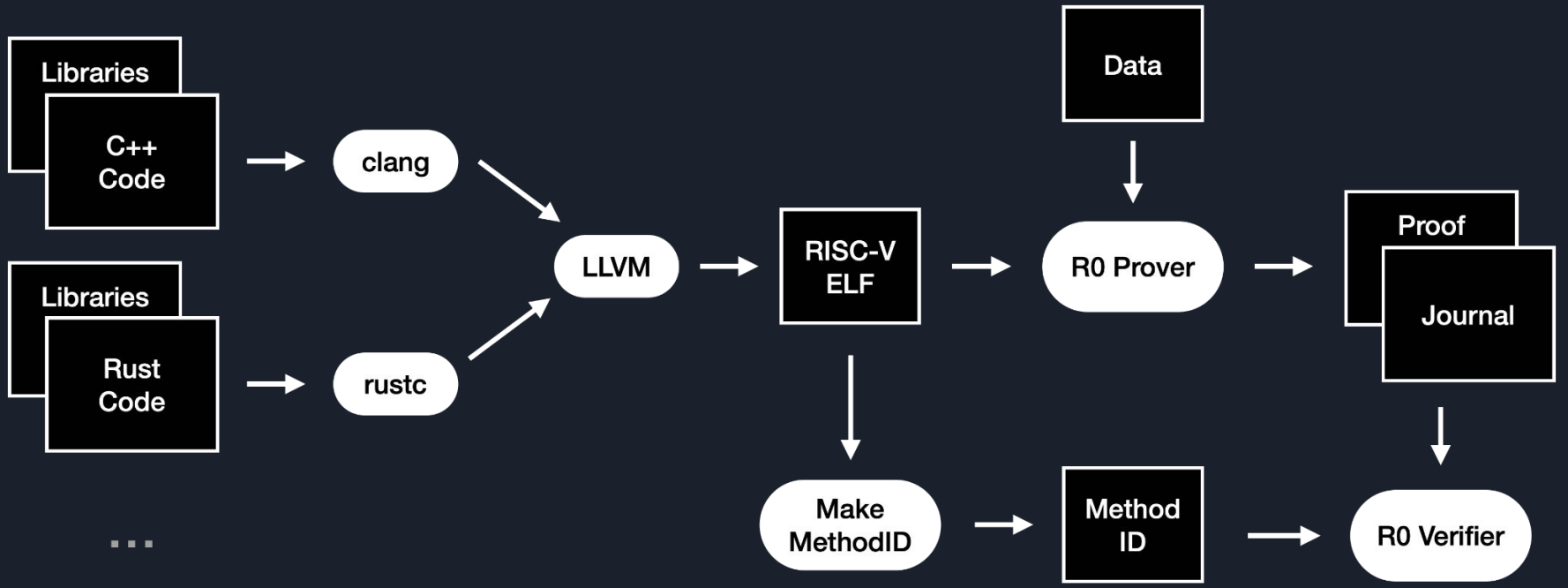
ZIRGEN: MLIR-based compiler for zk-STARKs

Frank Laub
RISC Zero

What is RISC Zero

- An open source (Apache 2) zkVM, including a working prover and verifier
- Implemented via a zk-STARK proof system
- Which emulates the RISC-V processor ISA

ZKP adoption thru traditional developer toolchains



Why ZIRGEN?

- Multiple complex circuits (RISC-V, SHA256, FFPU)
- Rapidly changing proving technology (Plookup, Hyperplonk, …)
- Optimizations at different levels
- Multiple hardware backends (CPU, GPU, FPGA, ASICs)
- Multiple abstractions for HW (CUDA, Metal, WebGPU, OpenCL, …)

^^^

Sounds like a we need a compiler!

What is ZIRGEN?

