPLE_CL
IMAGE_SAMPLE_CL_O
IMAGE_SAMPLE_C_B
IMAGE_SAMPLE_C_B_CL
IMAGE_SAMPLE_C_B_CL_O
IMAGE_SAMPLE_C_B_O
IMAGE_SAMPLE_C_CD
IMAGE_SAMPLE_C_CD_CL
IMAGE_SAMPLE_C_CD_CL_O
IMAGE_SAMPLE_C_CD_O
IMAGE_SAMPLE_C_CL
IMAGE_SAMPLE_C_CL_O
IMAGE_SAMPLE_C_D
IMAGE_SAMPLE_C_D_CL
IMAGE_SAMPLE_C_D_CL_O
IMAGE_SAMPLE_C_D_O
IMAGE_SAMPLE_C_L
IMAGE_SAMPLE_C_LZ
IMAGE_SAMPLE_C_LZ_O
IMAGE_SAMPLE_C_L_O
IMAGE_SAMPLE_C_O
IMAGE_SAMPLE_D
IMAGE_SAMPLE_D_CL
IMAGE_SAMPLE_D_CL_O
IMAGE_SAMPLE_D_O
IMAGE_SAMPLE_L
IMAGE_SAMPLE_LZ
IMAGE_SAMPLE_LZ_O
IMAGE_SAMPLE_L_O
IMAGE_SAMPLE_O
IMAGE_STORE
IMAGE_STORE_MIP
IMAGE_STORE_MIP_PCK
IMAGE_STORE_PCK

Some intrinsic examples

```
%v = call <4 x float> @llvm.amdgcn.image.sample.1d.v4f32.f32(
                    i32 %dmask, float %s,
                    <8 x i32> %rsrc, <4 x i32> %samp, i1 %unorm, i32 %cachepolicy, i32 %texfail)

%v = call float @llvm.amdgcn.image.sample.d.o.2darray.f32.f32.f32(
                    i32 %dmask, i32 %offset, float %dsdh, float %dtdh, float %dsdv, float %dtdv, float %s, float %t, float %slice,
                    <8 x i32> %rsrc, <4 x i32> %samp, i1 %unorm, i32 %cachepolicy, i32 %texfail)

%v = call half @llvm.amdgcn.image.sample.c.l.2d.f16.f32(
                    i32 %dmask, float %zcompare, float %s, float %t, float %lod,
                    <8 x i32> %rsrc, <4 x i32> %samp, i1 %unorm, i32 %cachepolicy, i32 %texfail)

%v = call <4 x float> @llvm.amdgcn.image.sample.b.2d.v4f32.f32.f16(
                    i32 %dmask, float %bias, half %s, half %t,
                    <8 x i32> %rsrc, <4 x i32> %samp, i1 %unorm, i32 %cachepolicy, i32 %texfail)
```

Artifacts generated by TableGen

- Intrinsics
- Machine instructions
  - Multiple copies per “base opcode”: # of data channels, encoding, …
- NO SelectionDAG ISel patterns
  - Early versions used patterns, turned out too difficult to maintain
  - Instruction selection now in C++, aided by generic tables
- Generic tables
  - BaseOpcode enum and information (# extra address arguments, derivatives, …)
  - Mapping between intrinsic and (base opcode, image dimension)
  - Mapping between machine instruction and (base opcode, # of channels, encoding, …)

Generating argument lists, part 1: gradient arguments

```
class AMDGPUArg<LLVMType ty, string name> {
  LLVMType Type = ty;
  string Name = name;
}

class makeArgList<list<string> names, LLVMType basety> {
  list<AMDGPUArg> ret =
    !listconcat([AMDGPUArg<basety, names[0]>],
                !foreach(name, !tail(names), AMDGPUArg<LLVMMatchType<0>, name>));
}

class AMDGPUDimProps<string name, list<string> coord_names, list<string> slice_names> {
  ...
  list<AMDGPUArg> GradientArgs =
    makeArgList<!listconcat(!foreach(name, coord_names, "d" # name # "dh"),
                            !foreach(name, coord_names, "d" # name # "dv")),
                llvm_anyfloat_ty>.ret;
  bits<8> NumGradients = !size(GradientArgs);
  ...
}

def AMDGPUDim2DArray : AMDGPUDimProps<"2darray", ["s", "t"], ["slice"]>;
```

```
[AMDGPUArg<llvm_anyfloat_ty, "dsdh">,
 AMDGPUArg<LLVMMatchType<0>, "dtdh">,
 AMDGPUArg<LLVMMatchType<0>, "dsdv">,
 AMDGPUArg<LLVMMatchType<0>, "dtdv">]
```

