Enabling hardware/software co-design with RISC-V And LLVM

Alex Bradbury
asb@lowrisc.org @asbradbury @lowRISC

LLVM RISC-V: Why you should be interested

What **isn’t** LLVM?

- Low level virtual machine
- A target independent representation (see PNaCl, SPIR-V, WASM)
- A C compiler (though clang, part of the wider LLVM project, is)
- Magic go-faster pixie dust

What is LLVM

- A permissively licensed compiler infrastructure
- Home to a range of projects beyond LLVM core, e.g. Clang, LLD, LLDB, compiler-rt, libcxx, …
- Codegen backend to various language frontends (e.g. Rust, Julia, Swift)
- A high quality codebase with widespread adoption in industry and academia

LLVM RISC-V: Why is lowRISC interested?

What is lowRISC?

- Not for profit, established in 2014
  - Serve the community of people interested in or who may benefit from open source hardware. Hobbyists, academics, startups, established companies.
- Aim to bring the benefits of open source we enjoy in the software world to hardware
  - ‘Linux of the hardware world’
- Producing a complete SoC platform for others to build upon (multi-core, Linux capable, 64-bit)
- Achieve our main aims by *doing*. Engineering and research are our main activities

Tagged memory

- Associate tags (metadata) with each memory location
- Initial motivation is security - protection against control-flow hijacking attacks
- Also exploring other uses
- Needs compiler and other software support for a full evaluation

Minion cores

- Small microcontroller class cores, initially for I/O
  - Soft peripherals, I/O preprocessing/filtering
  - Offload fine-grain tasks e.g. security policies, debug, performance monitoring
  - Secure, isolated execution (memory safe languages like Rust?)
  - Virtualized devices
- Long term vision: minions distributed through the SoC
  - Explore and evaluate new instruction set extensions to specialise these cores

RISC-V LLVM backend: implementation goals

- Act as a reference backend
- Highly documented
- Clean set of incremental patches, maintained over the long term
- Upstreamed
- Contribute back, improving upstream LLVM where possible

To achieve the above aims, this is a fresh implementation built with these goals in mind.

RISC-V LLVM backend: implementation goals justification

- Lower the barrier for groups who want/need to do compiler work as part of their architectural exploration and to evaluate RISC-V extension proposals
- Support uses of RISC-V and lowRISC in education and research
- Make it as easy as possible to customise the port, and contribute these changes upstream
- Reduce maintenance cost for those who have to maintain changes out of tree (e.g. for long term customer support)

RISC-V is set to be the 32/64-bit architecture with the widest diversity of implementations. This puts extra pressure on the quality of our core tooling.

RISC-V LLVM backend: implementation approach

- Build on my experience authoring and maintaining an LLVM backend over the past 6 years
- Build up from the machine code layer
- Avoid the ‘copy and paste’ trap
- Work incrementally, adding detailed test cases. ‘Slices’ of functionality at a time
- Individual design decisions all motivated by the knowledge people will be extending, adding new instructions etc

Compiling with LLVM: an example

Note: this is a whirlwind tour. Just hoping to give you a flavour

Let’s start with the bit most of us know - RISC-V assembly.

Contrived example: RV32 assuming a static code model

```
example:
    lui  t0, %hi(g_foo)
    lw   t0, %lo(g_foo)(t0)
    lui  t1, %hi(g_bar)
    lw   t1, %lo(g_bar)(t1)
    add  t0, t0, t1
    add  a0, t0, a0
    jalr zero, ra, 0
```

Compiling with LLVM: example (MC layer)

Challenge: convert assembly input to encoded instructions.

