

# Building an LLVM Backend

Fraser Cormack  
Pierre-André Saulais

Codeplay Software  
@codeplaysoft

March 26, 2015

# Codeplay

- Based here in Edinburgh
- 31 exceedingly talented developers
- Some of the things we get up to:
  - ▶ Compilers (customers' & proprietary)
  - ▶ Debuggers & profilers
  - ▶ Collaborative R&D
  - ▶ Academic collaborations
  - ▶ Test-suites & test infrastructure
  - ▶ Helping define new standards
- A lot of our work uses LLVM

# Partners and Customers



QUALCOMM



FIXSTARS



EUTECHNYX



ARM



Think Silicon



Movidius



Imagination



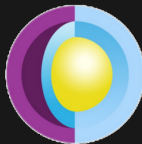
Geomerics



real eyes



BROADCOM



THE UNIVERSITY of EDINBURGH



epcc



AiGameDev.com

# Introduction

- LLVM backend crash course, for beginners
  - ▶ Introduction to the different representations used in the backend
  - ▶ How-tos for the most common backend 'features'
- Simple target created for this tutorial
  - ▶ Can be used to see how LLVM works
  - ▶ Can be used as a skeleton to bootstrap a new target

# What you need to start

- Know a little bit about LLVM IR:  
[llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html)
- xdot.py to visualize graphs when debugging:  
[github.com/jrfonseca/xdot.py](https://github.com/jrfonseca/xdot.py)
- Check out and build our LLVM repo from GitHub:  
[github.com/codeplaysoftware/llvm-leg](https://github.com/codeplaysoftware/llvm-leg)
- Slides from this informative and well-presented talk!

# Overview

## Part 1: Background

- What & why: LLVM
- Our backend: LEG
- LLVM Backend: The big picture

## 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: Conclusion

# Part 1

## Background

# Part 1: Background

- What & why: LLVM
- Our backend: LEG
- LLVM Backend: The big picture



# What is LLVM?

- Collection of modular compiler and toolchain technologies
- Started as a research project
- Apple took it on in 2005
- Grown to include many sub-projects:
  - ▶ LLVM Core - Optimizers, code generators
  - ▶ Clang - C/C++/Objective-C front-end
  - ▶ LLDB - Debugger
  - ▶ libc++ - Conformant & performant C++ STL
  - ▶ lld - System linker

# Why LLVM?

- Open source
- Nice, permissive license
- Modular
- Active community

# Part 1: Background

- What & why: LLVM
- Our backend: LEG
- LLVM Backend: The big picture

# 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
  - ▶ 32-bit load/store
  - ▶ integer comparison, branch, branch and link

# LEG assembly examples

- Function that adds two integers
  - ▶ The arguments are passed in registers: a in r0, b in r1
  - ▶ The return value is stored in r0
- Returns to the caller using b
  - ▶ lr contains the function's return address

```
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

# LEG assembly examples

- Function that loads a value, adds a constant and stores the result
  - ▶ `ldr` loads a value from memory at the given address
  - ▶ `mov` copies an immediate value to the register
  - ▶ `str` stores a value to memory at the given address

```
void ex7(int *src, int *dst) {  
    int x = src[0];  
    int result = x + 42;  
    dst[0] = result;  
}
```

`ex7.c`

```
ex7:  
    ldr r0, [r0]  
    mov r2, #42  
    add r0, r0, r2  
    str r0, [r1]  
    bx lr
```

`ex7.s`

# LEG assembly examples

- Function that either adds or subtracts two integers
- The result depends on which argument is greater
  - ▶ `cmp` compares the arguments
  - ▶ `b` branches to the `'if_else'` basic block, using:
    - ▶ The `'le'` predicate
    - ▶ The implicit condition flag register set by `cmp`
  - ▶ Fall-through to `'if_then'` if the predicate doesn't match

```
int ex6(int a, int b) {  
    if (a > b) {  
        return a + b;  
    } else {  
        return a - b;  
    }  
}
```

ex6.c

```
.ex6:  
    cmp r0, r1  
    ble .if_else  
.if_then:  
    add r0, r0, r1  
    bx lr  
.if_else:  
    sub r0, r0, r1  
    bx lr
```

ex6.s

# Part 1: Background

- What & why: LLVM
- Our backend: LEG
- LLVM Backend: The big picture



# 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 representations only):
  - ▶ `llc foo.ll -print-after-all 2>&1 > foo.log`

IR → SelectionDAG → MachineDAG → MachineInstr → MCIInst

# A look at an IR module

- Linear representation
  - ▶ Basic blocks are lists of instructions
  - ▶ Control Flow Graph (CFG) of basic blocks
- 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 datalayout = "e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32-f64-..."
target triple = "leg"
```

