How TableGen's DAGISel Backend Works

rtc-draper edited this page on May 26, 2014 · 10 revisions

Introduction

This document describes the LLVM **D**irected **A**cyclic **G**raph **I**nstruction **Sel**ector (DAGISel) TableGen backend.

1. **Why would I want to read this document?** You are interested in how LLVM does Instruction Selection. Particularly if you want to debug instruction selection on a specific backend, you want to modify it to make improvements, or you want to add a new backend for a register-based instruction set.
2. **What should I know to be able to follow along with this document?** You should be pretty familiar with LLVM and how LLVM Backend's work. See Getting started with LLVM for more information.
3. **What will I have learned by the end of this document?** You will know how LLVM processes the TableGen files to develop a MatcherTable structure that converts basic blocks of LLVM IR in directed acyclic graph (DAG) form to machine instructions in an equivalent DAG form. While this is not the whole compilation process, it is one of the critical pieces of LLVM's instruction selection process.

High Level Call Flow

The major functions to invoke the DAGISel Backend are as follows:

Function Prototype	Description
main(int argc, char **argv)	entry point, passes argv[0]
TableGenMain(argv[0], &LLVMTableGenMain)	Parses input file and opens output file as a raw_ostream.
LLVMTableGenMain(raw_ostream &OS, RecordKeeper &Records)	Calls the "emitter" functions using the Records (input files) and OS(output file)
EmitDAGISel(RecordKeeper &RK,	Calls DAGISelEmitter(RK).run(OS)

Function Prototype	Description
raw_ostream &OS)	
DAGISelEmitter(RecordKeeper &R)	Initialize CodeGenDAGPatterns with the Records.
CodeGenDAGPatterns(RecordKeeper &R)	This class parses the records through a number of "Parse*" functions, which often call Records.getAllDerivedDefinitions from the Records object. CodeGenDAGPatterns only does parsing relevant to ISel, and the element access is done through the RecordKeeper object.
DAGISelEmitter::run(raw_ostream &OS)	Use the CodeGenDAGPatterns object to generate PatternToMatch objects, create Matcher objects to generate commands to match each patterns, optimize these Matcher patterns, and finally emit them.

Inside of CodeGenDAGPattern are tree structures. Inside of the DAGISelEmitter are matcher objects corresponding to commands found in the MatcherTable array in the *ISelDAGPatterns.inc files generated by TableGen.

TableGen Patterns

CodeGenDAGPatterns::ParsePatterns(), called in the constructor, generates the final patterns (see the CodeGenDAGPatterns(RecordKeeper &R) constructor for the other functions that `` `ParsePatterns()` `` relies on. This function specifically gets the "Pattern" element from the Record definitions. As an example, we will now walk through the t2ADCri ARM instruction (t2 => thumb v2, ADC => ADD with Carry, r => register, i => immediate) in the following subsection.

Breakdown of a Sample Instruction t2ADCri

The instruction is defined in ARMInstrThumb2:

```
let hasPostISelHook = 1 in {
defm t2ADC  : T2I_adde_sube_irs<0b1010, "adc",
              BinOpWithFlagFrag<(ARMadde node:$LHS, node:$RHS, node:$FLAG)>, 1>;
defm t2SBC  : T2I_adde_sube_irs<0b1011, "sbc",
              BinOpWithFlagFrag<(ARMsube node:$LHS, node:$RHS, node:$FLAG)>>;
}
```

This definition turns into this Record object:

```
def t2ADCri {   // Instruction InstTemplate Encoding InstARM Thumb2sI T2sI T2sTwoRegImm Requir
  field bits<32> Inst = { 1, 1, 1, 1, 0, imm{11}, 0, 1, 0, 1, 0, s{0}, Rn{3}, Rn{2}, Rn{1}, Rn{0},
  field bits<32> Unpredictable = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  field bits<32> SoftFail = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  string Namespace = "ARM";
  dag OutOperandList = (outs rGPR:$Rd);
  dag InOperandList = (ins rGPR:$Rn, t2_so_imm:$imm, pred:$p, cc_out:$s);
  string AsmString = "adc${s}${p}       $Rd, $Rn, $imm";
  list<dag> Pattern = [(set rGPR:$Rd, CPSR, (anonymous.val.3708 rGPR:$Rn, t2_so_imm:$imm, CPSR))];
  list<Register> Uses = [CPSR];
  list<Register> Defs = [CPSR];
  ...
```

