Building an LLVM Backend

LLVM 2014 tutorial

Fraser Cormack
Pierre-André Saulais

Codeplay Software
@codeplaysoft

October 26, 2014
Introduction

• LLVM backend crash course, for beginners
  ▶ How-tos and tips
  ▶ Solution to common problems

• Simple target created for this tutorial
  ▶ Can be used to see how LLVM works
  ▶ Can be used as a skeleton to bootstrap new target
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/codeplaysoftware/llvm-leg
• Slides from this informative and well-presented talk!
Overview

Part 1: Background

Part 2: Creating your own target
  - Describing the target machine
  - Describing the instruction set

Part 3: How-tos for specific tasks
  - Instruction printing
  - Instruction encoding
  - Selection DAG manipulation

Part 4: Troubleshooting and resources
Background
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
int foo(int a, int b) {
    int result = a + b;  // r0 + r1
    return result;      // r0
}
```

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

Fraser Cormack, Pierre-André Saulais
Part 1: Background
LLVM Backend: The big picture

• Pipeline structure
  ▶ Transforms your program many times using different stages
  ▶ Starts target-independent, then gets increasingly target-specific

• Different representations are used
  ▶ Tells you roughly where you are in the pipeline
  ▶ Different instruction namespaces

• Check it out (IR and MI only):
  ▶ `llc foo.ll -print-after-all 2>&1 > foo.log`

IR → SelectionDAG → **MachineDAG** → MachineInstr → MCInst
A look at an IR module

- Linear representation
- High-level, target-agnostic
  - Exceptions: data layout, triple, intrinsics
- Most instructions define values
  - Typed (e.g. i32, float, <4 x i32>)
  - Defined once (SSA), no registers

```
target triple = "leg"

define i32 @foo(i32 %a, i32 %b) {
    %c = add i32 %a, %b
    ret i32 %c
}
```
A look at a SelectionDAG graph

- Graph representation
- Operations as nodes
  - Mostly target-agnostic
    - Semantics defined by LLVM
    - ISD namespace for opcodes
  - Produce typed value(s)
- Dependencies as edges
  - Data
  - Order ("chain")
  - Scheduling ("glue")

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst
A look at a MachineDAG graph

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

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst
Before and after instruction selection

Before:

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

Fraser Cormack, Pierre-André Saulais
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>

Successors according to CFG: BB#1

Kill: last use of a value stored in a register

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst
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)

- ...
Part 2: Creating your own target

- Describing the target machine
- Describing the instruction set
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/index.html)
  - [llvm.org/docs/TableGen/BackEnds.html](llvm.org/docs/TableGen/BackEnds.html)
Describing registers with TableGen

• TableGen provides the ‘Register’ class
  ▶ Can use the ‘HWEncoding’ field for encodings

• Referenced as “LEG::R0” in C++

```plaintext
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
Describing registers with TableGen

• Can automate trivial definitions

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

• Group registers into register classes

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

Fraser Cormack, Pierre-André Saulais
Calling convention lowering: TableGen

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

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

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

- Generates functions used in ISelLowering via function pointers
Calling convention lowering: The big picture

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

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

Fraser Cormack, Pierre-André Saulais
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

```cpp
// 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
Part 2: Creating your own target

- Describing the target machine
- Describing the instruction set
Describing instructions: Overview

• Let’s start with a simple instruction: ADDrr
  ▶ Adds two registers together

• We 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)

```python
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
```

- Result:

```python
%R0<def> = ADDrr %R0<kill>, %R1<kill>
```

Fraser Cormack, Pierre-André Saulais
Part 2: Creating your own target
28 / 72
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

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

• Result:

```assembly
%R0<def> = ADDrr %R0<kill>, %R1<kill>
```
Constants

- Using constants in IR produces errors at this point
- We need to specify how to generate (‘materialize’) constants
- For example, with a ‘move’ instruction
  - E.g. MOVLO for 16-bit constants

**Example:**

```assembly
%c = add i32 %a, 2  
```

**Result:**

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

• Let’s define this move instruction:

```python
def MOVLOi16 : InstLEG<(outs GRRegs:$dst),
    (ins i32imm:$src),
    "movw $dst, $src",
    [(set i32:$dst, i32imm:$src)]> {
    let isMoveImm = 1;
}
```

• Example:

```assembly
%c = add i32 %a, 2
```

• Result:

```assembly
%R1<def> = MOVLOi16 2
%R0<def> = ADDrr %R0<kill>, %R1<kill>
```
Constants

• What if the instruction accepts an immediate operand?

