Supporting custom RISC-V extensions in LLVM

Alex Bradbury asb@igalia.com

RISC-V Summit Europe, 2024-06-24

Tutorial overview

Aims

- Share knowledge on what is needed to implement custom RISC-V extensions.
- Grow intuition on challenges involved and what kind of questions to ask.
- Provide pointers on where to find out more

Tutorial overview

Structure

- Introduction to LLVM
- Introduction to RISC-V extensions
- Implementing RISC-V extension support
- Upstreaming and final thoughts

Introduction to LLVM

What is LLVM?

- A collection of modular compiler and toolchain technologies
- Modern C++ implementation
- Library-based design
- Permissively licensed
- C/C++ toolchain (Clang) and equivalents to various binutils tools
- Primary backend for e.g. Rust
- Used by many downstream vendor toolchains

What is in LLVM?

- **clang/**
- *compiler-rt/*
- flang/
- mlir/
- libc/
- libcxx/
- *lld/*
- lldb/
- **llvm/**
- ….

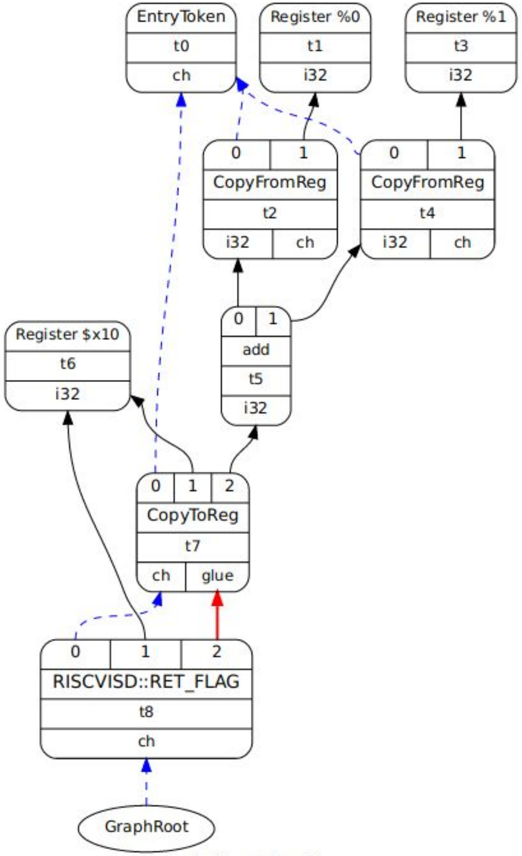
LLVM compilation flow

```
int add(int a, int b) {
  return a+b;
}
```

LLVM compilation flow

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

LLVM compilation flow



isel input for add:

LLVM compilation flow

```
add.o:    file format ELF32-riscv

Disassembly of section .text:
0000000000000000 add:
     0:    33 05 b5 00     add    a0, a0, a1
     4:    67 80 00 00     ret
```

LLVM IR

- Types, instructions, intrinsics
- Dump IR from clang with `-emit-llvm -S` (best to use -O1 or above)
- See https://llvm.org/docs/LangRef.html

```
define float @sqrt_f32(float %a) nounwind {
  %1 = call float @llvm.sqrt.f32(float %a)
  ret float %1
}
```

SelectionDAG pipeline

- Build initial DAG
- Optimize SelectionDAG (DAG combiner)
- Legalize SelectionDAG types (eliminate any types unsupported by the target)
- Optimize SelectionDAG (DAG combiner)
- Legalize SelectionDAG operations (eliminate operations unsupported by the target)
- Optimize SelectionDAG (DAG combiner)
- Select instructions (translate to DAG of target instructions)
- SelectionDAG scheduling and MachineFunction emission

TableGen

- LLVM-specific domain specific language.
- Multiple uses, but we'll use it primarily for instruction definitions and codegen patterns.
- Definitions, classes, multiclasses
- Produces output in different forms depending on the TableGen backend invoked (not to be confused with LLVM backend)
- See https://llvm.org/docs/TableGen/ and examples on next slides.