You can generate this definition with the following command: `llvm-tblgen /opt/llvm-trunk/lib/Target/ARM/ARM.td -I/opt/llvm-trunk/include -I/opt/llvm-trunk/lib/Target/ARM/ > ARM.stuff`

The TableGen pattern is on the following line:

```
list<dag> Pattern = [(set rGPR:$Rd, CPSR, (anonymous.val.3708 rGPR:$Rn, t2_so_imm:$imm, CPSR))
```

This Record breaks down as follows:

- The first line (`Inst` line) is the actual bitwise opcode for this instruction.
- `CPSR` is the Current Program Status Register in ARM-speak
- `rGPR` is a General Purpose Register
- `$Rd` is the destination
- `$Rn` is the source
- `t2_so_imm:$imm` is a thumb2 immediate (int, and 8 bits in this case) and an optional second operand to this form of the `ADC` instruction (the other variant being `ADCrr`).
- The pattern line is in a lisp-like form and should be read as "define a set of the following 3 elements ..." where the 3 elements are 2 register definitions/destinations and an instruction with operands.
- In this case, `CPSR` is implicitly defined and used, which is why it's last as a destination and as a source. Recall that `ADC` adds 2 operands and an additional +1 if the processors carry flag is set.

`anonymous.val.3708` references the following block in the target description table:

```
def anonymous.val.3708 {        // SDPatternOperator PatFrag BinOpWithFlagFrag
  string PatFrag:pred = "";
  SDNodeXForm PatFrag:xform = NOOP_SDNodeXForm;
  dag Operands = (ops node:$LHS, node:$RHS, node:$FLAG);
  dag Fragment = (ARMadde node:$LHS, node:$RHS, node:$FLAG);
  string PredicateCode = "";
  string ImmediateCode = "";
  SDNodeXForm OperandTransform = NOOP_SDNodeXForm;
  string NAME = ?;
}
```

`ARMadde` is the following block:

```
def ARMadde {   // SDPatternOperator SDNode
  list<SDNodeProperty> SDNode:props = [];
  string SDNode:sdclass = "SDNode";
  string Opcode = "ARMISD::ADDE";
  string SDClass = "SDNode";
  list<SDNodeProperty> Properties = [];
  SDTypeProfile TypeProfile = SDTBinaryArithWithFlagsInOut;
  string NAME = ?;
}
```

`SDTBinaryArithWithFlagsInOut` refers to:

```
def SDTBinaryArithWithFlagsInOut {      // SDTypeProfile
  int NumResults = 2;
  int NumOperands = 3;
  list<SDTypeConstraint> Constraints = [anonymous.val.3636, anonymous.val.3637, anonymous.val.3638,
  string NAME = ?;
}
```

`Constraints` are as follows:

```
def anonymous.val.3636 {        // SDTypeConstraint SDTCisSameAs
  int OperandNum = 0;
  int OtherOperandNum = 2;
  string NAME = ?;
}
def anonymous.val.3637 {        // SDTypeConstraint SDTCisSameAs
  int OperandNum = 0;
  int OtherOperandNum = 3;
  string NAME = ?;
}
def anonymous.val.3638 {        // SDTypeConstraint SDTCisInt
  int OperandNum = 0;
  string NAME = ?;
}
def anonymous.val.3639 {        // SDTypeConstraint SDTCisVT
  int OperandNum = 1;
  ValueType VT = i32;
  string NAME = ?;
}
...
def anonymous.val.3640 {        // SDTypeConstraint SDTCisVT
```

```
  int OperandNum = 4;
  ValueType VT = i32;
  string NAME = ?;
}
```

Operands 2 and 3 are the same as 0 (the registers), and 1 and 4 (`CPSR`) is an `i32` valuetype.

How the t2ADCri Instruction Gets Matched

Selection of `t2ADCri` is two-stage. A combined manual (in the legalizer) and TableGen'd matching system.