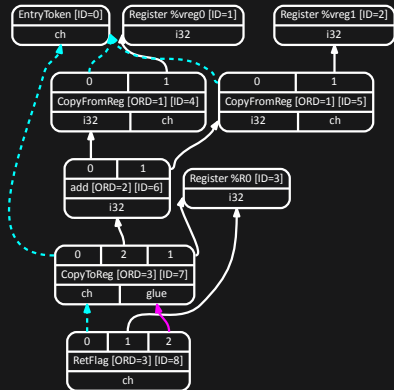
```
define i32 @foo(i32 %a, i32 %b) {
    %c = add i32 %a, %b
    ret i32 %c
}
```

ex1b.ll

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

# A look at a SelectionDAG graph

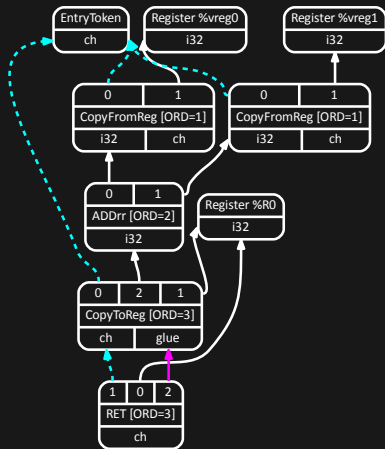
- Graph representation
- Operations as nodes
  - ▶ Mostly target-agnostic
    - ▶ Semantics defined by LLVM
    - ▶ ISD namespace for opcodes
  - ▶ Produce typed value(s)
- Dependencies as edges
  - ▶ Data
  - ▶ Order ("chain")
  - ▶ Scheduling ("glue")



IR → SelectionDAG → MachineDAG → MachineInstr → MCIInst

# A look at a MachineDAG graph

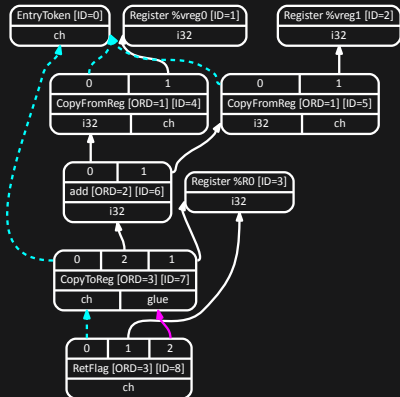
- Very similar to SelectionDAG
- Target instructions as nodes
  - ▶ Result of instruction selection
  - ▶ LEG namespace
- Similar dependencies
- Similar types



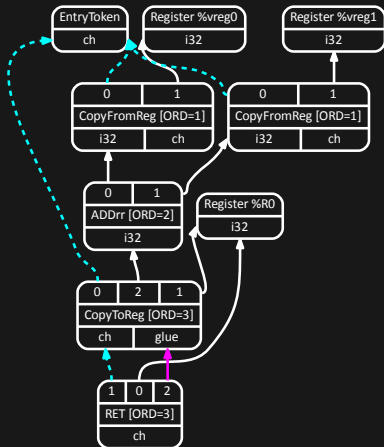
IR → SelectionDAG → MachineDAG → MachineInstr → MCIInst

# Before and after instruction selection

Before:



After:



IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

# A look at a MachineInstr block

- Untyped, uses register classes instead
  - ▶ Virtual registers (before register allocation)
  - ▶ Machine registers (after RA)
- Target-specific instructions (LEG namespace)
  - ▶ Few exceptions (TargetOpcode namespace)
- Instruction operands have flags:
  - ▶ Def for outputs, use for inputs
  - ▶ Kill: last use of a value stored in a register

```
BB#0: derived from LLVM BB %entry
```

```
Live Ins: %R0 %R1
```

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

```
Successors according to CFG: BB#1
```

[ex1/ex1-mi.txt](#)

IR → SelectionDAG → MachineDAG → **MachineInstr** → MCIInst

# 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)
- ...



## 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](http://llvm.org/docs/TableGen/index.html)
  - ▶ [llvm.org/docs/TableGen/BackEnds.html](http://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++

```
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

```
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](#)

## 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)

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

- Result:

```
%R0<def> = ADDrr %R0<kill>, %R1<kill> ex1/ex1-mi.txt
```

# 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:$dst), LEGInstrInfo.td
                (ins GRRegs:$src1, GRRegs:$src2),
                "add $dst, $src1, $src2",
                [(set i32:$dst, (add i32:$src1, i32:$src2))];
```

- Result:

```
%R0<def> = ADDrr %R0<kill>, %R1<kill> ex1/ex1-mi.txt
```



# 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:

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

[ex2/ex2.ll](#)

- Result:

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

# Constants

- Let's define this move instruction:

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

[LEGInstrInfo.td](#)

- Example:

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

[ex2/ex2.ll](#)

- Result:

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

[ex2/ex2-ADDrr-MOVLO-mi.txt](#)

# Constants

- What if the instruction accepts an immediate operand?