TableGen: Instruction definitions

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_ri<bits<3> funct3, string opcodestr>
    : RVInstI<funct3, OPC_OP_IMM, (outs GPR:$rd), (ins GPR:$rs1,
simm12:$imm12),
              opcodestr, "$rd, $rs1, $imm12">,
    Sched<[WriteIALU, ReadIALU]>;


let isReMaterializable = 1, isAsCheapAsAMove = 1 in
def ADDI  : ALU_ri<0b000, "addi">;
```

TableGen: Instruction definitions

```
./bin/llvm-tblgen -I
../llvm/lib/Target/RISCV/ -I
../llvm/include/ -I ../llvm/lib/Target/
../llvm/lib/Target/RISCV/RISCV.td | less
```

(Yes the output is unreadable on this slide, the point is instructions have many many properties and you can check them manually)

```
def ADDI {      // InstructionEncoding Instruction RVInstCommon RVInst RVInstIBase RVInstI Sched ALU_ri
  field bits<32> Inst = { imm12{11}, imm12{10}, imm12{9}, imm12{8}, imm12{7}, imm12{6}, imm12{5}, imm12{4}, imm12{3}, imm12{2}, imm12{1}, imm12{0},
 rd{2}, rd{1}, rd{0}, 0, 0, 1, 0, 0, 1, 1 };
  field bits<32> SoftFail = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
  int Size = 4;
  string DecoderNamespace = "";
  list<Predicate> Predicates = [];
  string DecoderMethod = "";
  bit hasCompleteDecoder = 1;
  string Namespace = "RISCV";
  dag OutOperandList = (outs GPR:$rd);
  dag InOperandList = (ins GPR:$rs1, simm12:$imm12);
  string AsmString = "addi      $rd, $rs1, $imm12";
  EncodingByHwMode EncodingInfos = ?;
  list<dag> Pattern = [];
  list<Register> Uses = [];
  list<Register> Defs = [];
  int CodeSize = 0;
  int AddedComplexity = 0;
  bit isPreISelOpcode = 0;
  bit isReturn = 0;
  bit isBranch = 0;
  bit isEHScopeReturn = 0;
  bit isIndirectBranch = 0;
  bit isCompare = 0;
  bit isMoveImm = 0;
  bit isMoveReg = 0;
  bit isBitcast = 0;
  bit isSelect = 0;
  bit isBarrier = 0;
  bit isCall = 0;
  bit isAdd = 0;
  bit isTrap = 0;
  bit canFoldAsLoad = 0;
  bit mayLoad = 0;
  bit mayStore = 0;
  bit mayRaiseFPException = 0;
  bit isConvertibleToThreeAddress = 0;
  bit isCommutable = 0;
  bit isTerminator = 0;
  bit isReMaterializable = 1;
  bit isPredicable = 0;
  bit isUnpredicable = 0;
  bit hasDelaySlot = 0;
  bit usesCustomInserter = 0;
  bit hasPostISelHook = 0;
  bit hasCtrlDep = 0;
  bit isNotDuplicable = 0;
  bit isConvergent = 0;
  bit isAuthenticated = 0;
  bit isAsCheapAsAMove = 1;
  bit hasExtraSrcRegAllocReq = 0;
  bit hasExtraDefRegAllocReq = 0;
  bit isRegSequence = 0;
  bit isPseudo = 0;
  bit isMeta = 0;
  bit isExtractSubreg = 0;
  bit isInsertSubreg = 0;
  bit variadicOpsAreDefs = 0;
  bit hasSideEffects = 0;
  bit isCodeGenOnly = 0;
  bit isAsmParserOnly = 0;
  bit hasNoSchedulingInfo = 0;
  InstrItinClass Itinerary = NoItinerary;
  list<SchedReadWrite> SchedRW = [WriteIALU, ReadIALU];
  string Constraints = "";
  string DisableEncoding = "";
  string PostEncoderMethod = "";
  bits<64> TSFlags = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 1 };
  string AsmMatchConverter = "";
  string TwoOperandAliasConstraint = "";
  string AsmVariantName = "";
  bit UseNamedOperandTable = 0;
  bit UseLogicalOperandMappings = 0;
  bit FastISelShouldIgnore = 0;
  bit HasPositionOrder = 0;
  RISCVVConstraint RVVConstraint = NoConstraint;
  bits<3> VLMul = { 0, 0, 0 };
  bit ForceTailAgnostic = 0;
  bit IsTiedPseudo = 0;
  bit HasSEWOp = 0;
  bit HasVLOp = 0;
  bit HasVecPolicyOp = 0;
  bit IsRVVWideningReduction = 0;
  bit UsesMaskPolicy = 0;
  bit IsSignExtendingOpW = 0;
  bit HasRoundModeOp = 0;
  bit UsesVXRM = 0;
  bits<2> TargetOverlapConstraintType = { 0, 0 };
  bits<5> rs1 = { ?, ?, ?, ?, ? };
  bits<5> rd = { ?, ?, ?, ?, ? };
  bits<12> imm12 = { ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? };
}
```

TableGen: Pattern definitions