Stage 1 (Legalize)

In the first stage an `ISD::ADDE` instruction is manually translated to `ARMISD::ADDE` through a function call in ARMISelLowering.cpp:

```
static SDValue LowerADDC_ADDE_SUBC_SUBE(SDValue Op, SelectionDAG &DAG) {
  EVT VT = Op.getNode()->getValueType(0);
  SDVTList VTs = DAG.getVTList(VT, MVT::i32);

  unsigned Opc;
  bool ExtraOp = false;
  switch (Op.getOpcode()) {
  default: llvm_unreachable("Invalid code");
  case ISD::ADDC: Opc = ARMISD::ADDC; break;
  case ISD::ADDE: Opc = ARMISD::ADDE; ExtraOp = true; break;
  case ISD::SUBC: Opc = ARMISD::SUBC; break;
  case ISD::SUBE: Opc = ARMISD::SUBE; ExtraOp = true; break;
  }

  if (!ExtraOp)
    return DAG.getNode(Opc, Op->getDebugLoc(), VTs, Op.getOperand(0),
                       Op.getOperand(1));
  return DAG.getNode(Opc, Op->getDebugLoc(), VTs, Op.getOperand(0),
                     Op.getOperand(1), Op.getOperand(2));
}
```

This function gets called during ARMTargetLowering::LowerOperation function:

```
SDValue ARMTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) const {
  switch (Op.getOpcode()) {
  default: llvm_unreachable("Don't know how to custom lower this!");
  case ISD::ConstantPool:  return LowerConstantPool(Op, DAG);
  case ISD::BlockAddress:  return LowerBlockAddress(Op, DAG);
  case ISD::GlobalAddress:
    return Subtarget->isTargetDarwin() ? LowerGlobalAddressDarwin(Op, DAG) :
      LowerGlobalAddressELF(Op, DAG);
  case ISD::GlobalTLSAddress: return LowerGlobalTLSAddress(Op, DAG);
  case ISD::SELECT:        return LowerSELECT(Op, DAG);
```

```
    case ISD::SELECT_CC:     return LowerSELECT_CC(Op, DAG);
    case ISD::BR_CC:         return LowerBR_CC(Op, DAG);
    case ISD::BR_JT:         return LowerBR_JT(Op, DAG);
    case ISD::VASTART:       return LowerVASTART(Op, DAG);
    case ISD::MEMBARRIER:    return LowerMEMBARRIER(Op, DAG, Subtarget);
    case ISD::ATOMIC_FENCE:  return LowerATOMIC_FENCE(Op, DAG, Subtarget);
    case ISD::PREFETCH:      return LowerPREFETCH(Op, DAG, Subtarget);
    case ISD::SINT_TO_FP:
    case ISD::UINT_TO_FP:    return LowerINT_TO_FP(Op, DAG);
    case ISD::FP_TO_SINT:
    case ISD::FP_TO_UINT:    return LowerFP_TO_INT(Op, DAG);
    case ISD::FCOPYSIGN:     return LowerFCOPYSIGN(Op, DAG);
    case ISD::RETURNADDR:    return LowerRETURNADDR(Op, DAG);
    case ISD::FRAMEADDR:     return LowerFRAMEADDR(Op, DAG);
    case ISD::GLOBAL_OFFSET_TABLE: return LowerGLOBAL_OFFSET_TABLE(Op, DAG);
    case ISD::EH_SJLJ_SETJMP: return LowerEH_SJLJ_SETJMP(Op, DAG);
    case ISD::EH_SJLJ_LONGJMP: return LowerEH_SJLJ_LONGJMP(Op, DAG);
    case ISD::INTRINSIC_WO_CHAIN: return LowerINTRINSIC_WO_CHAIN(Op, DAG,
                                                                Subtarget);
    case ISD::BITCAST:       return ExpandBITCAST(Op.getNode(), DAG);
    case ISD::SHL:
    case ISD::SRL:
    case ISD::SRA:           return LowerShift(Op.getNode(), DAG, Subtarget);
    case ISD::SHL_PARTS:     return LowerShiftLeftParts(Op, DAG);
    case ISD::SRL_PARTS:
    case ISD::SRA_PARTS:     return LowerShiftRightParts(Op, DAG);
    case ISD::CTTZ:          return LowerCTTZ(Op.getNode(), DAG, Subtarget);
    case ISD::CTPOP:         return LowerCTPOP(Op.getNode(), DAG, Subtarget);
    case ISD::SETCC:         return LowerVSETCC(Op, DAG);
    case ISD::ConstantFP:    return LowerConstantFP(Op, DAG, Subtarget);
    case ISD::BUILD_VECTOR:  return LowerBUILD_VECTOR(Op, DAG, Subtarget);
    case ISD::VECTOR_SHUFFLE: return LowerVECTOR_SHUFFLE(Op, DAG);
    case ISD::INSERT_VECTOR_ELT: return LowerINSERT_VECTOR_ELT(Op, DAG);
    case ISD::EXTRACT_VECTOR_ELT: return LowerEXTRACT_VECTOR_ELT(Op, DAG);
    case ISD::CONCAT_VECTORS: return LowerCONCAT_VECTORS(Op, DAG);
    case ISD::FLT_ROUNDS_:   return LowerFLT_ROUNDS_(Op, DAG);
    case ISD::MUL:           return LowerMUL(Op, DAG);
    case ISD::SDIV:          return LowerSDIV(Op, DAG);
    case ISD::UDIV:          return LowerUDIV(Op, DAG);
    case ISD::ADDC:
    case ISD::ADDE:
    case ISD::SUBC:
    case ISD::SUBE:          return LowerADDC_ADDE_SUBC_SUBE(Op, DAG);
    case ISD::ATOMIC_LOAD:
    case ISD::ATOMIC_STORE:  return LowerAtomicLoadStore(Op, DAG);
    }
  }
```