```python
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))]);
```

• Example:

```plaintext
%c = add i32 %a, 2
```

• Result:

```plaintext
%R0<def> = ADDri %R0<kill>, 2
```
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)

```python
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))]>;
```

Fraser Cormack, Pierre-André Saulais
Matching multiple DAG nodes

With MLA pattern:

With MLA pattern:

Without MLA pattern:
Frame lowering

• Hooks invoked when values are stored on the stack
  ▶ In debug builds (-O0), but not only

• Used to reserve space in the stack
  ▶ Usually by increasing or decreasing the stack pointer (SP)
  ▶ May need to align the stack pointer too

• Generate code at the beginning (‘prologue’) and end (‘epilogue’) of functions
  ▶ LEGFrameLowering::emitPrologue()
  ▶ LEGFrameLowering::emitEpilogue()
Frame lowering

• Example that uses the stack:

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

• Result when compiled with -O2:

```assembly
%SP<def> = SUBri %SP, 4 ; Prologue
%R1<def> = MOVLOi16 2
STR %R1<kill>, %SP, 0; mem:ST4[%p]
%R0<def> = ADDri %R0<kill>, 2
%SP<def> = ADDri %SP, 4 ; Epilogue
```

ex4/ex4.ll

ex4/ex4-O2-mi.txt
Frame lowering

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

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

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

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

LEGInstrInfo.cpp

Fraser Cormack, Pierre-André Saulais
Part 2: Creating your own target
Part 3

How-tos for specific tasks
Part 3: How-tos for specific tasks

- Instruction printing
- Instruction encoding
- Selection DAG manipulation
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:

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

```cpp
def ADDrr : InstLEG<(outs GRRegs:$dst),
    (ins GRRegs:$src1, GRRegs:$src2),
    "add $dst, $src1, $src2"> {
    ...
}
```

- `LEGInstPrinter::printOperand()` will be called on each operand.
Instruction printer

• LEGInstPrinter::printOperand() prints the assembly string of a given operand...

```cpp
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;
    }
    /* ... */
}
```

• ...and is given the stream to print it to
Instruction printer

• That’s it!
• Directives and labels handled for us
  ▶ Can emit target-specific syntax if we wish

```assembler
.text
.file "ex1.ll"
.globl foo
.type foo,@function
foo:                           # @foo
  # BB#0:                  # %entry
  add r0, r0, r1
# BB#1:                  # %exit
  bx lr
.Ltmp0:                      ex1/ex1.s
```
Part 3: How-tos for specific tasks

- Instruction printing
- Instruction encoding
- Selection DAG manipulation
Instruction encoding

• A few new classes:
  ▶ An MCObjectStreamer (again, stock)
  ▶ LEGMCCodeEmitter
  ▶ LEGObjectWriter
  ▶ LEGAsmBackend

• You will also need your LEGAsmPrinter
Instruction encoding

Example encoding:

<table>
<thead>
<tr>
<th>31...28</th>
<th>27...25</th>
<th>24...21</th>
<th>20</th>
<th>19...16</th>
<th>15...12</th>
<th>11...4</th>
<th>3...0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>000</td>
<td>opcode</td>
<td>0</td>
<td>src1</td>
<td>dst</td>
<td>00000000</td>
<td>src2</td>
</tr>
</tbody>
</table>

How can we achieve this?

