Building an LLVM Backend

Fraser Cormack
Pierre-Andre Saulais

Codeplay Software
@codeplaysoft

Introduction

- Yet another talk about creating a LLVM target?
- LLVM backend crash course, for beginners
  - How-tos and tips
  - Solution to common problems
- Example target created for this tutorial
  - Can be used to see how LLVM works
  - Can be used as a skeleton to bootstrap new target

Overview

- Part 1: Background
- Part 2: Creating your own target
- Part 3: How-tos for specific tasks
- Part 4: Troubleshooting and tips

Part 1

Background

What you need to start

- Know a little bit about LLVM IR: llvm.org/docs/LangRef.html
- xdot.py to visualize graphs when debugging: github.com/jrfonseca/xdot.py
- Check out and build our LLVM repo from GitHub: github.com/frasercrmck/llvm-leg
- This informative and well-presented talk!

Example target: LEG

- Simple, RISC-like architecture
  - Very small subset of ARM
- 12 integer registers (32-bit)
  - r0, r1, …, r9, sp (stack pointer), lr (return address)
- Instructions:
  - 32-bit arithmetic (add, subtract, multiply, mad)
  - 32-bit register move, 16-bit constant moves
  - load, store, branch, branch and link

Calling convention for LEG

- How values are passed to/from a function
- Arguments in r0 (1st), r1 (2nd), …, r3 (4th)
  - Further arguments passed on the stack
- Return value in r0

```
int foo(int a, int b) {
    int result = a + b;   // r0 + r1
    return result;        // r0
}
```
ex1.c

```
.foo:
    add r0, r0, r1
    b lr
```
ex1.s

The big picture

- Pipeline structure of the backend:

  IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

- Transforms your program many times
  - Same program, few different representations
  - Different instruction namespaces
  - Check it out (IR and MI only):
    - llc foo.ll -print-after-all 2>&1 > foo.log

A look at an IR module

- High-level, linear representation
- Typed values, no registers
- Target-agnostic
  - Exceptions: data layout, triple, intrinsics

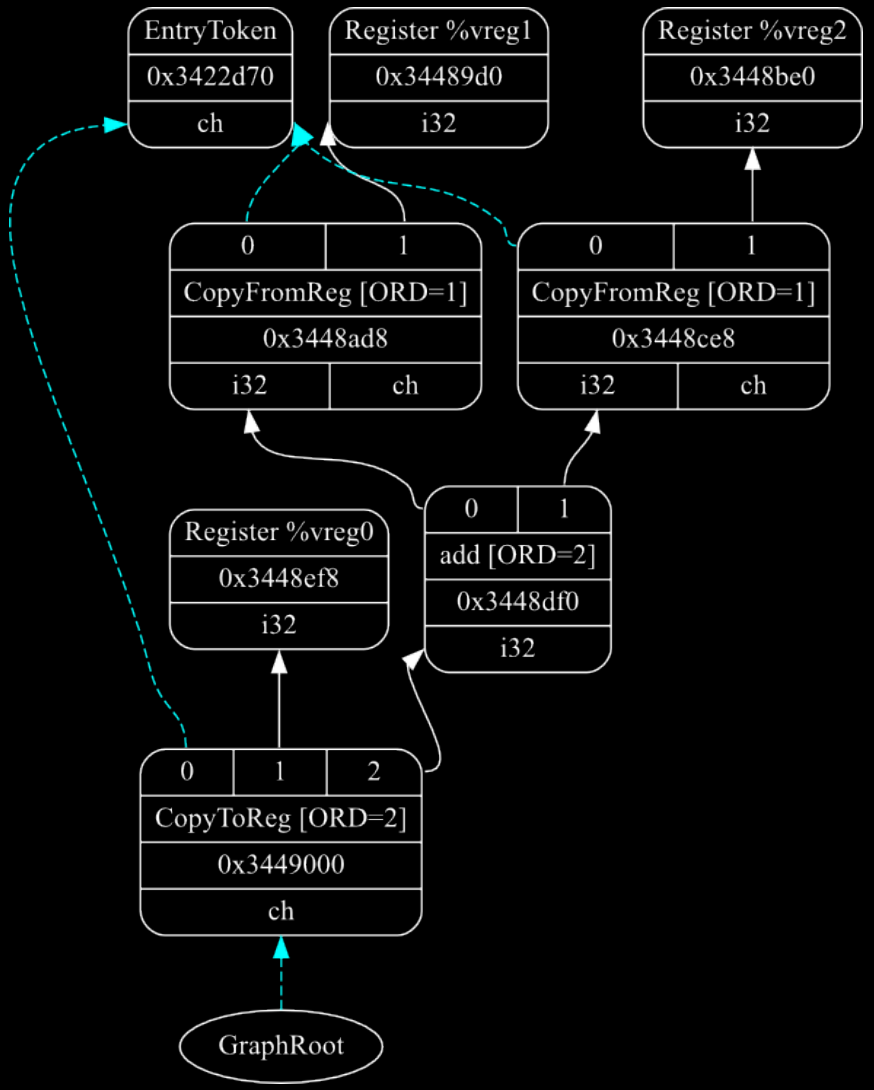
```
target datalayout = "e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-
i64:32-f64:32-a:0:32-n32"
target triple = "leg"

define i32 @foo(i32 %a, i32 %b) {
    %c = add nsw i32 %a, %b
    ret i32 %c
}
```