Note that, for each architecture, there are a limited number of instructions which need some manual intervention, and `t2ADCri` is one of them.

This function is called in a number of places in the Legalize phase. Before that, calls to `ExpandIntegerResult` trigger the `ISD::ADDE` (converted from `ISD::ADD`), which in the `t2ADCri` case is triggered when you need to add 64 bit (or other large) numbers and the second add takes care of the top half and carries over from the first add. An example of LLVM selection code that triggers such an instruction is found in TargetLoweringBase.cpp:

```
// Every integer value type larger than this largest register takes twice as
// many registers to represent as the previous ValueType.
for (unsigned ExpandedReg = LargestIntReg + 1;
     ExpandedReg <= MVT::LAST_INTEGER_VALUETYPE; ++ExpandedReg) {
  NumRegistersForVT[ExpandedReg] = 2*NumRegistersForVT[ExpandedReg-1];
  RegisterTypeForVT[ExpandedReg] = (MVT::SimpleValueType)LargestIntReg;
  TransformToType[ExpandedReg] = (MVT::SimpleValueType)(ExpandedReg - 1);
  ValueTypeActions.setTypeAction((MVT::SimpleValueType)ExpandedReg,
                                 TypeExpandInteger);
}
```

The last line in that code block sets `TypeExpandInteger` which will trigger the legalizer to call ExpandIntegerResult, which in the case of an add will call `ExpandIntRes_ADDSUB`, which then creates an `ADDC` and `ADDE` instruction for the low and high bits of the extended integer value, respectively.

Stage 2 (ISel)

In the second stage, `t2ADCri` is matched in the `MatcherTable` and converted into it's "machine-ready" form. The `MatcherTable` code for the `t2ADCri` function is as follows:

```
/*37354*/     /*Scope*/ 35, /*->37390*/
/*37355*/       OPC_CheckPredicate, 9, // Predicate_t2_so_imm
/*37357*/       OPC_MoveParent,
/*37358*/       OPC_RecordChild2, // #2 = physreg input CPSR
/*37359*/       OPC_CheckType, MVT::i32,
/*37361*/       OPC_CheckPatternPredicate, 5, // (Subtarget->isThumb2())
/*37363*/       OPC_EmitConvertToTarget, 1,
/*37365*/       OPC_EmitInteger, MVT::i32, 14,
/*37368*/       OPC_EmitRegister, MVT::i32, 0 /*zero_reg*/,
/*37371*/       OPC_EmitRegister, MVT::i32, 0 /*zero_reg*/,
/*37374*/       OPC_EmitCopyToReg, 2, ARM::CPSR,
/*37377*/       OPC_MorphNodeTo, TARGET_VAL(ARM::t2ADCri), 0|OPFL_GlueInput,
                    2/*#VTs*/, MVT::i32, MVT::i32, 5/*#Ops*/, 0, 3, 4, 5, 6,
                // Src: (ARMadde:i32:i32 rGPR:i32:$Rn, (imm:i32)<<P:Predicate_t2_so_imm>>:$imm, CPS
                // Dst: (t2ADCri:i32:i32 rGPR:i32:$Rn, (imm:i32):$imm)
```

The `Src` and `Dst` lines summarize the operation. The before pattern (in the DAG structure) should match the `Src`, and the after pattern should match the `Dst`. These lines can be interpreted as follows:

```
// Src: (ARMadde:i32:i32 rGPR:i32:$Rn, (imm:i32)<<P:Predicate_t2_so_imm>>:$imm, CPSR:i32) - Co
```

- `ARMadde:i32:i32` instruction
- 1st operand is a `GPR` which is the first operand and the destination register
- 2nd operand is a 32 bit immediate which meets the Predicate `t2_so_imm` (can fit in 8 bits)
- 3rd operand is the `CPSR`

```
// Dst: (t2ADCri:i32:i32 rGPR:i32:$Rn, (imm:i32):$imm)
```

- `t2ADCri:i32:i32` instruction
- 1st operand is the `GPR` as above
- 2nd operand is the immediate value
- Note that the `CPSR` , since it is implied, is no longer listed in the operands list.