```
%R0<def> = ADD%R0<def>, %R1<kill>
```

<table>
<thead>
<tr>
<th>31...28</th>
<th>27...25</th>
<th>24...21</th>
<th>20</th>
<th>19...16</th>
<th>15...12</th>
<th>11...4</th>
<th>3...0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>000</td>
<td>1100</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>00000000</td>
<td>0001</td>
</tr>
</tbody>
</table>
Instruction encoding

• TableGen recognises the ‘Inst’ field:

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

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

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

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

```python
def ADDrr : InstLeg<(outs GRRegs:$dst),
            (ins GRRegs:$src1, GRRegs:$src2) ... > {
    bits<4> src1; bits<4> src2; bits<4> dst;
    let Inst{31-25} = 0b1100000;
    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
```

• `LEGMCCodeEmitter::getMachineOpValue()` will be called on each operand
Instruction encoding

- `LEGMCCodeEmitter::getMachineOpValue()` returns the binary encoding of a given operand...

```cpp
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());
    }
    /* ... */
}
```

- ...which is placed, masked, and shifted into position by TableGen-generated code
Relocations and fixups

- For values that need fixing up, record the relocation and return zero.
- LLVM will keep track of the relocation for us and help us fix it up later.

```cpp
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;
}
```
Relocations and fixups

- Defining a target-specific fixup:

```cpp
enum Fixups {
    fixup_leg_mov_hi16_pcrel = FirstTargetFixupKind,
    fixup_leg_mov_lo16_pcrel,
    LastTargetFixupKind,
    NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
};
```

```cpp
const MCFixupKindInfo& getFixupKindInfo(MCFixupKind K) const {
    const static MCFixupKindInfo I[LEG::NumTargetFixupKinds] = {
        // Name          Offset Size Flags
        { "fixup_leg_mov_hi16_pcrel", 0, 32, MCFixupKindInfo::FKF_IsPCRel },
        { "fixup_leg_mov_lo16_pcrel", 0, 32, MCFixupKindInfo::FKF_IsPCRel },
    };
    /* ... */
    return I[K - FirstTargetFixupKind];
}
```
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
Part 3: How-tos for specific tasks

- Instruction printing
- Instruction encoding
- Selection DAG manipulation
Custom Selection DAG nodes

- To represent target-specific operations in the DAG
  - Example: 32-bit immediate move

- How?
  - Add a value in the LEGISD enum
  - Update LEGTargetLowering::getTargetnodeName()
  - Add TableGen node definitions
    - Type definition: number of inputs, outputs, constraints
    - Node definition: tablegen name, opcode, type

- Custom nodes can be used in TableGen selection patterns

```
def MoveImm32Ty : SDTypeProfile<1, 1, [SDTCisSameAs<0, 1>, SDTCisInt<0>>];
def movei32 : SDNode<"LEGISD::MOVi32", MoveImm32Ty>;  
```
Custom DAG lowering

- To handle DAG nodes in a special way
  - Replaces an existing node with one or more other DAG nodes
  - Matches nodes by opcode (e.g. ISD::Constant)
  - Matches nodes by type (e.g. i32)

- How?
  - Call `setOperationAction(nodeOpcode, type, Custom)`
  - Create a function to handle it (e.g. `LowerOPCODE`)
  - Update `LowerOperation` to call `LowerOPCODE`

- This all happens in `LEGTargetLowering` (`LEGISelLowering.cpp`)
Custom DAG lowering: LowerOPCODE

- LowerOPCODE takes a DAG node (Op) and returns a DAG node
- You can:
  - Return a different node
  - Return Op → no change
  - Return SDValue() → node not supported (LLVM will expand it)
- All of the above can be done conditionally (e.g. depending on VT)

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

• Simply call `DAG.getNode()` with these arguments:
  ▶ Node opcode (e.g. `LEGISD::MOVi32`), type, operand(s)

• Nodes are target-independent (ISD) or not (LEGISD)

• Use `DAG.getMachineNode()` in `LEGISelDAGToDAG`

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

`LEGISelLowering.cpp`
Lowering to multiple instructions

- Example: 32-bit immediate load
  - MOVLO: loads 16-bit ’low’ part
  - MOVHI: loads 16-bit ’high’ part
- The two instructions must be ordered
  - MOVLO clears the ’high’ part
  - MOVHI, MOVLO gives the wrong result
  - Make MOVHI read the output of MOVLO
- Example: 0x00010002
Lowering to multiple instructions

- To define MOVHI, we need an extra operand (‘fakesrc’)
  - Source and destination registers must be the same
  - Use ‘Constraints’ in Tablegen