- C++-based eDSL for constructing arithmetic circuits
- MLIR stack for optimizing circuits and generating constraints
- Hardware-specific code generation
- Support for STARKs, PLONK, and Plookup

ZK Circuits

- Applications
  - Fibonacci
  - Verifier (recursion)*
- Virtual Machines
  - RISC-V
  - MIPS*
  - WASM*
- Accelerators
  - SHA-256
  - Poseidon*
  - BigInt*
  - FFPU (Finite field processing unit)

* Not yet implemented

Compiler approach to circuit generation

- Frontends
  - eDSL
  - DSL
  - Translators
- Passes
  - Canonicalization
  - CSE (Common Subexpression Elimination)
  - DCE (Dead Code Elimination)
  - MakePolynomial
  - ComputeDegree
- Backends
  - C++
  - Rust
  - CUDA
  - FFPU
  - WASM

Compiler pipeline

- fib.cpp (eDSL)
- Zirgen dialect
- Generated source code
  - Step functions
    - step_exec.cpp
    - step_verify_bytes.cpp
    - step_verify_mem.cpp
    - step_accum.cpp
  - Polynomials
    - poly_fp.cpp
    - poly_ext.rs

MLIR: Multi-Level Intermediate Representation

“A collection of modular and reusable software components that enables the progressive lowering of operations, to efficiently target hardware in a common way.”

Google Brain - MLIR team

MLIR: Multi-Level Intermediate Representation

- LLVM IR v2: Lessons learned
- IR matches level of abstraction
- Enables transformations
- Optimizations at appropriate abstraction level
- LIT: LLVM Integrated Tester
- Development workflow
- TableGen

```
def EqualZeroOp : ZirgenOp<"eqz", [IsEval]> {
 let summary = "Require a number to be equal to zero, or fail";
 let arguments = (ins Val:$in);
 let assemblyFormat = [{ $in `:` type($in) attr-dict }];
 let hasCanonicalizer = 1;
}
```

eDSL: embedded Domain Specific Language

- Module: container for components
- Component: like a function, has inputs/outputs and a body
- Val: generic value type, parameters: field prime, extension size
- Register: a named value
- Buffer: an array of values
- NONDET: Non-deterministic block
- IF: Conditional block

Zirgen Dialect

- Types
  - Val
  - Constraint
  - Buffer
- Attributes
  - Tap
  - Polynomial
- Ops
  - ConstOp
  - NondetOp
  - IfOp
  - AllocOp
  - BackOp
  - SliceOp
  - GetOp
  - SetOp
  - GetGlobalOp
  - SetGlobalOp

- Ops (cont.)
  - EqualZeroOp
  - BarrierOp
  - UnaryOp
  - BinaryOp
  - IsZeroOp
  - BitAndOp
  - FailOp
  - TrueOp
  - AndEqzOp
  - AndCondOp
  - ExternOp

eDSL: C++ example

Fibonacci sequence

```
Module module;
module.addFunc<3>("fib", {const_buf(3), global_buf(1), mut_buf(1)},
 [](Buffer ctrl, Buffer out, Buffer data) {
 Register val = data[0];
 IF(ctrl[0]) { // init
   val = 1;
 }
 IF(ctrl[1]) { // body
   val = BACK(1, val) + BACK(2, val);
 }
 IF(ctrl[2]) { // fini
   out[0] = val;
 }
});
```

zirgen dialect

```
func.func @fib(%ctrl: !zirgen.buffer< 3, constant>,
               %out: !zirgen.buffer< 1, global>,
               %data: !zirgen.buffer< 1, mutable>) ->
              !zirgen.val<default> {
 %0 = zirgen.const  1
 %1 = zirgen.get  %ctrl[0] back  0 : <3, constant>
 zirgen.if  %1 : <default> {
   zirgen.set  %data : <1, mutable>[ 0] = %0 : <default>
 }
 %2 = zirgen.get  %ctrl[1] back  0 : <3, constant>
 zirgen.if  %2 : <default> {
   %7 = zirgen.get  %data[0] back  1 : <1, mutable>
   %8 = zirgen.get  %data[0] back  2 : <1, mutable>
   %9 = zirgen.add  %7 : <default>,  %8 : <default>
   zirgen.set  %data : <1, mutable>[ 0] = %9 : <default>
 }
 %3 = zirgen.get  %ctrl[2] back  0 : <3, constant>
 zirgen.if  %3 : <default> {
   %7 = zirgen.get  %data[0] back  0 : <1, mutable>
   zirgen.set_global  %out : <1, global>[ 0] = %7 : <default>
 }
 %4 = zirgen.sub  %0 : <default>,  %3 : <default>
 %5 = zirgen.add  %1 : <default>,  %2 : <default>
 %6 = zirgen.add  %5 : <default>,  %3 : <default>
 return  %4 : !zirgen.val<default>
}
```