Walkthrough of MatcherTable operations

MatcherTable operations are defined and performed in SelectionDAGISel.cpp. The actual Matcher table (and target specific functions) are stored in a target specific file:

- lib/Target/ARM/ARMGenDAGISel.inc

To get this file, you need to compile LLVM with ARM target support.

The following subsections are an explanation of the operation of these different `MatcherTable` operations.

OPC_CheckPredicate

```
/*37355*/       OPC_CheckPredicate, 9, // Predicate_t2_so_imm
```

A target specific function, case 9 in this situation is `Predicate_t2_so_imm` (note that this code block is defined in the `ARMGenDAGISel.inc` file:

```
case 9: { // Predicate_t2_so_imm
  int64_t Imm = cast<ConstantSDNode>(Node)->getSExtValue();

  return ARM_AM::getT2SOImmVal(Imm) != -1;

}
```

getT2SOImmVal is defined in ARMAddressingModes.h:

```
/// getT2SOImmVal - Given a 32-bit immediate, if it is something that can fit
/// into a Thumb-2 shifter_operand immediate operand, return the 12-bit
/// encoding for it.  If not, return -1.
/// See ARM Reference Manual A6.3.2.
static inline int getT2SOImmVal(unsigned Arg) {
  // If 'Arg' is an 8-bit splat, then get the encoded value.
  int Splat = getT2SOImmValSplatVal(Arg);
  if (Splat != -1)
    return Splat;

  // If 'Arg' can be handled with a single shifter_op return the value.
  int Rot = getT2SOImmValRotateVal(Arg);
  if (Rot != -1)
    return Rot;

  return -1;
}
```

Note that, in this example, the `MatcherTable` performs this operation in a tree at the 2nd operand (the immediate). You can look further up the table in the `ARMGenDAGISel.inc` file to determine how it gets there (first by checking that the instruction is an `ARMISD::ADDE` operation, then by checking that the first operand is an `i32 GPR`, and so on).

OPC_MoveParent

```
/*37357*/       OPC_MoveParent,
```

Pop the current node off of the NodeStack and move to the parent (the `ARMISD::ADDE` node).

OPC_RecordChild2

```
/*37358*/       OPC_RecordChild2, // #2 = physreg input CPSR
```

Save the second child (3rd operand) onto the output stack.

OPC_CheckType

```
/*37359*/       OPC_CheckType, MVT::i32,
```

Check that the instruction returns an `MVT::i32` type.

OPC_CheckPatternPredicate

```
/*37361*/       OPC_CheckPatternPredicate, 5, // (Subtarget->isThumb2())
```

Check that the instruction is from pattern predicate #5 (thumb 2). This is a machine specific function that calls into the sub-target description, as follows:

```
case 5: return (Subtarget->isThumb2());
```

OPC_EmitConvertToTarget

```
/*37363*/       OPC_EmitConvertToTarget, 1,
```

Convert the 2nd operand (array index 1) to the target type and save it.

OPC_EmitInteger

```
/*37365*/       OPC_EmitInteger, MVT::i32, 14,
```

Create an integer (14) and save it as an `MVT::i32`. This shows up as a `pred:` value in the list form of the decoded output. Definition is:

```
// ARM Predicate operand. Default to 14 = always (AL). Second part is CC
// register whose default is 0 (no register).
def CondCodeOperand : AsmOperandClass { let Name = "CondCode"; }
def pred : PredicateOperand<OtherVT, (ops i32imm, i32imm),
                                     (ops (i32 14), (i32 zero_reg))> {
  let PrintMethod = "printPredicateOperand";
  let ParserMatchClass = CondCodeOperand;
  let DecoderMethod = "DecodePredicateOperand";
}
```

OPC_EmitRegister

```
/*37368*/       OPC_EmitRegister, MVT::i32, 0 /*zero_reg*/,
```

Save register 0 (zero reg is unknown in arm speak). (`%noreg` shows up in decoded output under `pred`).

