How An IR Statement Becomes An Instruction

Edit New page

Jump to bottom

rtc-draper edited this page on Apr 12, 2014 · 4 revisions

The following is a (dated) walkthrough of the call chains that occur during compilation:

```
tools/llc/llc.cpp:366
-> addPassesToEmitFile causes the specified target to add the appropriate passes
   to generate the source code file.

include/llvm/Target/TargetMachine.h:247
-> Each machine implements this interface.

lib/CodeGen/LLVMTargetMachine.cpp:137
-> Where addPassesToEmitFile is implemented for all architectures at this time.
-> Calls addPassesToGenerateCode, line 144
-> Creates target-specific MCInstPrinter or MCCodeEmitter to print instructions.

lib/CodeGen/LLVMTargetMachine.cpp:87
-> static MCContext *addPassesToGenerateCode(...)
-> Performs the following (which do what you expect):
        PassConfig->addIRPasses();
        PassConfig->addCodeGenPrepare();
        PassConfig->addPassesToHandleExceptions();
        PassConfig->addISelPrepare();
-> Key line is: (127)
  if (PassConfig->addInstSelector())

lib/Target/ARM/ARMTargetMachine.cpp:133/151
-> implements addInstSelector().
-> Calls createARMISelDag(...)

lib/Target/ARM/ARMISelDAGToDAG.cpp:3463
-> ARMDAGToDAGISel(...)
-> This is the instruction selection pass impl.

lib/Target/ARM/ARMISelDAGToDAG.cpp:70
-> SelectionDAGISel(...)

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:269
-> This initializes a bunch of different things:
SelectionDAGISel::SelectionDAGISel(const TargetMachine &tm,
```

```
                              CodeGenOpt::Level OL) :
  MachineFunctionPass(ID), TM(tm), TLI(*tm.getTargetLowering()),
  FuncInfo(new FunctionLoweringInfo(TLI)),
  CurDAG(new SelectionDAG(tm, OL)),
  SDB(new SelectionDAGBuilder(*CurDAG, *FuncInfo, OL)),
  GFI(),
  OptLevel(OL),
  DAGSize(0) {
        initializeGCModuleInfoPass(*PassRegistry::getPassRegistry());
        initializeAliasAnalysisAnalysisGroup(*PassRegistry::getPassRegistry());
        initializeBranchProbabilityInfoPass(*PassRegistry::getPassRegistry());
        initializeTargetLibraryInfoPass(*PassRegistry::getPassRegistry());
}

/**
 * At this point, the pass is scheduled to run...
 * Inheritance is as follows:
 * Pass
 *      -> FunctionPass
 *              -> MachineFunctionPass
 *                      -> SelectionDAGISel
 *                              -> ARMDAGToDAGISel
 * The function "runOnFunction(Function &F)" is used to run the pass.
 **/

include/llvm/Pass.h:303
   -> virtual bool runOnFunction(Function &F) = 0;

lib/CodeGen/MachineFunctionPass.cpp:26
-> runOnMachineFunction(MF) (which is = 0 in the .h file).

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:339
-> bool SelectionDAGISel::runOnMachineFunction(MachineFunction &mf) {
-> Calls "SelectAllBasicBlocks(Fn)", which performs code generation on each block.
-> It does a lot of post-generation cleanup at the function level(see source for details).

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:987
-> void SelectionDAGISel::SelectAllBasicBlocks(const Function &Fn) {
-> Differentiates between fastIsel and regular isel, insts which can't be done under fastisel
fall back to the regular one
-> 1171: SelectBasicBlock(Begin, BI, HadTailCall); -- CodeGen's the BB (non-fastisel, which we
skip for now).
-> Also updates some state (phi nodes, and debug info).

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:493
-> void SelectionDAGISel::SelectBasicBlock(BasicBlock::const_iterator Begin,
-> Calls CodeGenAndEmitDAG(), which operates on the basic block

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:552
-> void SelectionDAGISel::SelectBasicBlock(BasicBlock::const_iterator Begin,
-> This function has a few steps to it:
        -> A DAG combiner (which simplifies the DAG)
        -> A DAG Legalizer (which converts into operations the targets support).
```

```
        -> Another DAG combiner step.
        -> Legalize again if it changes
        -> DoInstructionSelection() (which does what you think it does)
        -> Schedule machine code, which reorders instructions based on state

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:731
-> void SelectionDAGISel::DoInstructionSelection() {
-> Topologically sorts the nodes by node and operand reference order.
        The order is operands then operation.
        unsigned SelectionDAG::AssignTopologicalOrder()
        (lib/CodeGen/SelectionDAG/SelectionDAG.cpp:5691)
-> Calls "Select" which is where the magic happens.

include/llvm/CodeGen/SelectionDAGISel.h:80
->  /// Select - Main hook targets implement to select a node.
        virtual SDNode *Select(SDNode *N) = 0;

lib/Target/ARM/ARMISelDAGToDAG.cpp:2547
-> SDNode *ARMDAGToDAGISel::Select(SDNode *N) {
-> Contains cases for specially handled SDNodes (of which there are alot).
-> calls "SelectCode(N)" on line 3446

${build}/lib/Target/ARM/ARMGenDAGISel.inc:13
-> Autogenerated by tablegen
-> SDNode *ARMDAGToDAGISel::Select(SDNode *N) {
-> Defines FSM "MatcherTable" which matches patterns in the graph with instruction patterns.
-> Calls "SelectCodeCommon" on line 30329

lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:2184
-> SDNode *SelectionDAGISel::SelectCodeCommon(SDNode *NodeToMatch, const unsigned char
*MatcherTable,
             unsigned TableSize) {
-> This runs the state machine operations to regenerate the graph.

/**
 * Later on, InstrEmitter is used to generate the instruction. See
lib/CodeGen/SelectionDAG/InstrEmitter.cpp:689 (EmitMachineNode)
 **/
```

Pages 8

Find a page...

Home

A Beginner's Guide to Fracture

+ Add a custom sidebar

Clone this wiki locally

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