LLVM backend development by example (RISC-V)

Alex Bradbury asb@lowrisc.org @asbradbury

17th October 2018

About this tutorial

- Can’t cover everything, hope to cover a useful “slice”
- Go into detail, but not minutiae (read the code for that)
- Give you a starting point to go further:
  - High level overview
  - Deep-dive into an example
  - Where to go for more info
  - What to do when things don’t work first time

Follow-up by coming to tomorrow’s Coding Lab (2pm-3.30pm tomorrow)

RISC-V background

- RISC-V is an instruction set architecture (ISA) developed as an extensible open standard
- Has a range of open source and proprietary implementations
- Has 32-bit, 64-bit and 128-bit base instruction sets
- Base integer instruction set contains <50 instructions. Standard extensions are referred to with a single letter, e.g. ‘M’ adding multiply/divide, ‘F’ for single-precision floating point. ISA variants are referred to with a compact string, e.g. RV32IMAC
- Vendors are free to introduce their own custom instruction set extensions
- See http://www.riscv.org
- Be sure to check the RISC-V themed posters in the poster session tomorrow (MC layer fuzzing, support for the compressed instruction set).

Compilation flow (simplified)

Codegen:

.c -> LLVM IR -> SelectionDAG -> MachineInstr -> MCInst -> .o

Assembler:

.s -> MCInst -> .o

Our approach: start with the common requirement, the ability to encode MCInst into an output ELF.

MC layer: plan of attack

- How to describe an instruction’s encoding and assembly syntax (TableGen)
- Describing registers and other operands
- Assembly parsing
- Necessary infrastructure
- Testing
- Where to go for more info
- Debugging / problem solving

Describing an instruction: ADD

Use the TableGen domain-specific language.

See lib/Target/RISCV/RISCVInstrInfo.td

```
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add\t$rd, $rs1, $rs2";
}
```

Describing an instruction: ADD

Encoding

```
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add\t$rd, $rs1, $rs2";
}
```

Describing an instruction: ADD

Assembly parsing / printing

```
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add\t$rd, $rs1, $rs2";
}
```

Describing an instruction: ADD

Introducing classes to reduce duplication across instructions.

```
class RVInstR<bits<7> funct7, bits<3> funct3,
              RISCVOpcode opcode, dag outs,
              dag ins, string opcodestr,
              string argstr>
    : RVInst<outs, ins, opcodestr,
             argstr, [], InstFormatR> {
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;

    let Inst{31-25} = funct7;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = funct3;
    let Inst{11-7} = rd;
    let Opcode = opcode.Value;
}
```

Describing an instruction: ADD

Introducing classes to reduce duplication across instructions and using these to describe similar instructions.

```
class ALU_rr<bits<7> funct7, bits<3> funct3,
             string opcodestr>
    : RVInstR<funct7, funct3, OPC_OP,
              (outs GPR:$rd),
              (ins GPR:$rs1, GPR:$rs2),
              opcodestr, "$rd, $rs1, $rs2">;

def ADD  : ALU_rr<0b0000000, 0b000, "add">;
def SUB  : ALU_rr<0b0100000, 0b000, "sub">;
def SLL  : ALU_rr<0b0000000, 0b001, "sll">;
def SLT  : ALU_rr<0b0000000, 0b010, "slt">;
def SLTU : ALU_rr<0b0000000, 0b011, "sltu">;
def XOR  : ALU_rr<0b0000000, 0b100, "xor">;
def SRL  : ALU_rr<0b0000000, 0b101, "srl">;
def SRA  : ALU_rr<0b0100000, 0b101, "sra">;
def OR   : ALU_rr<0b0000000, 0b110, "or">;
def AND  : ALU_rr<0b0000000, 0b111, "and">;
```

Describing an instruction: ADDI

Similar to before.

Next: What exactly are ‘simm12’ and ‘GPR’? How do they ensure illegal input is rejected?

```
def ADDI : RVInstI<0b000, OPC_OP_IMM,
                   (outs GPR:$rd),
                   (ins GPR:$rs1, simm12:$imm12),
                   "addi", "$rd, $rs1, $imm12">;
```

Describing registers

1) Define registers, their encoding, and their assembly names
2) Put them in a RegisterClass