Generating argument lists, part 2: combining argument lists

```
class AMDGPUDimProfile<string opmod,
                                          AMDGPUDimProps dim> {
  // These are intended to be overwritten by subclasses (late evaluation!)
  bit IsSample = 0;
  list<AMDGPUArg> ExtraAddrArgs = [];
  bit Gradients = 0;
  string LodClampMip = "";

  int NumRetAndDataAnyTypes = ...;

  list<AMDGPUArg> AddrArgs =
    arglistconcat<[ExtraAddrArgs,
                       !if(Gradients, dim.GradientArgs, []),
                       !listconcat(!if(IsSample, dim.CoordSliceArgs, dim.CoordSliceIntArgs),
                                   !if(!eq(LodClampMip, ""),
                                       []<AMDGPUArg>,
                                       [AMDGPUArg<LLVMMatchType<0>, LodClampMip>]))],
                      NumRetAndDataAnyTypes>.ret;
}
```

```
// Need to concatenate, and adjust LLVMMatchType references, for example in:

[AMDGPUArg<llvm_anyfloat_ty, "dsdh">, AMDGPUArg<LLVMMatchType<0>, "dtdh">,
 AMDGPUArg<LLVMMatchType<0>, "dsdv">, AMDGPUArg<LLVMMatchType<0>, "dtdv">]

[AMDGPUArg<llvm_anyfloat_ty, “s”>, AMDGPUArg<LLVMMatchType<0>, “t”>, AMDGPUArg<LLVMMatchType<0>, “slice”>]
```

Generating argument lists, part 3: adjusting LLVMMatchType

```
class arglistmatchshift<list<AMDGPUArg> arglist, int shift> {
  list<AMDGPUArg> ret =
    !foreach(arg, arglist,
                 !if(!isa<LLVMMatchType>(arg.Type),
                     AMDGPUArg<LLVMMatchType<!add(!cast<LLVMMatchType>(arg.Type).Number, shift)>,
                                     arg.Name>,
                     arg));
}

class arglistconcat<list<list<AMDGPUArg>> arglists, int shift = 0> {
  list<AMDGPUArg> ret =
    !foldl([]<AMDGPUArg>, arglists, lhs, rhs,
             !listconcat(lhs,
                         arglistmatchshift<rhs,
                                           !add(shift, !foldl(0, lhs, a, b,
                                                             !add(a, b.Type.isAny)))>.ret));
}
```

```
// Need to concatenate, and adjust LLVMMatchType references, for example in:

[AMDGPUArg<llvm_anyfloat_ty, "dsdh">, AMDGPUArg<LLVMMatchType<0>, "dtdh">,
 AMDGPUArg<LLVMMatchType<0>, "dsdv">, AMDGPUArg<LLVMMatchType<0>, "dtdv">]

[AMDGPUArg<llvm_anyfloat_ty, “s”>, AMDGPUArg<LLVMMatchType<0>, “t”>, AMDGPUArg<LLVMMatchType<0>, “slice”>]
```

Final thoughts

Agenda

- What is TableGen?
  - TableGen, the tool and the language
  - Uses in LLVM
- TableGen language features
  - Type system
  - Classes
  - Let-statements and late evaluation
  - Multiclasses, foreach, and defset
  - Built-ins and “functional programming”
- Example: AMDGPU image intrinsics and instructions
  - Generic searchable tables backend

Some possible directions for the future

- Type system
  - string vs. code – remove code type?
  - Introduce explicit top/bottom types (unset / any)
    - Allow heterogenous lists
- Extend # operator to list and dag concatenation
- Eliminate or cleanup multiclass inheritance
- Backends!
  - Better error messages!
  - Fix some of the DAG patterns pain?
    - More orthogonality between features (complex patterns, predicates, etc.)

Thank you!

DISCLAIMER AND ATTRIBUTIONS