Pass: MakePolynomial

Translate execution step function into a polynomial constraint

```
func.func @fib(%arg0: !zirgen.buffer<3, constant>,
               %arg1: !zirgen.buffer<1, global>,
               %arg2: !zirgen.buffer<1, mutable>) ->
                !zirgen.val<default> {
 %0 = zirgen.const 1
 %1 = zirgen.get %arg0[0] back 0 : <3, constant>
 zirgen.if %1 : <default> {
   zirgen.set %arg2 : <1, mutable>[0] = %0 : <default>
 }
 %2 = zirgen.get %arg0[1] back 0 : <3, constant>
 zirgen.if %2 : <default> {
   %7 = zirgen.get %arg2[0] back 1 : <1, mutable>
   %8 = zirgen.get %arg2[0] back 2 : <1, mutable>
   %9 = zirgen.add %7 : <default>, %8 : <default>
   zirgen.set %arg2 : <1, mutable>[0] = %9 : <default>
 }
 %3 = zirgen.get %arg0[2] back 0 : <3, constant>
 zirgen.if %3 : <default> {
   %7 = zirgen.get %arg2[0] back 0 : <1, mutable>
   zirgen.set_global %arg1 : <1, global>[0] = %7 : <default>
 }
 %4 = zirgen.sub %0 : <default>, %3 : <default>
 %5 = zirgen.add %1 : <default>, %2 : <default>
 %6 = zirgen.add %5 : <default>, %3 : <default>
 return %4 : !zirgen.val<default>
}
```

```
func.func @fib(%arg0: !zirgen.buffer<3, constant>,
               %arg1: !zirgen.buffer<1, global>,
               %arg2: !zirgen.buffer<1, mutable>) ->
               !zirgen.constraint {
 %0 = zirgen.const 1
 %1 = zirgen.true
 %2 = zirgen.get %arg0[0] back 0 : <3, constant>
 %3 = zirgen.get %arg2[0] back 0 : <1, mutable>
 %4 = zirgen.sub %3 : <default>, %0 : <default>
 %5 = zirgen.and_eqz %1, %4 : <default>
 %6 = zirgen.and_cond %1, %2 : <default>, %5
 %7 = zirgen.get %arg0[1] back 0 : <3, constant>
 %8 = zirgen.get %arg2[0] back 1 : <1, mutable>
 %9 = zirgen.get %arg2[0] back 2 : <1, mutable>
 %10 = zirgen.add %8 : <default>, %9 : <default>
 %11 = zirgen.sub %3 : <default>, %10 : <default>
 %12 = zirgen.and_eqz %1, %11 : <default>
 %13 = zirgen.and_cond %6, %7 : <default>, %12
 %14 = zirgen.get %arg0[2] back 0 : <3, constant>
 %15 = zirgen.get_global %arg1[0] : <1, global>
 %16 = zirgen.sub %15 : <default>, %3 : <default>
 %17 = zirgen.and_eqz %1, %16 : <default>
 %18 = zirgen.and_cond %13, %14 : <default>, %17
 return %18 : !zirgen.constraint
}
```