Parsing, selecting appropriate relocations, resolving static references where possible, checking for overflowed immediate values, handling assembler mnemonics, encoding the instruction

```
lui   t0, %hi(g_foo)
# encoding: [0xb7,0bAAAA0010,A,A]
#   fixup A - offset: 0, value: %hi(g_foo), kind: fixup_riscv_hi20
# <MCInst #110 LUI
#  <MCOperand Reg:11>
#  <MCOperand Expr:(%hi(g_foo))>>
addi  t0, t0, %lo(g_foo)
# encoding: [0x93,0x82,0bAAAA0010,A]
#   fixup A - offset: 0, value: %lo(g_foo), kind:
fixup_riscv_lo12_i
# <MCInst #83 ADDI
#  <MCOperand Reg:11>
#  <MCOperand Reg:11>
#  <MCOperand Expr:(%lo(g_foo))>>
lw    t0, 0(t0)
# encoding: [0x83,0xa2,0x02,0x00]
# <MCInst #111 LW
#  <MCOperand Reg:11>
#  <MCOperand Reg:11>
#  <MCOperand Imm:0>>
...
```

Compiling with LLVM: example (IR)

LLVM IR input that could result in the previous assembly.

How do we get from this to to RISC-V instructions?

```
@g_foo = global i32 0
@g_bar = global i32 1


define i32 @example(i32 %a) nounwind {
  %1 = load volatile i32, i32* @g_foo
  %2 = load volatile i32, i32* @g_bar
  %3 = add i32 %1, %2
  %4 = add i32 %3, %a
  ret i32 %4
}
```

```
t0: ch = EntryToken
t4: i32 = Constant<0>
t6: i32,ch = load<Volatile LD4[@g_foo]> t0, GlobalAddress:i32<i32* @g_foo> 0, undef:i32
t8: i32,ch = load<Volatile LD4[@g_bar]> t6:1, GlobalAddress:i32<i32* @g_bar> 0, undef:i32
          t9: i32 = add t6, t8
          t2: i32,ch = CopyFromReg t0, Register:i32 %vreg0
     t10: i32 = add t9, t2
t12: ch,glue = CopyToReg t8:1, Register:i32 %X10_32, t10
t13: ch = RISCVISD::RET_FLAG t12, Register:i32 %X10_32, t12:1
```

```
t0: ch = EntryToken
          t20: i32 = LUI TargetGlobalAddress:i32<i32* @g_foo> 0 [TF=2]
      t21: i32 = ADDI t20, TargetGlobalAddress:i32<i32* @g_foo> 0 [TF=1]
t6: i32,ch = LW<Mem:Volatile LD4[@g_foo](dereferenceable)> t21, TargetConstant:i32<0>, t0
          t16: i32 = LUI TargetGlobalAddress:i32<i32* @g_bar> 0 [TF=2]
      t17: i32 = ADDI t16, TargetGlobalAddress:i32<i32* @g_bar> 0 [TF=1]
t8: i32,ch = LW<Mem:Volatile LD4[@g_bar](dereferenceable)> t17, TargetConstant:i32<0>, t6:1
          t9: i32 = ADD t6, t8
          t2: i32,ch = CopyFromReg t0, Register:i32 %vreg0
      t10: i32 = ADD t9, t2
t12: ch,glue = CopyToReg t8:1, Register:i32 %X10_32, t10
t13: ch = PseudoRET Register:i32 %X10_32, t12, t12:1
```

add -> ADDI happens through the application of patterns expressed as S-expr like
`(set GPR:$rd, (add GPR:$rs1, simm12:$imm12))`

```
%vreg0<def> = COPY %X10_32; GPR:%vreg0
%vreg1<def> = LUI <ga:@g_foo>[TF=2]; GPR:%vreg1
%vreg2<def> = ADDI %vreg1<kill>, <ga:@g_foo>[TF=1]; GPR:%vreg2,%vreg1
%vreg3<def> = LW %vreg2<kill>, 0; mem:Volatile LD4[@g_foo](dereferenceable) GPR:%vreg3,%vreg2
%vreg4<def> = LUI <ga:@g_bar>[TF=2]; GPR:%vreg4
%vreg5<def> = ADDI %vreg4<kill>, <ga:@g_bar>[TF=1]; GPR:%vreg5,%vreg4
%vreg6<def> = LW %vreg5<kill>, 0; mem:Volatile LD4[@g_bar](dereferenceable) GPR:%vreg6,%vreg5
%vreg7<def> = ADD %vreg3<kill>, %vreg6<kill>; GPR:%vreg7,%vreg3,%vreg6
%vreg8<def> = ADD %vreg7<kill>, %vreg0; GPR:%vreg8,%vreg7,%vreg0
%X10_32<def> = COPY %vreg8; GPR:%vreg8
PseudoRET %X10_32<imp-use>
```

Adding a count leading zeros instruction

This is almost the simplest possible case for adding a new instruction - no new instruction formats, relocations, or complex selection logic.