Definition is here:

```
// Conditional code result for instructions whose 's' bit is set, e.g. subs.
def CCOutOperand : AsmOperandClass { let Name = "CCOut"; }
def cc_out : OptionalDefOperand<OtherVT, (ops CCR), (ops (i32 zero_reg))> {
  let EncoderMethod = "getCCOutOpValue";
  let PrintMethod = "printSBitModifierOperand";
  let ParserMatchClass = CCOutOperand;
```

```
    let DecoderMethod = "DecodeCCOutOperand";
  }

  // Same as cc_out except it defaults to setting CPSR.
  def s_cc_out : OptionalDefOperand<OtherVT, (ops CCR), (ops (i32 CPSR))> {
    let EncoderMethod = "getCCOutOpValue";
    let PrintMethod = "printSBitModifierOperand";
    let ParserMatchClass = CCOutOperand;
    let DecoderMethod = "DecodeCCOutOperand";
  }
```

OPC_EmitCopyToReg

```
/*37374*/       OPC_EmitCopyToReg, 2, ARM::CPSR,
```

This emits a `CopyToReg` node with the 3rd operand copying into the `ARM::CPSR` register.

OPC_MorphNodeTo

```
/*37377*/       OPC_MorphNodeTo, TARGET_VAL(ARM::t2ADCri), 0|OPFL_GlueInput,
                    2/*#VTs*/, MVT::i32, MVT::i32, 5/*#Ops*/, 0, 3, 4, 5, 6,
```

Create the `ARM::t2ADCri` node, and use current operand 0 (the `GPR`) and 2 (`CPSR`) as the destinations, then use 0 (`GPR`), 3 (the emitted integer), 4 (`pred:14`), 5 (`pred:%noreg`), 6 (`opt:%noreg`) as the operands for the instruction. Note also that the `OPFL_GlueInput` causes the backend to add `MVT::Glue` as the first value type (by convention this indicates the machine node as an instruction that is glued, or partially ordered, to another instruction).

Summary of t2ADCri

Looking back at the first pattern line and it's reference:

```
list<dag> Pattern = [(set rGPR:$Rd, CPSR, (anonymous.val.3708 rGPR:$Rn, t2_so_imm:$imm, CPSR))
...
def anonymous.val.3708 {        // SDPatternOperator PatFrag BinOpWithFlagFrag
  string PatFrag:pred = "";
  SDNodeXForm PatFrag:xform = NOOP_SDNodeXForm;
  dag Operands = (ops node:$LHS, node:$RHS, node:$FLAG);
  dag Fragment = (ARMadde node:$LHS, node:$RHS, node:$FLAG);
  string PredicateCode = "";
  string ImmediateCode = "";
  SDNodeXForm OperandTransform = NOOP_SDNodeXForm;
  string NAME = ?;
}
```

We have enough info to conclude that the input pattern for this instruction is something like:

```
ARMadde rGPR:$Rn, t2_so_imm:$imm, CPSR
```

The output pattern is determined using all of the other additional instruction information. We also know that the immediate has to fit within the 8 bit `t2_so_imm` encoding (as specified in the predicate function).

How TableGen Patterns Get Created

The *.td files are put into a RecordKeeper object, then this object is used to create the CodeGenDAGPatterns object.

Code Gen Dag Patterns

The `CodeGenDAGPatterns` constructor handles all of the parsing to create the patterns:

```
CodeGenDAGPatterns::CodeGenDAGPatterns(RecordKeeper &R) :
  Records(R), Target(R) {

  Intrinsics = LoadIntrinsics(Records, false);
  TgtIntrinsics = LoadIntrinsics(Records, true);
  ParseNodeInfo();
  ParseNodeTransforms();
  ParseComplexPatterns();
  ParsePatternFragments();
  ParseDefaultOperands();
  ParseInstructions();
  ParsePatterns();

  // Generate variants.  For example, commutative patterns can match
  // multiple ways.  Add them to PatternsToMatch as well.
  GenerateVariants();

  // Infer instruction flags.  For example, we can detect loads,
  // stores, and side effects in many cases by examining an
  // instruction's pattern.
  InferInstructionFlags();

  // Verify that instruction flags match the patterns.
  VerifyInstructionFlags();
}
```

ParseInstructions() is the most important of these calls. All the instructions are grabbed through:

```
std::vector<Record*> Instrs = Records.getAllDerivedDefinitions("Instruction");
```