ex1.ll

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

A look at a SelectionDAG graph

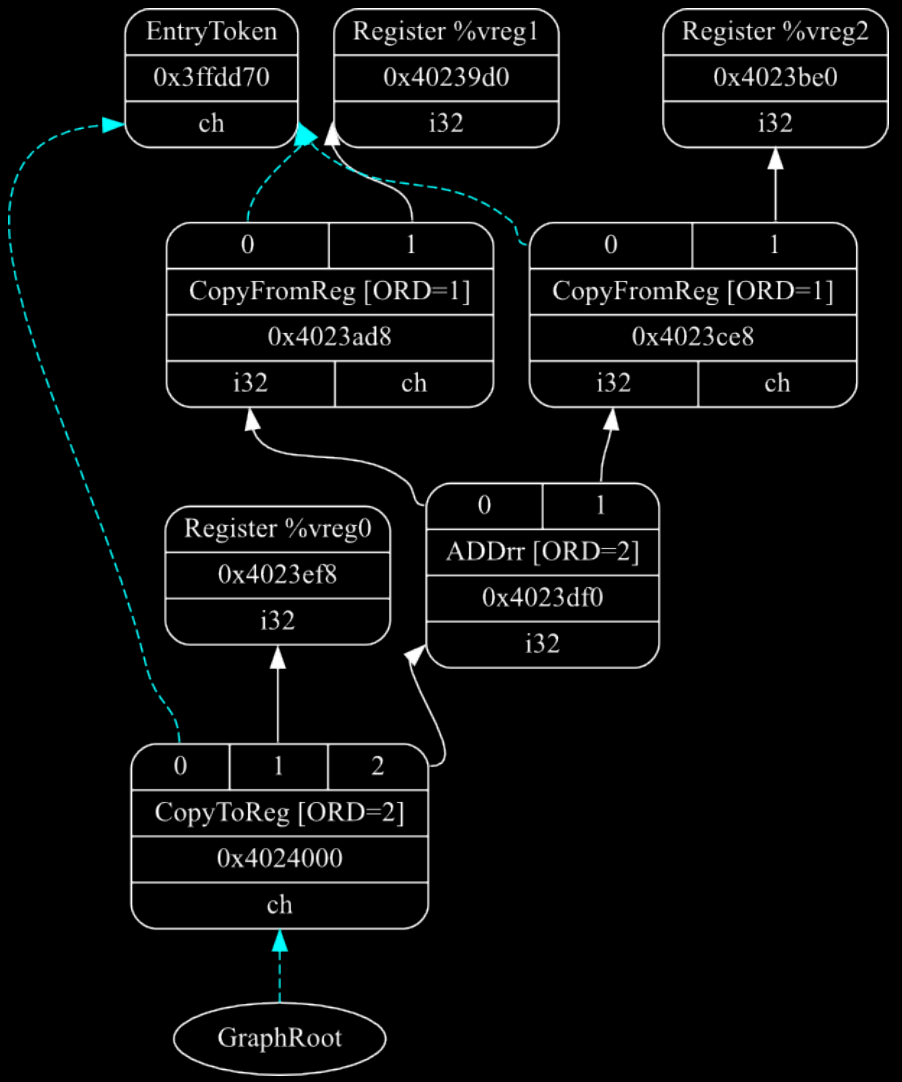
- Graph representation
- Operations as nodes
  - Mostly target-agnostic
  - LEGISD (ISD) namespace
- Dependencies as edges
  - Data
  - Order (“chain”)
  - Scheduling (“glue”)
- Typed values



IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

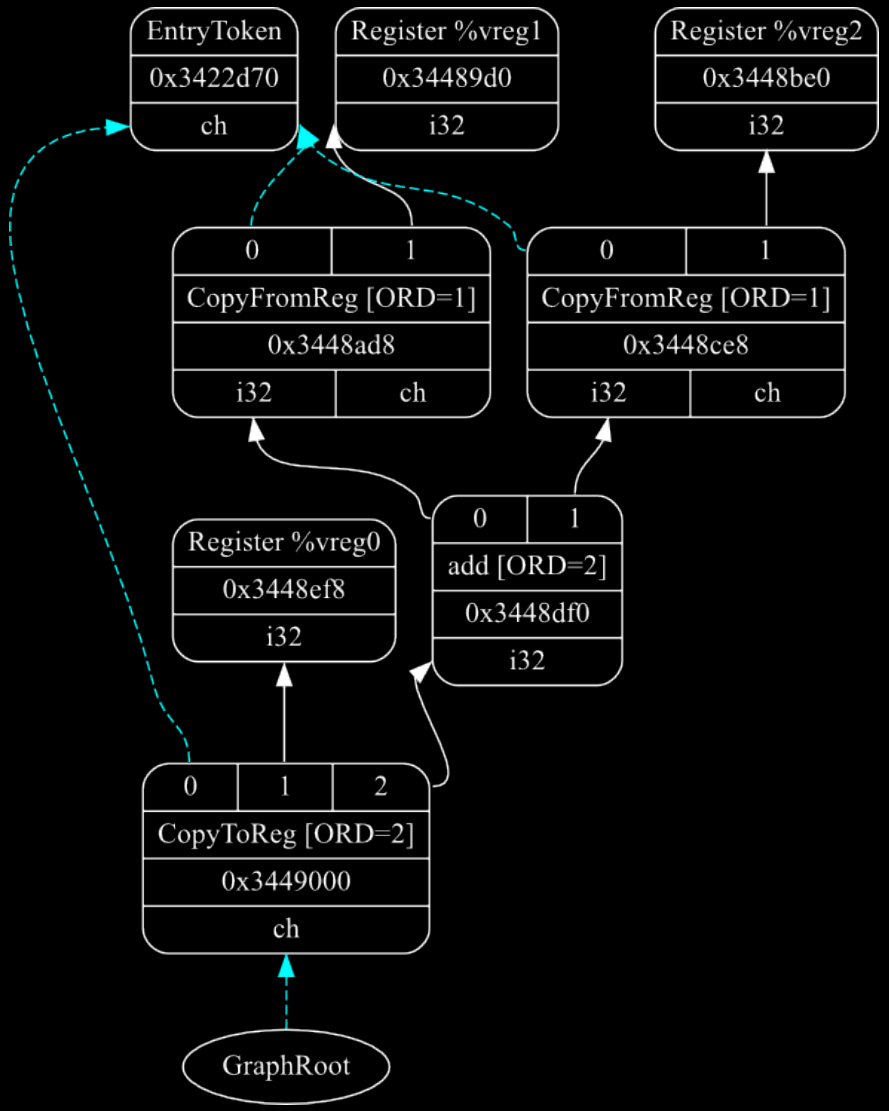
A look at a MachineDAG graph

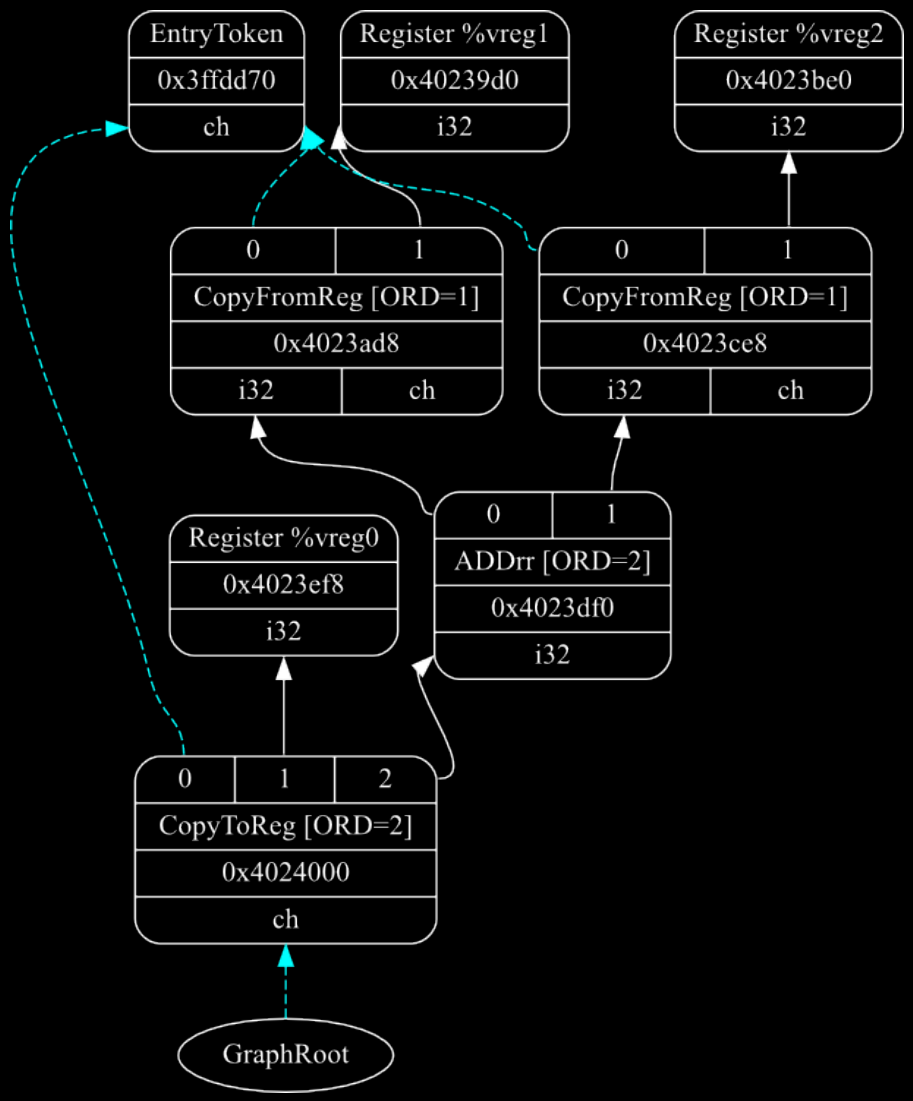
- Very similar to SelectionDAG
- Instructions as nodes
  - Result of instruction selection
  - LEG namespace
- Similar dependencies
- Similar types



IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

Before and after ISel





IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

A look at a MachineInstr block

- Untyped, uses register classes instead
- Target-specific instructions (LEG namespace)
  - Few exceptions (TargetOpcode namespace)

```
BB#0: derived from LLVM BB %entry
    Live Ins: %R0 %R1

   %R0<def> = ADDrr %R0<kill>, %R1<kill>  ; %R0, %R1, %R2: GRRegs

    Successors according to CFG: BB#1
```

ex1-mi.txt

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

Part 2

Creating your own target

Bits of your ISA you need to describe

- Target machine
  - Registers, register classes
  - Calling conventions
- Instruction set
  - Operands and patterns
  - Assembly printing and/or instruction encoding
  - Schedule (not part of this talk)
- …

TableGen

- C++-style syntax
- Different set of backends
  - RegisterInfo, InstrInfo, AsmWriter, ...
- TableGen backends generate .inc files
  - Included by your C++ files
- More information:
  - llvm.org/docs/TableGen/index.html
  - llvm.org/docs/TableGen/BackEnds.html

Describing registers with TableGen

- TableGen provides the 'Register' class
  - Can use the 'HWEncoding' field for encodings

```
class LEGReg<bits<16> Enc, string n> : Register<n> {
  Let HWEncoding = Enc;
  let Namespace = “LEG”;
}

def R0 : LEGReg< 0, “r0” >;
...
def SP : LEGReg< 10, “sp” >;
```

LEGRegisterInfo.td

- Referenced as “LEG::R0” in C++

Describing registers with TableGen

- Can automate trivial definitions

```
foreach i = 0-9 in {
  def R#i : R<i, “r”#i>;
}
```

LEGRegisterInfo.td

- Group registers into register classes

```
def GRRegs : RegisterClass<”LEG”, [i32], 32,
  (add SP, (sequence "R%i", 0, 9))>;
```

LEGRegisterInfo.td

Calling convention lowering: TableGen

```
def CC_LEG : CallingConv<[
    // Promote i8/i16 arguments to i32
    CCIfType<[i8, i16], CCPromoteToType<i32>>,

    // The first 4 arguments are passed in registers
    CCIfType<[i32], CCAssignToReg<[R0, R1, R2, R3]>>,

    // Fall-back, and use the stack
    CCIfType<[i32], CCAssignToStack<4, 4>>
  ]>;
```