NB: The RISC-V backend actually uses register classes parameterised by GPR length (XLEN).

```
class RISCVReg<bits<5> Enc, string n, list<string>
alt = []> : Register<n> {
  let HWEncoding{4-0} = Enc;
  let AltNames = alt;
  let Namespace = “RISCV”;
}

let RegAltNameIndices = [ABIRegAltName] in {
def X0
  : RISCVReg<0, "x0", ["zero"]>, DwarfRegNum<[0]>;
def X1
  : RISCVReg<1, "x1", ["ra"]>, DwarfRegNum<[1]>;
 [...] // omitted for brevity
}

def GPR : RegisterClass<"RISCV", [i32], 32, (add
  (sequence "X%u_32", 0, 31)
)>;
```

Immediate operands

The associated ParserMatchClass specifies how this immediate type hooks in to the assembly parser for validation, error reporting etc.

```
class SImmAsmOperand<int width> : AsmOperandClass {
  let Name = "SImm" # width;
  let RenderMethod = "addImmOperands";
  let DiagnosticType = !strconcat("Invalid", Name);
}

def simm12 : Operand<XLenVT> {
  let ParserMatchClass = SImmAsmOperand<12>;
  let EncoderMethod = "getImmOpValue";
  let DecoderMethod = "decodeSImmOperand<12>";
}
```

Implementing RISCVAsmParser

- Generated methods will do a lot of the work for us: MatchRegisterName, MatchRegisterAltName, MatchInstructionImpl
- Unlike most of LLVM, false typically indicates success
- You must implement:
    - RISCVOperand which represents a parsed token, register or immediate and contains methods for validating it (e.g. isSImm12)
    - The top-level MatchAndEmitInstruction which mostly calls MatchInstructionImpl, but you must provide diagnostic handling
    - ParseInstruction, ParseRegister

Code example: RISCVAsmParser ::ParseInstruction

Create `RISCVOperands` while parsing.

`RISCVOperand` contains methods such as `isSimm12()`.

Beware: false signals success (LLVM parser convention)

```
bool RISCVAsmParser::ParseInstruction(ParseInstructionInfo &Info,
                                      StringRef Name, SMLoc NameLoc,
                                      OperandVector &Operands) {
  Operands.push_back(RISCVOperand::createToken(Name, NameLoc, isRV64()));

  if (getLexer().is(AsmToken::EndOfStatement))
    return false;

  if (parseOperand(Operands, Name))
    return true;

  // Parse until end of statement, consuming commas between operands
  unsigned OperandIdx = 1;
  while (getLexer().is(AsmToken::Comma)) {
    getLexer().Lex();

    if (parseOperand(Operands, Name))
      return true;

    ++OperandIdx;
  }

  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    SMLoc Loc = getLexer().getLoc();
    getParser().eatToEndOfStatement();
    return Error(Loc, "unexpected token");
  }

  getParser().Lex(); // Consume the EndOfStatement.
  return false;
}
```

Hooking it all up: needed infrastructure

- Directory structure
  - lib/Target/RISCV
- Build system: CMakeLists.txt, LLVMBuild.txt
- Target registration
- Triple parsing
- Architecture-specific definitions, e.g. reloc numbers
- RISCVMCAsmInfo (details such as comment delimiter)
- RISCVAsmBackend and RISCVELFObjectWriter (mostly fixup/reloc handling so stubbed out for now),
- RISCVMCCodeEmitter (produces encoded instructions for an MCInst, but tablegenerated getBinaryCodeForInstr does most of the work)
- Test infrastructure: using lit and FileCheck

Testing the MC layer

- FileCheck: Checks for expected patterns in test output
- lit: LLVM test runner
- See test/MC/RISCV/*
- This test checks round trip .s -> .o -> .s
- Also want to test invalid inputs are rejected and sensible diagnostics generated

Augment hand-written tests with automated fuzzing.

```
# RUN: llvm-mc %s -triple=riscv32 -riscv-no-aliases -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
# RUN: llvm-mc -filetype=obj -triple=riscv32 < %s \
# RUN:     | llvm-objdump -riscv-no-aliases -d -r - \
# RUN:     | FileCheck -check-prefixes=CHECK-OBJ,CHECK-ASM-AND-OBJ %s


# CHECK-ASM-AND-OBJ: addi ra, sp, 2
# CHECK-ASM: encoding: [0x93,0x00,0x21,0x00]
addi ra, sp, 2
# CHECK-ASM: addi ra, sp, %lo(foo)
# CHECK-ASM: encoding: [0x93,0x00,0bAAAA0001,A]
# CHECK-OBJ: addi ra, sp, 0
# CHECK-OBJ: R_RISCV_LO12
addi ra, sp, %lo(foo)
# CHECK-ASM-AND-OBJ: slti a0, a2, -20
# CHECK-ASM: encoding: [0x13,0x25,0xc6,0xfe]
slti a0, a2, -20
```