The first thing it does is check if the current instruction does not have a complete pattern:

```
// If there is no pattern, only collect minimal information about the
// instruction for its operand list.  We have to assume that there is one
// result, as we have no detailed info. A pattern which references the
// null_frag operator is as-if no pattern were specified. Normally this
// is from a multiclass expansion w/ a SDPatternOperator passed in as
// null_frag.
if (!LI || LI->getSize() == 0 || hasNullFragReference(LI)) {
  std::vector<Record*> Results;
  std::vector<Record*> Operands;

  CodeGenInstruction &InstInfo = Target.getInstruction(Instrs[i]);

  if (InstInfo.Operands.size() != 0) {
    if (InstInfo.Operands.NumDefs == 0) {
      // These produce no results
      for (unsigned j = 0, e = InstInfo.Operands.size(); j < e; ++j)
        Operands.push_back(InstInfo.Operands[j].Rec);
    } else {
      // Assume the first operand is the result.
      Results.push_back(InstInfo.Operands[0].Rec);

      // The rest are inputs.
      for (unsigned j = 1, e = InstInfo.Operands.size(); j < e; ++j)
        Operands.push_back(InstInfo.Operands[j].Rec);
    }
  }

  // Create and insert the instruction.
  std::vector<Record*> ImpResults;
  Instructions.insert(std::make_pair(Instrs[i],
                      DAGInstruction(0, Results, Operands, ImpResults)));
  continue;  // no pattern.
}
```

The above is important because we will need to "fill in the blanks" with instructions that don't have complete patterns.

`CodeGenDAGPatterns` creates a `TreePattern` object for the pattern, which is a list where each element in the instruction represents a tree of possible patterns that the instruction matches (i.e., a list of trees, where each tree represents the possible patterns which could be matched to this instruction).

A `TreePattern` is initialized for each instruction `Record` using the `ListInit` of the `Pattern` value. After the pattern is parsed and `TreePattern` is initialized, the pattern matching that of the instruction itself is checked against the pattern (this is what the `CGI` variable is doing). Results and operands for the instruction are checked against those found in the pattern, and if any ins/outs in the instruction exist which can't be found an error is reported.

Towards the end of the function, a `ResultPattern` is generated from `ResultNodeOperands` calculated during the check. This constitutes the resultant pattern, which includes transform functions that must be performed on the operand. `TheInst` becomes a `DAGInstruction` , and a temporary pattern representing the `ResultPattern` is set as the result pattern in the `DAGInstruction` object.

At the end, if a pattern exists it is added through `AddPatternToMatch` as a `PatternToMatch` object which records the Record, predicates, source and destination patterns, implicit destination registers, and numbers representing complexity and uid.

```
PatternToMatch(Record *srcrecord, ListInit *preds,
               TreePatternNode *src, TreePatternNode *dst,
               const std::vector<Record*> &dstregs,
               unsigned complexity, unsigned uid)
  : SrcRecord(srcrecord), Predicates(preds), SrcPattern(src), DstPattern(dst),
    Dstregs(dstregs), AddedComplexity(complexity), ID(uid) {}

Record          *SrcRecord;   // Originating Record for the pattern.
ListInit        *Predicates;  // Top level predicate conditions to match.
TreePatternNode *SrcPattern;  // Source pattern to match.
TreePatternNode *DstPattern;  // Resulting pattern.
std::vector<Record*> Dstregs; // Physical register defs being matched.
unsigned         AddedComplexity; // Add to matching pattern complexity.
unsigned         ID;          // Unique ID for the record.
```

Examples of `TreePattern` structures can be generated with:

- `llvm-tblgen /opt/llvm-trunk/lib/Target/X86/X86.td -I/opt/llvm-trunk/include -I/opt/llvm-trunk/lib/Target/X86/ -gen-dag-isel -debug > blah 2> debuginfo.txt`

```
ADC16ri:        (set GR16:i16:$dst, EFLAGS:i32, (X86adc_flag:i16:i32 GR16:i16:$src1, (imm:i16) c
ADC16ri8:       (set GR16:i16:$dst, EFLAGS:i32, (X86adc_flag:i16:i32 GR16:i16:$src1, (imm:i16)<<P:P
ADC16rm:        (set GR16:i16:$dst, EFLAGS:i32, (X86adc_flag:i16:i32 GR16:i16:$src1, (ld:i16 addr:i
ADC16rr:        (set GR16:i16:$dst, EFLAGS:i32, (X86adc_flag:i16:i32 GR16:i16:$src1, GR16:i16:$src2
LEA32r:         (set GR32:i32:$dst, lea32addr:i32:$src)
LEA64_32r:      (set GR32:i32:$dst, lea64_32addr:i32:$src)
LEA64r:         (set GR64:i64:$dst, lea64addr:i64:$src)
// Multiple TreePatternNode's:
SUB16mi: [
        (st (sub:i16 (ld:i16 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (imm
        (implicit EFLAGS:i32)
]
 SUB16mi8: [
        (st (sub:i16 (ld:i16 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (imm
        (implicit EFLAGS:i32)
]
```