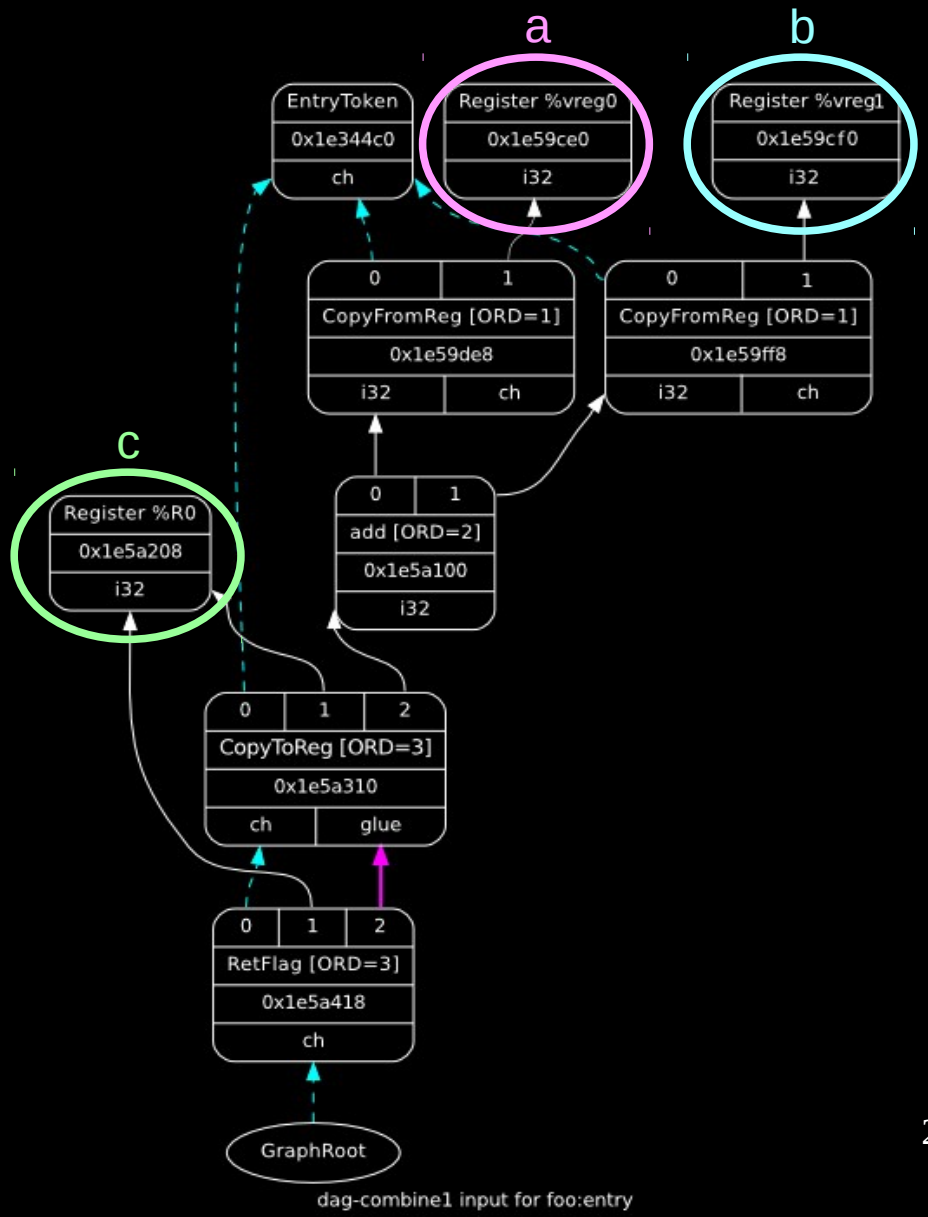
LEGCallingConv.td

- Generates functions used in ISelLowering via function pointers.

Calling convention lowering: The big picture

```
define i32 @foo(i32 %a, i32 %b) {
    %c = add nsw i32 %a, %b
    ret i32 %c
}                                    ex1.ll
```

- Two target hooks:
  - LowerFormalArguments()
  - LowerReturn()



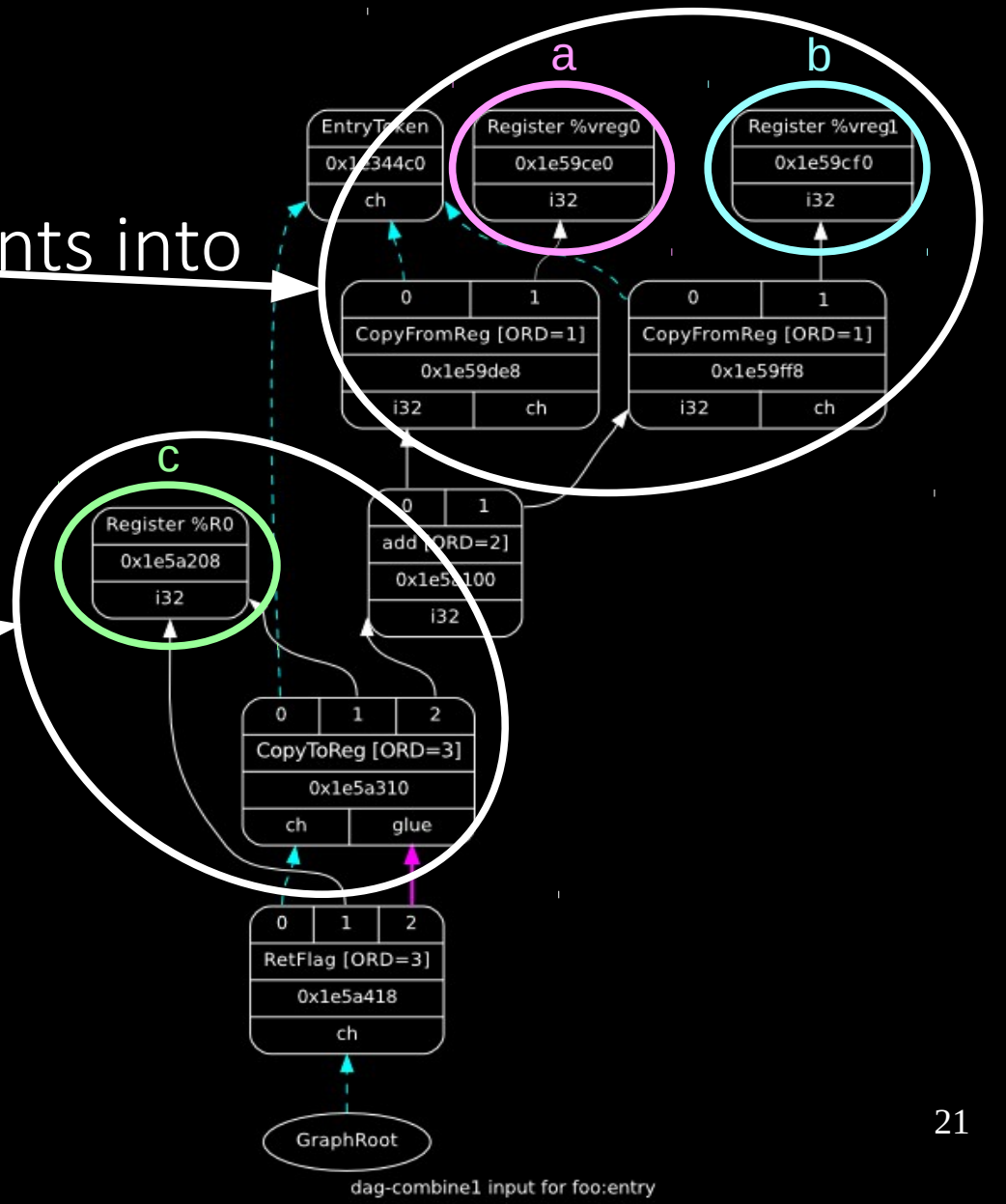
Calling convention lowering: The big picture