```
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](#)

- Example:

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

[ex2/ex2.ll](#)

- Result:

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

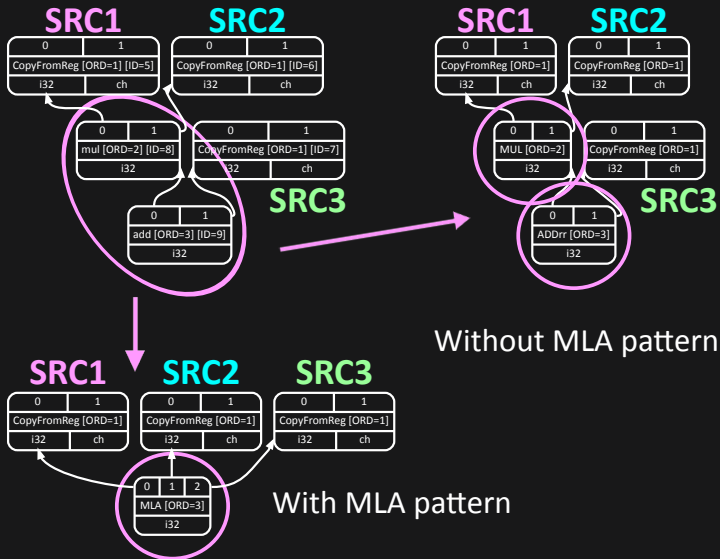
[ex2/ex2-ADDri-mi.txt](#)

# 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:$dst), LEGInstrInfo.td
    (ins GRRegs:$src1, GRRegs:$src2, GRRegs:$src3),
    "mla $dst, $src1, $src2, $src3",
    [(set i32:$dst,
        (add (mul i32:$src1, i32:$src2), i32:$src3))]>;
```

# Matching multiple DAG nodes



# 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:

```
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](#)

- `LEGINstrPrinter::printOperand()` will be called on each operand.

# Instruction printer

- `LEGIstPrinter::printOperand()` prints the assembly string of a given operand...

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

LEGIstPrinter.cpp

- ...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

```
.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 MCOjectStreamer (again, stock)
  - ▶ LEGMCCodeEmitter
  - ▶ LEGObjectWriter
  - ▶ LEGAsmBackend
- You will also need your LEGAsmPrinter

# Instruction encoding

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?

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

31...28	27...25	24...21	20	19...16	15...12	11...4	3...0
1110	000	1100	0	0000	0000	00000000	0001

# 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

- `LEGMCCodeEmitter::getMachineOpValue()` returns the binary encoding of a given 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](#)

- ...which is placed, masked, and shifted into position by TableGen-erated 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

```
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:

```
enum Fixups {
    fixup_leg_mov_hi16_pcrel = FirstTargetFixupKind,
    fixup_leg_mov_lo16_pcrel,

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

LEGFixups.h

```
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];
}
```

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

# Part 3: How-tos for specific tasks

- Instruction printing
- Instruction encoding
- Selection DAG manipulation

# Custom SelectionDAG 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>;
```

[LEGOptimizers.td](https://llvm.org/docs/LEGOptimizers.html)

# 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)

```
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



# Creating SelectionDAG nodes

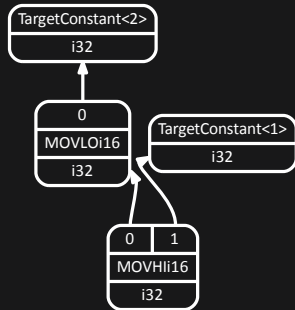
- Simply call `DAG.getNode()` with these arguments:
  - ▶ Node opcode (e.g. `LEGISD::MOVi32`), type, operand(s)
- Nodes are target-independent (ISD) or not (LEGISD)
- Use `DAG.getMachineNode()` in `LEGISelDAGToDAG`

```
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: 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

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

[LEGINstrInfo.td](#)

# 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

```
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

```
bool LEGInstrInfo::expandPostRAPpseudo(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;
}
```

LEGInstrInfo.cpp

# Lowering to multiple instructions

Example IR:

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

ex5/ex5.ll

Resulting assembly:

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

ex5/ex5.s

# Part 4

## Conclusion



# You're not alone!

- "Writing an LLVM Backend" at:  
[llvm.org/docs/WritingAnLLVMBackend.html](http://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
  - ▶ Multiple basic blocks, branches
  - ▶ Scheduling
  - ▶ Intrinsic
  - ▶ ...
- Introduced resources to go further

# Thank you!

- Q&A
- Happy to answer questions by email too:
  - ▶ [fraser@codeplay.com](mailto:fraser@codeplay.com)
  - ▶ [pierre-andre@codeplay.com](mailto:pierre-andre@codeplay.com)
- Check out our code from GitHub:
  - ▶ [github.com/codeplaysoftware/llvm-leg](https://github.com/codeplaysoftware/llvm-leg)