```
def : PatGprGpr<add, ADD>;
def : PatGprSimm12<add, ADDI>;

let Predicates = [IsRV64, NotHasStdExtZba] in {
def : Pat<(i64 (and GPR:$rs1, 0xffffffff)), (SRLI (i64 (SLLI
GPR:$rs1, 32)), 32)>;
```

Testing

- llvm/test/CodeGen/RISCV
- llvm/test/MC/RISCV
- clang/test/CodeGen/RISCV
- Runner: lit
- Test language: FileCheck
- update_llc_test_checks.py
- Separate execution tests, e.g. GCC torture suite

.s FileCheck test example

```
# 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 %s -triple riscv64 -riscv-no-aliases -show-encoding \

# CHECK-ASM-AND-OBJ: lui a0, 2
# CHECK-ASM: encoding: [0x37,0x25,0x00,0x00]
lui a0, 2

# CHECK-ASM-AND-OBJ: ori a0, a1, -2048
# CHECK-ASM: encoding: [0x13,0xe5,0x05,0x80]
ori a0, a1, -2048
```

.ll FileCheck test example

```
; 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
; RUN: llc -mtriple=riscv64 -verify-machineinstrs < %s \
; RUN:   | FileCheck %s -check-prefix=RV64I

define i32 @addi(i32 %a) nounwind {
; RV32I-LABEL: addi:
; RV32I:    # %bb.0:
; RV32I-NEXT:  addi a0, a0, 1
; RV32I-NEXT:  ret
;
; RV64I-LABEL: addi:
; RV64I:    # %bb.0:
; RV64I-NEXT:  addiw a0, a0, 1
; RV64I-NEXT:  ret
  %1 = add i32 %a, 1
  ret i32 %1
}
```

Building LLVM and running tests

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
mkdir -p build && cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE="Debug" \
-DLLVM_ENABLE_PROJECTS="clang;lld" \
-DLLVM_ENABLE_RUNTIMES="compiler-rt" \
-DBUILD_SHARED_LIBS=True -DLLVM_USE_SPLIT_DWARF=True \
-DLLVM_BUILD_TESTS=True \
-DLLVM_CCACHE_BUILD=ON \
-DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ \
-DLLVM_ENABLE_LLD=True \
-DLLVM_TARGETS_TO_BUILD="all" \
-DLLVM_APPEND_VC_REV=False ../llvm
cmake --build .
 ./bin/llvm-lit -s -v ../llvm/test/CodeGen/RISCV
```

Further info

- LLVM backend development by example
  https://www.youtube.com/watch?v=AFaIP-dF-RA
- https://llvm.org/docs/WritingAnLLVMBackend.html
- https://llvm.org/docs/CodeGenerator.html
- Code and patch reading

Introduction to RISC-V extensions

RISC-V extensions

"To improve efficiency and to reduce costs, SoCs add custom application-specific accelerators. To match the needs of SoCs while maintaining a stable software base, a free, open ISA should have:

i) a small core set of instructions that compilers and OS's can depend upon;

ii) standard but optional extensions for common ISA additions to help customize the SoC to the application; and

iii) space for entirely new opcodes to invoke the application-specific accelerators."