```python
def MOVHIi16 : InstLEG<(outs GRRegs:$dst),
    (ins GRRegs:$fakesrc, i32imm:$src),
    "movt $dst, $src",
    /* No pattern */> {
    let Constraints = "$fakesrc = $dst";
}
```
Lowering to multiple instructions

- Different ways to emit multiple instructions from one DAG node
  - Using custom C++ instruction selection code
    - Not covered here
  - Using a pseudo-instruction as a placeholder
Lowering to multiple instructions

• Using a pseudo instruction
  ▶ Behaves like a placeholder for ’real’ machine instruction(s)
  ▶ Lowered by a target hook into these instruction instructions
  ▶ Can be selected from the custom DAG node we previously defined

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

LEGInstrInfo.td
Lowering to multiple instructions

- The pseudo is lowered by a target hook

```cpp
bool LEGInstrInfo::expandPostRAPseudo(MachineInstr *MI) {
  if (MI->getOpcode() != LEG::MOVi32)
    return false;

  DebugLoc DL = MI->getDebugLoc();
  MachineBasicBlock &MBB = *MI->getParent();
  unsigned Dst = MI->getOperand(0).getReg();
  unsigned Imm = MI->getOperand(1).getImm();
  unsigned Lo16 = Imm & 0xffff;
  unsigned Hi16 = (Imm >> 16) & 0xffff;

  BuildMI(MBB, MI, DL, get(LEG::MOVLOi16), Dst).addImm(Lo16);
  BuildMI(MBB, MI, DL, get(LEG::MOVHIi16), Dst).addReg(Dst).addImm(Hi16);
  MBB.erase(MI);

  return true;
}
```
Lowering to multiple instructions

Example IR:

```assembly
define i32 @foo(i32 %a) #0 {
    %c = add i32 %a, 65538
    ret i32 %c
}
```

Resulting assembly:

```assembly
; Write 0x00010002 to r1.
movw r1, #2 ; Write 0x00000002
movt r1, #1 ; Write 0x0001XXXX
add r0, r0, r1
bx lr
```

Fraser Cormack, Pierre-André Saulais
Part 3: How-tos for specific tasks
Troubleshooting and resources
When something goes wrong

- Find which pass introduces the issue:
  - `llc -print-after-all`

- Dump the detailed output for the pass:
  - `llc foo.ll -debug-only codegen-dce 2>&1 > foo.log`

- Check the pass’s LLVM source file for the debug type:
  - `#define DEBUG_TYPE "codegen-dce"`

- Compare the `-print-after-all` or `-debug-only` outputs with and without the ‘problem’ change
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"

- LLVM doesn’t know how to map (‘lower’) a DAG node to an actual instruction
  - Missing pattern in LEGInstrInfo.td?
- Check the graph - verify that the following match up:
  - Number of operands
  - Order of operands
  - Types of operands
The dog ate my homework

- Why did my code disappear?
  - Dead Code Elimination may have removed it
  - Missing chain or glue constraints in the DAG
- 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:
  - TableGen attributes: mayLoad, mayStore, hasSideEffects...
Useful in-tree resources

• include/llvm/Target/Target*.h
  ▶ All target-specific hooks that can be overridden
  ▶ Check the Doxygen at: http://llvm.org/doxygen/

• include/llvm/Target/Target*.td
  ▶ All TableGen classes & fields that you can use in your target files

• include/llvm/CodeGen/ISDOpcodes.h
  ▶ All target-independent SelectionDAG nodes and their semantics
You’re not alone!

- "Writing an LLVM Backend" at: llvm.org/docs/WritingAnLLVMBackend.html
- Other backends
- 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/codeplaysoftware/llvm-leg