Building a production-ready RISC-V LLVM toolchain

Alex Bradbury    asb@lowrisc.org    @asbradbury
Structure of this talk

- Background: on LLVM, lowRISC, and RISC-V LLVM
- Status
- Challenges / selected topics
- Future
What is LLVM?
Why LLVM?
Why LLVM for RISC-V?
(Simplified) Clang/LLVM compilation flow
(Simplified) compilation flow:

C input

```c
#include <stdint.h>

int32_t add(int32_t a, int32_t b) {
    return a+b;
}
```
define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}
(Simplified) compilation flow:

SelectionDAG
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
What does it mean to support a new architecture in LLVM?

Work includes:

- Assembler support
- Instruction definitions
- Instruction patterns
- Target ‘hooks’
- Miscellaneous C++ logic
- Lots of tweaking, bug hunting, bug fixing, etc
Background: lowRISC

- Not-for-profit with a mission to demonstrate, promote, and support the use of open-source hardware
- Producing high quality, security-focused, open, and flexible IP
- Growing our engineer team (7 open positions): [www.lowrisc.org/jobs](http://www.lowrisc.org/jobs) to deliver our roadmap in collaboration with Google and other industry partners
- Expertise includes the LLVM Compiler, novel hardware security extensions and RISC-V tools, hardware and processor design.
- [www.lowrisc.org](http://www.lowrisc.org)
History of RISC-V LLVM
LLVM status

- All work is taking place upstream
- Assembler and codegen support for RV32IMAFDC, RV64IMAFDC
  - Passing 100% of the GCC torture suite, can compile and run SPEC etc
  - Multiple industry teams using for their 32-bit firmware work (though working around some limitations)
- Soon: hard float ABI, TLS, PIC, further testing+benchmarking
- Generated code performance ~90% of GCC, code size about 10-15% larger (but very input dependent)
- Still an “experimental” backend (more on that later)
- Also: LLD, compiler-rt, ...
- How to judge compiler maturity?
LLVM status (continued)

- LLD: Some patches upstream, incomplete (lacks linker relaxation)
- Compiler-rt: Builds for RISC-V and has some target-specific implementations
- LLDB: No work yet
- Rust: bare metal 32-bit and soon 64-bit toolchains available
- Julia, Swift: Not yet as far as I know
- ...
Contributions

Patches from at least:

- AndesTech
- Qualcomm
- Embecosm
- University of Cambridge
- TU Graz
- Google
- BSC
- ...

Code review, comments, feedback, bug reports etc from many more - thanks!
Growing the effort

● Outreach:
  ○ LLVM Weekly www.llvmweekly.org
  ○ RISC-V LLVM backend tutorial www.lowrisc.org/llvm/devmtg18

● More companies getting more involved as the backend matures and is closer to meeting their needs

● Growing the LLVM team at lowRISC CIC
Implementation approach
Performance testing
RISC-V as a compiler target

A whole family of variants to target:

- RV32I, RV32E or RV64I?
- M extension?
- A extension?
- F extension?
- D extension?
- C extension?
- -mabi=ilp32, ilp32e, ilp32f, ilp32d, lp64, lp64f, lp64d (and soon more!)
Targeting RISC-V: Selected topics

- Minimising backend code duplication
  - Avoid duplicating instruction definitions for RV64I
- RV64I is a separate base ISA with a limited set of 32-bit operations
  - Some challenges in efficient code generation here. Partial type legality as a solution?
- Lack of conditional move
Selected topics: Supporting the compressed instruction set
Selected topics: Supporting the RISC-V memory model
Example: cmpxchg

```assembly
define void @cmpxchg(i32* %ptr, i32 %cmp, i32 %val) {
    %res = cmpxchg i32* %ptr, i32 %cmp,
         i32 %val monotonic monotonic
    ret void
}

.loophead:
    lr.w a3, (a0)
    bne a3, a1, .loopexit

.looptail:
    sc.w a4, a2, (a0)
    bnez a4, .loophead

.loopexit:
    ret
```
Example: cmpxchg

i8

slli a3, a0, 3
andi a3, a3, 24
addi a4, zero, 255
sll a4, a4, a3
andi a2, a2, 255
sll a2, a2, a3
andi a1, a1, 255
sll a1, a1, a3
andi a0, a0, -4

.loophead:
  lr.w a3, (a0)
  and a5, a3, a4
  bne a5, a1, .loopexit

.looptail:
  xor a5, a3, a2
  and a5, a5, a4
  xor a5, a3, a5
  sc.w a5, a5, (a0)
  bnez a5, .loophead

.loopexit:
  ret

How to meet the architectural forward progress guarantee?
Selected topics: Testing and fuzzing
Targeting RISC-V: Selected topics

- Minimising backend code duplication
  - Avoid duplicating instruction definitions for RV64I
- RV64I is a separate base ISA with a limited set of 32-bit operations
  - Some challenges in efficient code generation here. Partial type legality as a solution?
- Lack of conditional move
Roadmap

- lowRISC has a long-term commitment to developing and maturing this toolchain
- End of March milestone:
  - PIC, TLS, initial hard float ABI support, MC layer improvements
- Middle of year
  - Can compile Linux kernel, testing across larger corpus (e.g. Linux userspace)
- Afterwards (relative prioritisation TBD)
  - Further performance, code size improvements. LLDB, LLD, sanitizers, ...
- LLVM release schedule, experimental vs standard targets
Further work / collaboration opportunities

- Vector instruction set support
- Bitmanip, other proposed extensions
- RISC-V based accelerators (see: MLIR)
- Further code compression analysis and optimisations
- Zfinx, new embedded ABIs
- Standards, specs. E.g. embedded metadata in ELF
- Lowering the barrier for adding new extensions
- Security extensions (see: previous lowRISC tagged memory work, recent ARM proposals)
- Achieving a feedback loop between application engineers, compiler authors, microarchitects, ISA spec authors etc
- Custom compressed instruction set generation
End

- lowRISC is hiring! lowrisc.org/jobs - 7 positions (both hardware and software devs)
- Want to get started hacking on LLVM? See my tutorial and coding lab http://lowrisc.org/llvm/devmtq18 (presented at the LLVM Dev Meeting 2018)
- Questions?

Contact: asb@lowrisc.org
Overflow / reference
Describing an instruction: ADD

Use the TableGen domain-specific language.

See lib/Target/RISCV/RISCVInstr Info.td

```python
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} = 0b00000000; /*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, $rs2";
}
```
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add\t$rd, $rs1, $rs2";
}
Describing an instruction: ADD

Assembly parsing / printing

```python
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} = 0b00000000; /*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.

```java
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.

```python
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">;
```
Instruction selection patterns: 
add(i)

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

def : Pat<(add GPR:$rs1, simm12:$imm12),
(ADDI GPR:$rs1, simm12:$imm12)>;
```
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.
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.
Selected topics: 
Linker relaxation
Selected topics:
ABI lowering