Source: Instruction Sets Should Be Free: The Case For RISC-V (Krste Asanović David A. Patterson)

What might an extension impact?

- New instruction definitions
- New machine state (user-visible registers or otherwise)
- CSRs
- New instruction formats and relocations
- Other semantic changes needing software stack changes
- ...

Types of RISC-V extension

- Standard, non-standard (vendor / custom), not-yet-ratified
- Non-standard (vendor/custom) extensions
  - Commonly uses encoding space marked as “custom”
  - Non-conforming extensions may use standard or reserved encodings.
- User-level, privileged, machine-level. e.g. F, D, Zicsr, Satp, …
- See also riscv-c-api-doc, riscv-toolchain-conventions repos
  - Instruction naming convention for vendor extensions : mnemonics start with two letter vendor prefix.

Needed ingredients for extension compiler support

- Semantics
- Concrete encoding
- (Ideally) instruction set simulator
- Definitions for any C intrinsics

Other topics

- Extension versioning
  - Single version strongly preferred
- When and why to add custom extension support
  - Potential for unlocking hardware/software co-design. May want additional tooling for this

Implementing RISC-V extension support

Basic implementation loop

- Take an incremental approach wherever possible.
  - e.g. MC layer, then codegen.
  - Simple cases then more complex ones, in separate commits.
- Loop:
  - Pick next simplest problem to solve
  - Write a test of some sort
  - Make code changes to get it working, explore any issues
  - Clean up / expand test case and clean up implementation
  - Commit

Minimal definitions for a new extension

- Need to define extension name and version and ensure ELF build attributes related to it can be recognised and generated.
- Add to RISCVFeatures.td

```
def FeatureStdExtB
    : RISCVExtension<"b", 1, 0,
                     "'B' (the collection of the Zba, Zbb, Zbs extensions)",
                     [FeatureStdExtZba, FeatureStdExtZbb, FeatureStdExtZbs]>;
def HasStdExtB : Predicate<"Subtarget->hasStdExtB()">,
                         AssemblerPredicate<(all_of FeatureStdExtB),
                         "'B' (the collection of the Zba, Zbb, Zbs extensions)">;
```

- Add tests in llvm/test/CodeGen/RISCV/attributes.ll and clang/test/Preprocessor/riscv-target-features.c

Overview of potential approaches/levels of support

- .insn only
- MC layer only
- MC layer + intrinsics
- MC layer + codegen
- MC layer + more complex codegen
- "" + ABI changes + linker changes / relocations + complex interactions + …

Note: Find a similar extension and learn from how that was implemented if you can.

CSRs only

- Note: we have some limitations here currently (dealing with encoding clashes, WIP)
- See RISCVSystemOperands.td

```
//===----------------------------------------------------------------------===//
// State Enable Extension (Smstateen)
//===----------------------------------------------------------------------===//

// sstateen0-sstateen3 at 0x10C-0x10F, mstateen0-mstateen3 at 0x30C-0x30F,
// mstateen0h-mstateen3h at 0x31C-0x31F, hstateen0-hstateen3 at 0x60C-0x60F,
// and hstateen0h-hstateen3h at 0x61C-0x61F.
foreach i = 0...3 in {
  def : SysReg<"sstateen"#i, !add(0x10C, i)>;
  def : SysReg<"mstateen"#i, !add(0x30C, i)>;
  let isRV32Only = 1 in
  def : SysReg<"mstateen"#i#"h", !add(0x31C, i)>;
  def : SysReg<"hstateen"#i, !add(0x60C, i)>;
  let isRV32Only = 1 in
  def : SysReg<"hstateen"#i#"h", !add(0x61C, i)>;
}
```