Where to go for more info

- llvm.org/docs
- LLVM mailing list
- riscv-llvm patchset (in-tree or github.com/lowrisc/riscv-llvm)
    - Useful especially for topics we missed, e.g. relocations+fixups
- llvmweekly.org
- **Read code**, especially other backends with similar properties
- Reading parent classes often gives useful insight
- Commit logs, git blame
- include/llvm/Target/Target.td

Delving deeper into the RISC-V MC layer and TableGen

- Study include/llvm/Target/Target.td
- View all records generated from TableGen:
  - ./bin/llvm-tblgen -I ../lib/Target/RISCV/ -I ../include/ -I ../lib/Target/ ../lib/Target/RISCV/RISCV.td
- View generated files:
  - $BUILDDIR/lib/Target/RISCV/RISCVGenRegisterInfo.inc
  - $BUILDDIR/lib/Target/RISCV/RISCVGenInstrInfo.inc
  - $BUILDDIR/lib/Target/RISCV/RISCVGenAsmMatcher.inc
  - And more

Codegen

LLVM IR example

```
define i32 @small_const() {
  ret i32 2047
}

define i32 @large_const() nounwind {
  ret i32 -559038737
}

define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}

define i32 @addi(i32 %a) {
  %1 = add i32 %a, 1234
  ret i32 %1
}
```

Understanding codegen: the plan

- Instruction selection patterns
- SelectionDAG and the lowering process
- Calling convention support, lowering returns and formal arguments
- Testing
- Debugging
- Instruction selection in C++
- Example: RV32D

Introducing the SelectionDAG

We will define “patterns” in order to match operations to machine instructions. These aren’t written directly against LLVM IR, but against a directed acyclic graph structure called the SelectionDAG

SelectionDAG processing:

- SelectionDAGBuilder: visit each IR instruction and generate appropriate SelectionDAG nodes
- DAGCombiner: optimisations
- LegalizeTypes: legalize types
- DAGCombiner: optimisations
- LegalizeDAG: legalize operations
- SelectionDAGISel: instruction selection (produce MachineSDNodes)
- ScheduleDAG: scheduling
- Then convert to MachineInstr

See SelectionDAGISel::DoInstructionSelection which drives this process.

Instruction selection patterns: immediates

- Use TableGen multiple inheritance so simm12 is also an ImmLeaf
- Patterns are defined with Pat<dag from, dag to>
- The simm12 ImmLeaf is a pattern fragment with a predicate
- See include/llvm/Target/TargetSelectionDAG.td

```
def simm12
  : Operand<XLenVT>,
    ImmLeaf<XLenVT, [{return isInt<12>(Imm);}]> {
  let ParserMatchClass = SImmAsmOperand<12>;
  let EncoderMethod = "getImmOpValue";
  let DecoderMethod = "decodeSImmOperand<12>";
}

def : Pat<(simm12:$imm), (ADDI X0, simm12:$imm)>;
```

Instruction selection patterns: immediates

Materialising 32-bit immediates requires manipulating the input using SDNodeXForm.

```
// Extract least significant 12 bits from an immediate value
// and sign extend them.
def LO12Sext : SDNodeXForm<imm, [{
  return CurDAG->getTargetConstant(
    SignExtend64<12>(N->getZExtValue()),SDLoc(N), N->getValueType(0)
    );
}]>;


// Extract the most significant 20 bits from an immediate value.
// Add 1 if bit 11 is 1, to compensate for the low 12 bits in the
// matching immediate addi or ld/st being negative.
def HI20 : SDNodeXForm<imm, [{
      return CurDAG->getTargetConstant(
        ((N->getZExtValue()+0x800) >> 12) & 0xfffff,
         SDLoc(N), N->getValueType(0));
}]>;

def : Pat<(simm32:$imm),
  (ADDI (LUI (HI20 imm:$imm)), (LO12Sext imm:$imm))>,
```

Instruction selection patterns: add(i)

Question: What will happen if we didn't define the ADDI pattern and the instruction selector encountered an add with constant operand?

The RISC-V backend chooses to split these pattern definitions from the instruction definition.

```
def : Pat<(add GPR:$rs1, GPR:$rs2),
          (ADD GPR:$rs1, GPR:$rs2)>;


def : Pat<(add GPR:$rs1, simm12:$imm12),
          (ADDI GPR:$rs1, simm12:$imm12)>;
```

More complex selection patterns: loads

This example introduces tablegen multiclasses, as well as the FrameIndex addressing mode.

