Diving into RISC-V LLVM: Supporting custom instruction set extensions

Alex Bradbury
asb@lowrisc.org @asbradbury @lowrisc

7th May 2018

What is LLVM?

RISC-V LLVM

- Status
- Approach

Focus for today

- Whirlwind tour, stepping through a simple instruction set extension example
  - Aim to give you a basic ‘feel’ for things rather than explaining every detail
- Assumption: you’re happy modifying your simulator or core implementation, but have little familiarity with LLVM internals.

Instructions to support

- Pick some instructions from github.com/cliffordwolf/xbitmanip – proposed bit manipulation instructions
  - clz
  - andc
  - grevi

Building LLVM

```
$ git clone https://llvm.org/git/llvm.git
$ git clone https://llvm.org/git/clang.git tools/clang
$ cd llvm && mkdir build && cd build
$ cmake -G Ninja -DCMAKE_BUILD_TYPE="Debug" \
  -DBUILD_SHARED_LIBS=True -DLLVM_USE_SPLIT_DWARF=True \
  -DLLVM_OPTIMIZED_TABLEGEN=True \
  -DLLVM_BUILD_TESTS=True \
  -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ \
  -DLLVM_ENABLE_LLD=True \
  -DLLVM_TARGETS_TO_BUILD="X86;" \
  -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="RISCV" ../
$ cmake --build .
```

New instructions the easy way: do nothing!

```
(Note: We've chosen brownfield encodings here. See
RISC-V ISA Manual for discussion of available encoding
space)

.insn i OPC_OP_IMM 0b001, t0, t1, 0b000000100000
.insn r OPC_OP 0b111, 0b1000000, t0, t1, t2
.insn I OPC_OP_IMM 0b001, t0, t1, 0b110000011111
```

Handy for quick experiments, but limited

Starting with some tests

```
+++ b/test/MC/RISCV/rv32xbitmanip-valid.s
@@ -0,0 +1,11 @@
+# RUN: llvm-mc %s -triple=riscv32 -mattr=+xbitmanip -show-encoding \
+# RUN:     | FileCheck -check-prefixes=CHECK,CHECK-INST %s
+# RUN: llvm-mc -filetype=obj -triple riscv32 -mattr=+xbitmanip < %s \
+# RUN:     | llvm-objdump -mattr=+xbitmanip -d - \
+# RUN:     | FileCheck -check-prefix=CHECK-INST %s
+
+clz t0, a0
+andc t1, a1, a2
+grevi t2, a3, 0b11000
+grevi t3, a4, 0
+grevi t4, a5, 0b11111
```

Starting with some tests

```
--- /dev/null
+++ b/test/MC/RISCV/rv32xbitmanip-invalid.s
@@ -0,0 +1,11 @@
+# RUN: not llvm-mc -triple riscv32 -mattr=+xbitmanip < %s 2>&1 |
FileCheck %s
+
+# Too many operands
+clz t0, a0, a1
+
+# Operands of incorrect register class
+clz t1, f1
+
+# Out of range immediate for grevi
+grevi a0, a0, 32
+grevi a0, a0, -1
```

Defining instructions: CLZ

- See lib/Target/RISCV/RISCVInstr{Info,Formats}.td

```
+let Predicates = [HasXBitmanip] in {
+let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
+def CLZ : RVInstI<0b001, OPC_OP_IMM, (outs GPR:$rd), (ins GPR:$rs1),
+                  "clz", "$rd, $rs1"> {
+  let imm12 = 0b000000100000;
+}
+…
+} // Predicates = [HasXBitmanip]
```

Defining instructions: ANDC

- See lib/Target/RISCV/RISCVInstr{Info,Formats}.td

```
+def ANDC : ALU_rr<0b1000000, 0b111, "andc">;
```

Defining instructions: GREVI

```
+let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
+def GREVI : RVInst<(outs GPR:$rd), (ins GPR:$rs1, uimm5:$mode),
+                   "grevi", "$rd, $rs1, $mode", [], InstFormatI> {
+  bits<5> mode;
+  bits<5> rs1;
+  bits<5> rd;
+
+  let Inst{31-25} = 0b1100000;
+  let Inst{24-20} = mode;
+  let Inst{19-15} = rs1;
+  let Inst{14-12} = 0b001;
+  let Inst{11-7} = rd;
+  let Opcode = OPC_OP_IMM.Value;
+}
```

Adding check lines to tests

```
+# CHECK-INST: clz t0, a0
+# CHECK: encoding: [0x93,0x12,0x05,0x02]
 clz t0, a0
+# CHECK-INST: andc t1, a1, a2
+# CHECK: encoding: [0x33,0xf3,0xc5,0x80]
 andc t1, a1, a2
+# CHECK-INST: grevi t2, a3, 24
+# CHECK: encoding: [0x93,0x93,0x86,0xc1]
 grevi t2, a3, 0b11000
+# CHECK-INST: grevi t3, a4, 0
+# CHECK: encoding: [0x13,0x1e,0x07,0xc0]
 grevi t3, a4, 0
+# CHECK-INST: grevi t4, a5, 31
+# CHECK: encoding: [0x93,0x9e,0xf7,0xc1]
 grevi t4, a5, 0b11111
```

Going further: codegen (tests)

```
+; RUN: llc -mtriple=riscv32 -mattr=+xbitmanip -verify-machineinstrs < %s \
+; RUN:   | FileCheck %s -check-prefix=RV32XBITMANIP
+
+declare i32 @llvm.ctlz.i32(i32, i1)
+
+define i32 @test_clz(i32 %a) nounwind {
+  %tmp = call i32 @llvm.ctlz.i32(i32 %a, i1 false)
+  ret i32 %tmp
+}
+
+define i32 @test_andc(i32 %a, i32 %b) nounwind {
+  %1 = xor i32 %b, -1
+  %2 = and i32 %a, %1
+  ret i32 %2
+}
See test/Codegen/* for many examples
```

Going further: codegen (patterns)

```
Add SelectionDAG patterns

RISCVInstrInfoXBitmanip.td:

+let Predicates = [HasXBitmanip] in {
+def : Pat<(ctlz GPR:$rs1), (CLZ GPR:$rs1)>;
+def : Pat<(and GPR:$rs1, (not GPR:$rs2)), (ANDC GPR:$rs1, GPR:$rs2)>;
+} // Predicates = [HasXBitmanip]

RISCVISelLowering.cpp:
-  setOperationAction(ISD::CTLZ, XLenVT, Expand);
+  if (!Subtarget.hasXBitmanip())
+    setOperationAction(ISD::CTLZ, XLenVT, Expand);
```

Further tips

- When things don’t work as expected -debug-only=isel is your friend
- update_llc_test_checks.py
- Study existing code: both in RISCV and in other backends (RISCV, Lanai, BPF backends are all very readable)
- What about GREVI?

Further resources

- llvm.org/docs
- llvmweekly.org
- github.com/lowrisc/riscv-llvm
- lowrisc.org/llvm
  - I’ll expand this into a written tutorial
- Talk to me afterwards or at the poster session

Questions?

- Talk to me afterwards or at the poster session
- Email: asb@lowrisc.org