.insn only

- The "do-nothing" option. Users use `.insn` to help construct appropriately encoded instructions in inline assembly or .s files.
- Could be wrapped up in helper functions.
- Examples
  - `.insn i  OP_IMM,  0, a0, a1, 13`
  - `.insn r  0x43,  0,  0, fa0, fa1, fa2, fa3`
  - `.insn sb BRANCH,  0, a0, a1, target`
- Some complication getting accurate ELF attributes set

MC layer only

- Add llvm/lib/Target/RISCV/RISCVInstrInfo$Foo.td and appropriate llvm/test/MC/RISCV tests
- May sometimes require register definitions (RISCVRegisterInfo.td) or changes to the assembly parser (RISCVAsmParser.cpp)
- See previous instruction definition example.

MC layer + intrinsics

- LLVM intrinsics only, or LLVM intrinsics + C intrinsics?
  - The former may be sufficient if a middle-end pass can select your intrinsics.
- Likely to touch:
  - clang/include/clang/Basic/BuiltinsRISCV.td
  - llvm/include/llvm/IR/IntrinsicsRISCV.td
  - llvm/lib/Target/RISCV/RISCVISelLowering.cpp

MC layer + intrinsics example: clmul from zbc

- clang/include/clang/Basic/BuiltinsRISCV.td:

```
def clmul_32 : RISCVBuiltin<"unsigned int(unsigned int,
unsigned int)", "zbc|zbkc">;
def clmul_64 : RISCVBuiltin<"uint64_t(uint64_t, uint64_t)", "zbc|zbkc,64bit">;
```

- clang/lib/CodeGen/CGBuiltin.cpp, add to switch:

```
case RISCV::BI__builtin_riscv_clmul_32:
case RISCV::BI__builtin_riscv_clmul_64:
    ID = Intrinsic::riscv_clmul;
    break;
```

MC layer + intrinsics example: clmul from zbc

- llvm/include/llvm/IR/IntrinsicsRISCV.td

```
// Zbc or Zbkc
 def int_riscv_clmul  : BitManipGPRGPRIntrinsics;
 def int_riscv_clmulh : BitManipGPRGPRIntrinsics;
```

- llvm/lib/Target/RISCV/RISCVISelLowering.cpp SDValue RISCVTargetLowering::LowerINTRINSIC_WO_CHAIN

```
case Intrinsic::riscv_clmul:
    …
    return DAG.getNode(RISCVISD::CLMUL, DL, XLenVT, Op.getOperand(1),
                       Op.getOperand(2));
```

- (Skipping over some type legalisation complexities for i32 variant on RV64)

MC layer + intrinsics example: clmul from zbc

- llvm/include/llvm/lib/Target/RISCVInstrInfoZb.td

```
def riscv_clmul   : SDNode<"RISCVISD::CLMUL",   SDTIntBinOp>;
def riscv_clmulh  : SDNode<"RISCVISD::CLMULH",  SDTIntBinOp>;
def riscv_clmulr  : SDNode<"RISCVISD::CLMULR",  SDTIntBinOp>;
…
let Predicates = [HasStdExtZbcOrZbkc] in {
def : PatGprGpr<riscv_clmul, CLMUL>;
def : PatGprGpr<riscv_clmulh, CLMULH>;
} // Predicates = [HasStdExtZbcOrZbkc]
```

Note: it's often possible to match the intrinsic directly without a custom SDag node

MC layer + codegen

- Pattern-based codegen - requires that the instruction is reasonably selectable. e.g. popcount, leading zeroes, trailing zeroes in Zbb
- RISCVInstrInfoZb.td

```
let Predicates = [HasStdExtZbb] in {
def : PatGpr<ctlz, CLZ>;
def : PatGpr<cttz, CTZ>;
def : PatGpr<ctpop, CPOP>;
} // Predicates = [HasStdExtZbb]
```