See both include/llvm/Target/TargetSelectionDAG.td and include/llvm/CodeGen/ISDOpcodes.h

```
multiclass LdPat<PatFrag LoadOp, RVInst Inst> {
  def : Pat<(LoadOp GPR:$rs1), (Inst GPR:$rs1, 0)>;
  def : Pat<(LoadOp AddrFI:$rs1), (Inst AddrFI:$rs1, 0)>;
  def : Pat<(LoadOp (add GPR:$rs1, simm12:$imm12)),
            (Inst GPR:$rs1, simm12:$imm12)>;
  def : Pat<(LoadOp (add AddrFI:$rs1, simm12:$imm12)),
            (Inst AddrFI:$rs1, simm12:$imm12)>;
  def : Pat<(LoadOp (IsOrAdd AddrFI:$rs1, simm12:$imm12)),
            (Inst AddrFI:$rs1, simm12:$imm12)>;
}

defm : LdPat<sextloadi8, LB>;
defm : LdPat<extloadi8, LB>;
defm : LdPat<sextloadi16, LH>;
defm : LdPat<extloadi16, LH>;
defm : LdPat<load, LW>, Requires<[IsRV32]>;
defm : LdPat<zextloadi8, LBU>;
defm : LdPat<zextloadi16, LHU>;
```

A trivial SelectionDAG example

```
SelectionDAG has 8 nodes:
  t0: ch = EntryToken
      t2: i32,ch = CopyFromReg t0, Register:i32 %0
    t4: i32 = add t2, Constant:i32<1234>
  t6: ch,glue = CopyToReg t0, Register:i32 $x10, t4
  t7: ch = RISCVISD::RET_FLAG t6, Register:i32 $x10, t6:1
```

More on SelectionDAG

- At any point in the SelectionDAG legalising+combining process, you may need or want to introduce target-specific DAG nodes. These are different to MachineSDNodes
- There's a huge amount of target-independent support code here, but you are responsible for providing necessary target-specific hooks to help guide the process.
- Despite the combining + legalisation is mostly "done for you", as a backend developer you'll likely spend a lot of time scrutinising this process. You may also want to push some logic up to the target-independent path and out of your backend.
- See also: last year's GlobalISel tutorial. GlobalISel is a proposed eventual replacement for SelectionDAG.
- Note: code generation **isn't** over once MachineInstr are produced. There's still register allocation, as well as target-independent and target-dependent MachineFunction passes

RISCVTargetLowering (RISCVISelLowering.cpp)

- Indicate legal types and operations, through addRegisterClass and setOperationAction calls in the constructor
- Any custom lowering (target-specific legalisation) and target DAG combines go here
- May implement target hooks used to influence codegen
- Must implement LowerFormalArguments, LowerReturn, and LowerCall, and others
  - E.g. LowerFormalArguments will assign locations to arguments (using calling convention implementation) and create DAG nodes (CopyFromReg or stack loads).
- Calling conventions can be specified using TableGen, custom C++, or a combination

Note: more support code is also needed, e.g. RISCVRegisterInfo, RISCVInstrInfo, RISCVFrameLowering

Testing

See test/CodeGen/RISCV/*.ll

Make heavy use of update_llc_test_checks.py to generate and maintain CHECK lines.

In-tree unit tests involve no execution. You need external executable tests (e.g. GCC torture suite, programs in LLVM's test-suite repo, …

High quality tests and high test coverage is _essential_ and has a high return on investment

```
; NOTE: Assertions have been autogenerated by
; utils/update_llc_test_checks.py
; RUN: llc -mtriple=riscv32 -verify-machineinstrs < %s
; RUN:   | FileCheck %s -check-prefix=RV32I

define i32 @addi(i32 %a) nounwind {
; RV32I-LABEL: addi:
; RV32I:       # %bb.0:
; RV32I-NEXT:    addi a0, a0, 1
; RV32I-NEXT:    ret
  %1 = add i32 %a, 1
  ret i32 %1
}
```

Debugging

- Write good, specific and minimised tests
- Ensure you have a debug+asserts build
- -debug flag to llc
- -print-after-all to llc
- llvm_unreachable, assert
- DAG.dump(), errs() << *Inst << “\n”, or fire up your favourite debugger
- sys::PrintStackTrace(llvm::errs())