LowerFormalArguments()

- Lowers incoming arguments into the DAG

LowerReturn()

- Lowers outgoing return values into the DAG



Calling convention lowering: LowerFormalArguments()

- Assigns locations to arguments, according to the TableGen-defined calling convention
- Creates DAG nodes for each location:
  - Registers: CopyFromReg nodes
  - Stack: frame indices and stack loads

```
// LEGTargetLowering::LowerFormalArguments()
SmallVector<CCValAssign, 16> ArgLocs;
CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(),
               getTargetMachine(), ArgLocs,
               *DAG.getContext());
CCInfo.AnalyzeFormalArguments(Ins, CC_LEG);
...
```

LEGISelLowering.cpp

Calling convention lowering: LowerReturn()

- Similar to LowerFormalArguments(), but the other way around
- Define another, 'RetCC_LEG', TableGen calling convention
- Call 'AnalyzeReturn()' with it
- Walk the return value locations and issue DAG nodes
- Return LEGISD::RET instruction

Calling convention lowering: LowerCall()

- Hybrid of LowerFormalArguments() and LowerReturn()
- Not explicitly covered here

Describing instructions: Overview

- Let's start with a simple instruction: ADDrr
- Define it in lib/Target/LEG/LEGInstrInfo.td
- What we need to specify:
  - Operands
  - Assembly string
  - Instruction pattern

Describing instructions: Operands

- List of definitions or outputs ('outs')
- List of uses or inputs ('ins')
- Operand class:
  - Register class (e.g. GRRegs)
  - Immediate (e.g. i32imm)
  - More complex operands (e.g. reg + imm for load/store)

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2),
                    "add $dst, $src1, $src2",
                    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

LEGInstrInfo.td

Describing instructions: Selection Patterns

- Matches nodes in the SelectionDAG
  - Nodes get turned into MachineInstrs during ISel
  - If pattern is omitted, selection needs to be done in C++
- Syntax:
  - One pair of parenthesis defines one node
  - Nodes have DAG operands, with 'MVT' type (e.g. i32)
  - Map DAG operands to MI operands

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2),
                    "add $dst, $src1, $src2",
                    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

LEGInstrInfo.td

Describing instructions: Result

- This is what we get after defining the pattern:
- Assembly was generated by the instruction printer
  - More on this later

```
# BB#0:                                 # %entry
    add r0, r0, r1
```

ex1.s

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2),
                    "add $dst, $src1, $src2",
                    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

LEGInstrInfo.td

Constants

- Compiling this IR produces the following error:

```
%2 = add nsw i32 %1, 2
```

ex2.ll

```
LLVM ERROR: Cannot select: 0x29d4350: i32 = Constant<2> [ID=2]
In function: main
```

- Specify how to generate ('materialize') constants
- For example, with a 'move' instruction
  - E.g. MOVW on ARM for 16-bit constants

Constants

- Let's define this move instruction:

```
def MOVWi16 : InstLEG<(outs GRRegs:$dst),
                      (ins i32imm:$src),
                      "movw $dst, $src",
                      [(set i32:$dst, i32imm:$src)]> {
  let isMoveImm = 1;
}
```

LEGInstrInfo.td

- The previous example translates to:

```
movw r1, #2
add r0, r0, r1
```

ex2-ADDrr-MOVW.s

Constants

- What if instructions take immediates?

```
def LEGimm8 : Operand<i32>, ImmLeaf<i32, [{
  return Imm >= 0 && Imm < 256;
}]>;

def ADDri : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, i32imm:$src2),
                    "add $dst, $src1, $src2",
                    [(set i32:$dst, (add i32:$src1,
                                         LEGimm8:$src2))]>;
```