And here is how the patterns are paired:

```
PATTERN: (st (imm:i8):$src, addr:iPTR:$dst)<<P:Predicate_unindexedstore>><<P:Predicate_store>>
RESULT:  (MOV8mi addr:iPTR:$dst, (imm:i8):$src)
PATTERN: (st (sub:i16 (ld:i16 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (im
RESULT:  (SUB16mi8:i32 addr:iPTR:$dst, (imm:i16):$src)
PATTERN: (st (sub:i32 (ld:i32 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (im
RESULT:  (SUB32mi8:i32 addr:iPTR:$dst, (imm:i32):$src)
PATTERN: (st (sub:i64 (ld:i64 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (im
RESULT:  (SUB64mi8:i32 addr:iPTR:$dst, (imm:i64):$src)
PATTERN: (st (sub:i8 (ld:i8 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (imm:
RESULT:  (SUB8mi:i32 addr:iPTR:$dst, (imm:i8):$src)
PATTERN: (st (sub:i16 (ld:i16 addr:iPTR:$dst)<<P:Predicate_unindexedload>><<P:Predicate_load>>, (im
RESULT:  (SUB16mi:i32 addr:iPTR:$dst, (imm:i16):$src)
PATTERN: (X86adc_flag:i32:i32 GR32:i32:$src1, (imm:i32):$src2, EFLAGS:i32)
RESULT:  (ADC32ri:i32:i32 GR32:i32:$src1, (imm:i32):$src2)
PATTERN: lea32addr:i32:$src
RESULT:  (LEA32r:i32 lea32addr:i32:$src)
PATTERN: lea64_32addr:i32:$src
RESULT:  (LEA64_32r:i32 lea64_32addr:i32:$src)
PATTERN: lea64addr:i64:$src
RESULT:  (LEA64r:i64 lea64addr:i64:$src)
```

After this, variants are created to handle different possible variations for commutative patterns and instruction flags are inferred to detect loads/stores and side effects based on the pattern.

Matcher Tables

Matcher tables are created by a `MatcherGen` object, which converts a pattern into a `Matcher` object. A matcher object has the following types:

```
Scope,                // Push a checking scope.
RecordNode,           // Record the current node.
RecordChild,          // Record a child of the current node.
RecordMemRef,         // Record the memref in the current node.
CaptureGlueInput,     // If the current node has an input glue, save it.
MoveChild,            // Move current node to specified child.
MoveParent,           // Move current node to parent.

// Predicate checking.
CheckSame,            // Fail if not same as prev match.
CheckPatternPredicate,
CheckPredicate,       // Fail if node predicate fails.
CheckOpcode,          // Fail if not opcode.
SwitchOpcode,         // Dispatch based on opcode.
CheckType,            // Fail if not correct type.
SwitchType,           // Dispatch based on type.
CheckChildType,       // Fail if child has wrong type.
CheckInteger,         // Fail if wrong val.
CheckCondCode,        // Fail if not condcode.
```

```
  CheckValueType,
  CheckComplexPat,
  CheckAndImm,
  CheckOrImm,
  CheckFoldableChainNode,

  // Node creation/emisssion.
  EmitInteger,          // Create a TargetConstant
  EmitStringInteger,    // Create a TargetConstant from a string.
  EmitRegister,         // Create a register.
  EmitConvertToTarget,  // Convert a imm/fpimm to target imm/fpimm
  EmitMergeInputChains, // Merge together a chains for an input.
  EmitCopyToReg,        // Emit a copytoreg into a physreg.
  EmitNode,             // Create a DAG node
  EmitNodeXForm,        // Run a SDNodeXForm
  MarkGlueResults,      // Indicate which interior nodes have glue results.
  CompleteMatch,        // Finish a match and update the results.
  MorphNodeTo           // Build a node, finish a match and update results.
```

These types refer to specific operations done on the basic block graphs as the compiler progresses through the instructions of the function. The `MatcherGen` creates a tree of matcher objects of various types to match the pattern as seen in the graph and perform operations for instruction selection.

Clone this wiki locally

```
https://github.com/draperlaboratory/fracture.wiki.git
```