We add instruction definitions to the ‘tablegen’ description, a domain-specific language used extensively in LLVM.

```
class FR<bits<7> funct7, bits<3> funct3, bits<7> opcode, dag outs,
         dag ins,string asmstr, list<dag> pattern> :
       RISCVInst<outs, ins, asmstr, pattern, FrmR>
{
      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;
}

+def CLZ : FR<0b0000010, 0b000, 0b0110011, (outs GPR:$rd),
+             (ins GPR:$rs1), "clz\t$rd, $rs1",
+             [(set GPR:$rd, (ctlz GPR:$rs1))]>;
```

Status and roadmap

- MC layer patch series published, mostly reviewed and applied upstream
- Initial codegen almost ready for cleaning up and public review
- Next milestone: Mid Jan - enough codegen for a reasonable portion of the GCC torture suite. Development effort becomes easier to parallelize
- Long term roadmap: depends on future funding and level of external participation

Opportunities for improving LLVM

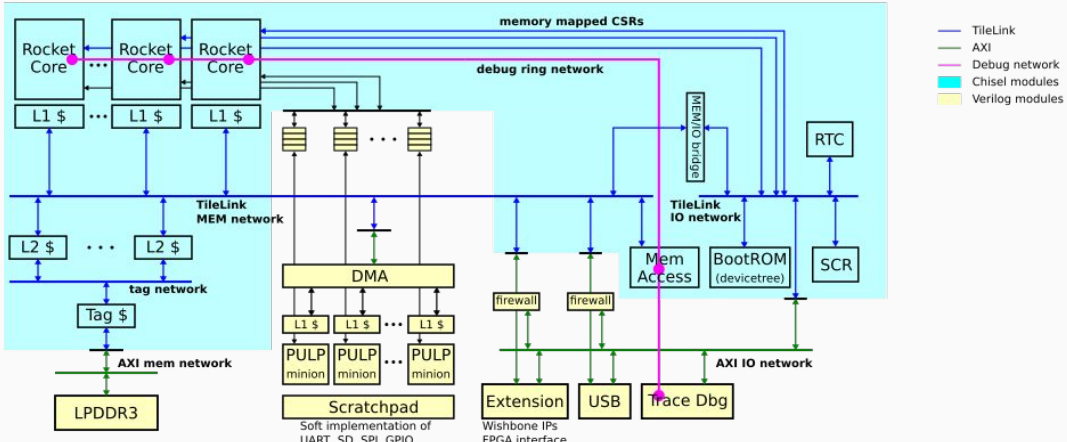
- Reference documentation and tutorials
- Variable-sized register classes: supporting RV32/RV64/RV128 without code duplication. See RFC from Krzysztof Parzyszek. Can be applied across other backends.
- Support for better code reuse amongst LLVM backends (see Linux kernel asm-generic infrastructure)
- Apply lessons from RISC-V backend implementation to clean up and improve other backends and relevant support code

Opportunities for the RISC-V community

- Quantitative exploration of proposals for e.g. bit manipulation instructions, packed SIMD
- Evaluating the proposed vector ISA. Don’t need to fix a spec in stone then throw it over the wall, there’s a huge potential benefit in involving compiler developers in the process
- Novel security features
- ...

lowRISC status

- New FPGA-targeted platform release for end of Feb. Featuring optimised tag cache, integrated minion cores. A complete base to iterate on
- Expect to launch a crowdfunding campaign for a 64-bit, multi-core, Linux-capable RISC-V SoC and development board in 2017



Conclusion

- Getting involved
  - Code reviews reviews.llvm.org
  - See github.com/lowrisc/riscv-llvm
  - PSABI doc github.com/riscv/riscv-elf-psabi-doc/
- Interested in LLVM in general? See www.llvmweekly.org!
- Questions?

Contact: asb@lowrisc.org