LEGInstrInfo.td

```
add  r0, r0, #2
```

ex2-ADDri.s

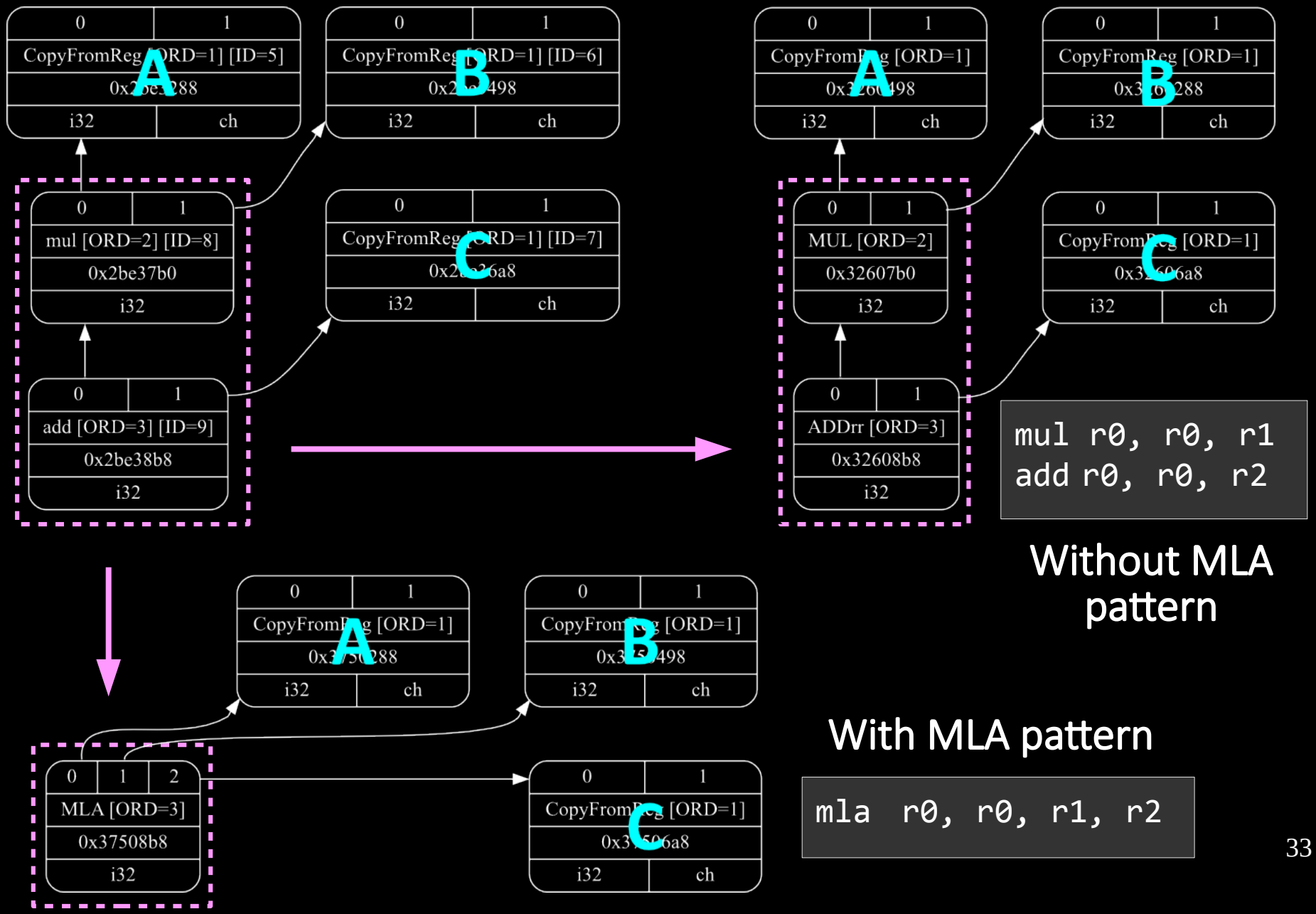
Matching multiple DAG nodes

- DAG nodes can be nested inside selection patterns
  - The output of one node is the input of another
- Allows operations to be combined
  - Reduces the number of generated instructions
  - Possibly improves performance or power consumption
- Example: multiply and add instruction (ex3.ll)

```
def MLA : InstLEG<(outs GRRegs:$dst),
                  (ins GRRegs:$src1, GRRegs:$src2, GRRegs:$src3),
                  "mla $dst, $src1, $src2, $src3",
                  [(set i32:$dst,
                   (add (mul i32:$src1, i32:$src2), i32:$src3))]>;
```

LEGInstrInfo.td

Matching multiple DAG nodes




```
mul r0, r0, r1
add r0, r0, r2
```

Without MLA pattern

With MLA pattern

```
mla  r0, r0, r1, r2
```

Frame lowering

- Hooks invoked when values are stored on the stack
  - In debug builds (-O0), but not only
- Adjust the stack at the beginning ('prologue') and end ('epilogue') of functions
  - LEGFrameLowering::emitPrologue()
  - LEGFrameLowering::emitEpilogue()
- Usually by increasing or decreasing the stack pointer
- May need to align the stack pointer too

Frame lowering

- Example that uses the stack:

```
%p = alloca i32, align 4
store i32 2, i32* %p
%b = load i32* %p, align 4
%c = add nsw i32 %a, %b
```

ex4.ll

- Output when compiled with -O2:

```
sub sp, sp, #4      ; prologue
movw r1, #2
str r1, [sp]
add r0, r0, #2
add sp, sp, #4      ; epilogue
```

ex4-O2.s

Frame lowering

- When compiling with -O0, hooks need to be defined
- Emit load/store instructions with frame indices
- Minimal implementation:

LEGFrameLowering.cpp

```
// LEGInstrInfo::storeRegToStackSlot()
BuildMI(MBB, I, I->getDebugLoc(), get(LEG::STR))
  .addReg(SrcReg, getKillRegState(KillSrc))
  .addFrameIndex(FrameIndex).addImm(0);

// LEGInstrInfo::loadRegFromStackSlot()
BuildMI(MBB, I, I->getDebugLoc(), get(LEG::LDR), DestReg)
  .addFrameIndex(FrameIndex).addImm(0);

// LEGInstrInfo::copyPhysReg()
BuildMI(MBB, I, I->getDebugLoc(), get(LEG::MOVrr), DestReg)
  .addReg(SrcReg, getKillRegState(KillSrc));
```

Instruction printer

- New classes:
  - LEGAsmPrinter
  - LEGMCInstLower
  - An MCAsmStreamer (usually stock)
  - LEGInstPrinter
- LEGAsmPrinter works as a gateway to the streamers
- This stages works with MCInsts, lowered from MachineInstrs by LEGMCInstLower

Instruction printer

- TableGen provides the 'AsmString' field:

```
class InstLEG<... , string asmstr> : Instruction {
  let AsmString = asmstr;
  ...
}                                                    LEGInstrFormats.td
```

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2),
                    “add $dst, $src1, $src2”> {
  ...
}                                                    LEGInstrInfo.td
```

- LEGInstPrinter::printOperand will be called on each operand

Instruction printer

- LEGInstPrinter::printOperand is given the stream to print to.