- RISCVISelLowering.cpp: Modify RISCVTargetLowering constructor so it doesn't do `setOperationAction(ISD::CTLZ, XLenVT, Expand);` if you have Zbb

MC layer + more complex codegen

- May require implementation additional hooks to be modified
- Custom C++ selection posible by modifying `RISCVDAGToDAGISel::Select(SDNode *Node)` in RISCVISelLowering.cpp
- See e.g. Zicond, Zfbfmin

Assortment of extension ideas to try

- Scalar efficiency SIG proposals (more work may be needed for 48-bit instrs)
- subleq (Reliable Computing with Ultra-Reduced Instruction Set Co-processors https://caesr.uwaterloo.ca/assets/pdfs/rajendiran_12_uriscdac.pdf)
- setmask (Efficient use of invisible registers in Thumb code, https://ieeexplore.ieee.org/document/1540946 MICRO 05)
- BAA/RPA (A Soft Processor Overlay with Tightly-coupled FPGA Accelerator https://arxiv.org/pdf/1606.06483)
- branch-on-random (https://zilles.cs.illinois.edu/papers/branch-on-random.cgo2008.pdf CGO 2008)

Upstreaming and final thoughts

Managing your code downstream

- Rebase or merge-based flows possible
- Regular updates + testing strongly recommended
  - Internal interfaces can change, easier to handle this incrementally.
  - If your downstream exposes a bug in a new pass, easier to isolate and get it fixed if working on a recent build.
- The cleaner and more well structured your patchset is, the easier to later upstream.

Upstreaming: Why and how

- Why
  - Avoid the issues mentioned before of changing interfaces.
  - Free your users from needing custom toolchain builds.
  - ..
- How
  - See https://llvm.org/docs/Contributing.html
  - Starting a thread on Discourse or discussing in the RISC-V LLVM sync-up call may be useful

Not-yet-ratified and vendor specific extensions policy

- Enable upstream collaboration on not-yet-ratified standards
  - Agreed policy on merging support behind ‘experimental’ flags (e.g. `-menable-experimental-extensions`) with explicit spec version
  - Usual code review standards apply
  - No backwards compatibility or support expectation for anything other than final ratified spec.
- Allow vendor extensions to be supported upstream, reducing need for fragmentation for vendor-specific toolchains.
  - e.g. XVentanaCondOps, Xsfvcp, XTHeadVDot (and many others)
  - Considerations for inclusion: complexity/ invasiveness, support story, user base, ...

Debugging tips

- Write good, specific and minimised tests
- Ensure you have a debug+asserts build
- `-debug` and `-debug-only=passname` flags 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())`
- Study existing tests. e.g. test/CodeGen/RISCV/*
- Make heavy use of update_{llc,mir}_test_checks.py to generate and maintain CHECK lines.

Debugging instruction selection

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

Then look up the listed 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<1234>
ISEL: Match complete!
```

Debugging instruction selection

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

Then look up the listed indices in $BUILDDIR/lib/Target/RISCV/RISCVGenDAGISel.inc

```
/*  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
```

Working on LLVM effectively

- LLVM is a huge and varied codebase => needed skills can vary a lot depending on where you work.
- Needed skills for most extension enablement work
  - A fair understanding of how a compiler works
  - A fair understanding of computer architecture
  - **Code reading, investigation, and debugging skills**
  - Not needed: Extreme level of C++ expertise, perfect knowledge of algorithms taught in compilers university courses.

Additional resources

- LLVM Discourse https://discourse.llvm.org
- RISC-V LLVM biweekly calls (see RISC-V category on Discourse)
- LLVM Weekly https://llvmweekly.org
- Code reading
- My previous backend tutorial
  https://www.youtube.com/watch?v=AFaIP-dF-RA

Questions?

Contact: asb@igalia.com