Debugging instruction selection

bin/llc -mtriple=riscv32 -verify-machineinstrs < foo.ll -debug-only=isel

Then look up these indices in

$BUILDDIR/lib/Target/RISCV/RISCVGenDAGISel.inc

```
ISEL: Starting selection on root node: t4: i32 = add t2,
      Constant:i32<1234>
ISEL: Starting pattern match
     Initial Opcode index to 9488
     TypeSwitch[i32] from 9499 to 9502
     Match failed at index 9506
     Continuing at 9519
     Match failed at index 9520
     Continuing at 9533
     Morphed node: t4: i32 = ADDI t2, TargetConstant:i32<12
ISEL: Match complete!

/*  9484*/  /*SwitchOpcode*/ 20|128,1/*148*/,
           TARGET_VAL(ISD::ADD),// ->9636
/*  9488*/    OPC_RecordChild0, // #0 = $Rs
/*  9489*/    OPC_RecordChild1, // #1 = $imm12
/*  9490*/    OPC_Scope, 105, /*->9597*/ // 3 children in
Scope
/*  9492*/      OPC_MoveChild1,
/*  9493*/      OPC_CheckOpcode, TARGET_VAL(ISD::Constant),
/*  9496*/      OPC_CheckPredicate, 2, // Predicate_simm12
/*  9498*/      OPC_MoveParent,
/*  9499*/      OPC_SwitchType /*2 cases */, 80, MVT::i32,/
->9582
```

Instruction selection in C++

Our ADDI pattern, but in C++

RISCVDAGToDAGISel::Select in lib/Target/RISCV/RISCVISelDAGToDAG.cpp

```
switch (Opcode) {
case ISD::ADD: {
  SDValue Op0 = Node->getOperand(0);
  SDValue Op1 = Node->getOperand(1);
  if (Op1.getOpcode() == ISD::Constant) {
    int64_t Imm =
      cast<ConstantSDNode>(Op1.getNode())->getSExtValue();
    if (!isInt<12>(Imm))
      break;
    SDValue SDImm = CurDAG->getTargetConstant(Imm, DL, VT);
    ReplaceNode(Node, CurDAG->getMachineNode(RISCV::ADDI,
                DL, VT, Op0, SDImm));
    return;
  }
  break;
}
}

// Call into tablegenned instruction selection
SelectCode(Node);
```

A hairier example: RV32D soft-float ABI

- The D extension adds double-precision floating point.
- f64 and i32 are legal types. There are no GPR <-> FPR move instructions for double-precision floats, must go via the stack.
- The legalizer can typically handle this, except sometimes these moves are introduced after legalisation.
    - e.g. an operation is legalised to an intrinsic call, the f64 must be passed/returned in a pair of i32. At this point, it’s illegal to bitcast to use BUILD_PAIR to create an i64 or to BITCAST an f64 to i64 in order to perform EXTRACT_ELEMENT
- We need to introduce custom handling

A hairier example: RV32D soft-float ABI Solution

- Introduce target-specific BuildPairF64 and SplitF64 nodes to directly convert f64 <-> (i32,i32)
- Modify calling convention implementation to properly respect rules for passing f64 in the soft-float ABI (reg+reg, reg+stack, or just stack)
- Generate these nodes in LowerCall, LowerReturn, and LowerFormalArguments when appropriate
- Add a target DAGCombine to remove redundant BuildPairF64+SplitF64 pairs
- Introduce pseudo-instructions with a custom inserter to select for these target-specific nodes
- Generate necessary stack loads/stores in the custom inserters

```
def SDT_RISCVBuildPairF64 : SDTypeProfile<1, 2, [SDTCisVT<0, f64>,
                                                 SDTCisVT<1, i32>,
                                                 SDTCisSameAs<1, 2>]>;

def RISCVBuildPairF64 : SDNode<"RISCVISD::BuildPairF64",
SDT_RISCVBuildPairF64>;
```

The end?

- This has been a whirlwind and selective tour, there's much more to learn.
- Check out resources such as the LLVM documentation, or read the source (e.g. my split-out educational patchset at github.com/lowrisc/riscv-llvm)
- Contact: asb@lowrisc.org
- Cement your new-found knowledge with some practical experimentation in the the Coding Lab tomorrow, 2pm!
    - Instructions https://www.lowrisc.org/llvm/devmtg18/
- Questions?

Overflow topics

- Prolog and epilog insertion
- Floating point
- Atomics lowering
- Compression support
- Instruction properties, branch analysis
- ...