```
void LEGInstPrinter::printOperand(const MCInst *MI, unsigned No,
                                  raw_ostream &O) {
  const MCOperand &Op = MI->getOperand(No);
  if (Op.isReg()) {
    // TableGen generates this function for us from
    // LEGRegisterInfo.td
    O << getRegisterName(Op.getReg());
    return;
  }
  if (Op.isImm()) {
    O << '#' << Op.getImm();
    return;
  }
  …
}
```

LEGInstPrinter.cpp

Instruction printer

```
    .text
    .file   "foo.ll"
    .globl  foo
    .type   foo,@function
foo:                                    # @foo
# BB#0:                                 # %entry
    add r0, r0, r1
    b lr
.Ltmp0:
    .size   foo, .Ltmp0-foo
```

ex1.s

- That's it!
- Directives & labels handled for us
  - Can emit target-specific syntax if we wish

Part 3

How-to for specific tasks

Instruction encoding

- A few new classes:
  - LEGAsmBackend
  - LEGMCCodeEmitter
  - LEGObjectWriter
  - An MCObjectStreamer (again, stock)
- You will also need your LEGAsmPrinter

Instruction encoding

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2),
                    "add $dst, $src1, $src2",
                    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

LEGInstrInfo.td

- Example encoding:

31 … 28	27 … 25	24 …21	20	19 … 16	15 … 12	11 … 4	3 … 0
1110	000	opcode	0	src1	dst	00000000	src2

- How can we achieve this?

Instruction encoding

- TableGen recognises the 'Inst' field:

```
class InstLEG< ... > : Instruction {
  field bits<32> Inst;
  ...
}                                                    LEGInstrFormats.td
```

- Used to define the binary encoding of each instruction in TableGen:

```
def ADDrr : InstLEG< ... > {
  let Inst{31-25} = 0b110000;
  let Inst{24-21} = 0b1100;       // Opcode
  let Inst{20}    = 0b0;
  let Inst{11-4}  = 0b00000000;
}                                                    LEGInstrInfo.td
```

Instruction encoding

- For operand-based encoding, need bit fields with the same names as the operands:

```
def ADDrr : InstLEG<(outs GRRegs:$dst),
                    (ins GRRegs:$src1, GRRegs:$src2) ... > {
  bits<4> src1; bits<4> src2; bits<4> dst;
  let Inst{31-25} = 0b110000;
  let Inst{24-21} = 0b1100;       // Opcode
  let Inst{20}    = 0b0;
  let Inst{19-16} = src1;         // Operand 1
  let Inst{15-12} = dst;          // Destination
  let Inst{11-4}  = 0b00000000;
  let Inst{3-0}   = src2;         // Operand 2
```

LEGInstrInfo.td

- LEGMCCodeEmitter::getMachineOpValue() will be called on each operand

Instruction encoding

- Returns the encoding of each operand...

```
unsigned LEGMCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand MO,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const {
  if (MO.isReg()) {
    return
      CTX.getRegisterInfo()->getEncodingValue(MO.getReg());
  } if (MO.isImm()) {
    return static_cast<unsigned>(MO.getImm());
  }
  …
}
```

LEGMCCodeEmitter.cpp

...placed, masked, and shifted into position

Relocations and fixups

- For values that require fixing up, record the relocation into Fixups and return zero.

```
unsigned LEGMCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand MO,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const {
  ...

  assert(MO.isExpr()); // MO must be an expression

  const MCExpr *Expr = MO.getExpr();
  const MCExpr::ExprKind Kind = Expr->getFixupKind();

  Fixups.push_back(MCFixup::Create(0, Expr, Kind));
  return 0;
}
```

LEGMCCodeEmitter.cpp

Relocations and fixups

- Defining a target-specific fixup:

LEGFixups.h

```
enum Fixups {
  fixup_leg_pcrel = FirstTargetFixupKind,

  LastTargetFixupKind,
  NumTargetFixupKinds = LastTargetFixupKind – FirstTargetFixupKind
};
```

```
const MCFixupKindInfo& getFixupKindInfo(MCFixupKind K) const {
  const static MCFixupKindInfo I[LEG::NumTargetFixupKinds] = {
    // Name          Offset Size Flags
    { “fixup_leg_pcrel”, 0,  32, MCFixupKindInfo::FKF_IsPCRel },
  };
  ...
  return I[K – FirstTargetFixupKind];
}
```

LEGAsmBackend.cpp

Relocations and fixups

- We must then implement some hooks
- These are called at the end once the section layouts have been finalized
- LEGAsmBackend::processFixupValue()
  - Adjusts the fixup value, e.g., splitting the value across non-contiguous fields
- LEGAsmBackend::applyFixup()
  - Patches the fixed-up value into the binary stream

Custom SelectionDAG nodes

- To represent target-specific operations in the DAG
  - Example: 32-bit immediate move on ARM
- Add a value in the LEGISD enum
- Add to LEGTargetLowering::getTargetNodeName()
- Add TableGen node definition:

```
def MoveImm32Ty : SDTypeProfile<1, 1, [
  SDTCisSameAs<0, 1>, SDTCisInt<0>
]>;

def movei32 : SDNode<"LEGISD::MOVi32", MoveImm32Ty>;
```

LEGOperators.td

Custom DAG lowering

- To implement an operation in a more efficient way
  - E.g. by replacing it with a custom DAG node
- To work around target limitations
  - E.g. by replacing it with multiple DAG nodes
- Add support to LEGISelLowering::LowerOperation
- Set the operation action to 'custom' on given type:

```
LEGISelLowering::LEGISelLowering(...) {
  (...)
  setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
}
```

LEGISelLowering.cpp

Creating SelectionDAG nodes

- Simply call DAG.getNode() with these arguments:
  - Node ID (e.g. LEGISD::MOVi32), type, operand(s)
- Nodes are target-independent (ISD) or not (LEGISD)
- Use DAG.getMachineNode() in LEGISelDAGToDAG

```
SDValue LEGTargetLowering::LowerConstant(SDValue Op,
                                         SelectionDAG &DAG) const {
  EVT VT = Op.getValueType();
  ConstantSDNode *Val = cast<ConstantSDNode>(Op.getNode());
  SDValue TargetVal = DAG.getTargetConstant(Val->getZExtVaue(),
                                            MVT::i32);
  return DAG.getNode(LEGISD::MOVi32, VT, TargetVal);
}
```

LEGISelLowering.cpp

Lowering to multiple instructions

- Example: implement 32-bit immediate load with high/low pairs:
  - MOVW: load 16-bit 'low' immediate and clear high 16 bits
  - MOVT: load 16-bit 'high' immediate
- The final value is defined by the second instruction
- Different ways to implement this:
  - At the DAG level, in LEGISelDAGToDAG.cpp
  - Using a pseudo-instruction (placeholder)

Lowering to multiple instructions

- The two instructions must be ordered:
  - Because MOVW clears the high 16 bits
  - Achieved by having MOVT reading MOVW's output
- Source and destination registers must be the same
  - Use 'Constraints' in Tablegen (unless using pseudo)

```
def MOVTi16 : InstLEG<(outs GRRegs:$dst),
                      (ins GRRegs:$src1, i32imm:$src2),
                      "movw $dst, $src2",
                      [/* No pattern */]> {
  let Constraints = "$src1 = $dst";
}
```

LEGInstrInfo.td

Lowering to multiple instructions

- Define a pseudo instruction, can be selected from the custom DAG node we previously defined
- The pseudo is lowered by a target hook

```
def MOVi32 : InstLEG<(outs GRRegs:$dst), (ins i32imm:$src), “”,
                     [(set i32:$dst, (movei32 i32imm:$src))]> {
  let isPseudo = 1;
}
```

LEGInstrInfo.td

```
bool LEGInstrInfo::expandPostRAPseudo(MI) {
   if (MI->getOpcode() != LEG::MOVi32) return false;
   BuildMI(..., get(LEG::MOVWi16), Dst).addImm(Lo);
   BuildMI(..., get(LEG::MOVTi16), Dst).addReg(Dst).addImm(Hi);
   MBB.erase(MI);
   return true;
}
```

LEGInstrInfo.cpp

Lowering to multiple instructions

- Example IR:

```
define i32 @foo(i32 %a) #0 {
    %c = add nsw i32 %a, 65537 ; 0x00010001
    ret i32 %c
}
```

ex5.ll

```
; We want to write 0x00010001 to the register.
; Each instruction writes half of the value (16 bits)
movw r1, #1  ; Write 0x00000001 (write low bits, clear high bits)
movt r1, #1  ; Write 0x0001XXXX (write high bits, don't touch low bits)
add r0, r0, r1
b lr
```

ex5.s

Part 4

Troubleshooting and resources

When something goes wrong

- Find which pass introduces the issue:
  - llc -print-after-all
- Find the LLVM source file and category for this pass:
  - #define DEBUG_TYPE "codegen-dce"
- Dump the log:
  - llc foo.ll -debug-only codegen-dce 2>&1 > foo.log
- Compare (diff) the '-print-after-all' or '-debug-only' good and bad outputs to see where it goes wrong

Debugging LLVM

- MIs, BBs, functions, almost anything → call X.dump()
- DAGs, CFGs → call X.viewGraph() (pops up xdot)
- Or from the terminal: llc foo.ll-view-isel-dags
  - Try -view-dag1-combine-dags, -view-legalize-dags, -view-sched-dags, etc.
- To view graphs, make sure you build LLVM in debug mode!
  - Turn on LLVM_ENABLE_ASSERTIONS (i.e. NDEBUG should *not* be defined)

“Cannot select”

- When LLVM doesn't know how to map ('lower') a DAG node to an actual instruction
  - Missing pattern in LEGInstrInfo.td?
  - Without a pattern, lowering needs to be done in LEGISelDAGToDag::Select()
- Check the graph!

The dog ate my homework

- Why did my code disappear?
  - Missing chain/glue in the DAG
  - Dead code elimination may have removed it
- DCE does not touch instructions whose value is used by other instructions:
  - Root your use/def chains using a MI that has side-effects
- DCE does not touch instructions with side-effects:
  - mayLoad, mayStore, isTerminator, hasSideEffects

Useful in-tree resources

- include/llvm/Target/Target*.h
  - Shows all target-specific hooks that can be overridden, with useful comments
- include/llvm/Target/Target*.td
  - Shows all TableGen fields you can use in your target files
- include/llvm/CodeGen/ISDOpcodes.h
  - Lists all SelectionDAG nodes and descriptions of what they mean

You're not alone!

- "Writing an LLVM Backend" at llvm.org/docs/WritingAnLLVMBackend.html
- Other backends
- TableGen backend documentation
- LLVM-Dev mailing lists
- Anton Korobeynikov's 2009 and 2012 "Building a backend in 24 hours" tutorials

Summary

- Should be enough to create a very simple target!
- Many things were not covered in this talk:
  - Using different types and legalization
  - Scheduling
  - Intrinsics
  - …
- Introduced resources to go further

Thank you!

- Q & A
- Happy to answer questions by email too:
  - fraser@codeplay.com
  - pierre-andre@codeplay.com
- Check out our code from GitHub:
  - github.com/frasercrmck/llvm-leg