Pass: Canonicalization/CSE

```
func.func @fib(%arg0: !zirgen.buffer< 3, constant >,
               %arg1: !zirgen.buffer< 1, global>,
               %arg2: !zirgen.buffer< 1, mutable>)
->
              !zirgen.constraint {
 %0 = zirgen.true
 %1 = zirgen.const  1
 %2 = zirgen.get  %arg0[0] back  0  : <3, constant >
 %3 = zirgen.true
 %4 = zirgen.get  %arg2[0] back  0  : <1, mutable>
 %5 = zirgen.sub  %4  : <default>,  %1  : <default>
 %6 = zirgen.and eqz  %3,  %5  : <default>
 %7 = zirgen.and cond  %0,  %2  : <default>,  %6
 %8 = zirgen.get  %arg0[1] back  0  : <3, constant >
 %9 = zirgen.true
 %10 = zirgen.get  %arg2[0] back  1  : <1, mutable>
 %11 = zirgen.get  %arg2[0] back  2  : <1, mutable>
 %12 = zirgen.add  %10  : <default>,  %11  : <default>
 %13 = zirgen.get  %arg2[0] back  0  : <1, mutable>
 %14 = zirgen.sub  %13  : <default>,  %12  : <default>
 %15 = zirgen.and eqz  %9,  %14  : <default>
 %16 = zirgen.and cond  %7,  %8  : <default>,  %15
 %17 = zirgen.get  %arg0[2] back  0  : <3, constant >
 %18 = zirgen.true
 %19 = zirgen.get  %arg2[0] back  0  : <1, mutable>
 %20 = zirgen.get global  %arg1[0]  : <1, global>
 %21 = zirgen.sub  %20  : <default>,  %19  : <default>
 %22 = zirgen.and eqz  %18,  %21  : <default>
 %23 = zirgen.and cond  %16,  %17  : <default>,  %22
 %24 = zirgen.sub  %1  : <default>,  %17  : <default>
 %25 = zirgen.add  %2  : <default>,  %8  : <default>
 %26 = zirgen.add  %25  : <default>,  %17  : <default>
 return  %23  : !zirgen.constraint
}
```

```
func.func @fib(%arg0: !zirgen.buffer<3, constant>,
               %arg1: !zirgen.buffer<1, global>,
               %arg2: !zirgen.buffer<1, mutable>) ->
               !zirgen.constraint {
 %0 = zirgen.const 1
 %1 = zirgen.true
 %2 = zirgen.get %arg0[0] back 0 : <3, constant>
 %3 = zirgen.get %arg2[0] back 0 : <1, mutable>
 %4 = zirgen.sub %3 : <default>, %0 : <default>
 %5 = zirgen.and eqz %1, %4 : <default>
 %6 = zirgen.and cond %1, %2 : <default>, %5
 %7 = zirgen.get %arg0[1] back 0 : <3, constant>
 %8 = zirgen.get %arg2[0] back 1 : <1, mutable>
 %9 = zirgen.get %arg2[0] back 2 : <1, mutable>
 %10 = zirgen.add %8 : <default>, %9 : <default>
 %11 = zirgen.sub %3 : <default>, %10 : <default>
 %12 = zirgen.and eqz %1, %11 : <default>
 %13 = zirgen.and cond %6, %7 : <default>, %12
 %14 = zirgen.get %arg0[2] back 0 : <3, constant>
 %15 = zirgen.get global %arg1[0] : <1, global>
 %16 = zirgen.sub %15 : <default>, %3 : <default>
 %17 = zirgen.and eqz %1, %16 : <default>
 %18 = zirgen.and cond %13, %14 : <default>, %17
 return %18 : !zirgen.constraint
}
```

Pass: ComputeDegree

Annotate ops with the degree for a given polynomial

```
func.func @fib(%arg0: !zirgen.buffer< 3, constant >,
               %arg1: !zirgen.buffer< 1, global>,
               %arg2: !zirgen.buffer< 1, mutable>)
->
              !zirgen.constraint {
 %0 = zirgen.const  1
 %1 = zirgen.true
 %2 = zirgen.get  %arg0[0] back  0 : <3, constant >
 %3 = zirgen.get  %arg2[0] back  0 : <1, mutable>
 %4 = zirgen.sub  %3 : <default>,  %0 : <default>
 %5 = zirgen.and eqz  %1, %4 : <default>
 %6 = zirgen.and cond  %1, %2 : <default>,  %5
 %7 = zirgen.get  %arg0[1] back  0 : <3, constant >
 %8 = zirgen.get  %arg2[0] back  1 : <1, mutable>
 %9 = zirgen.get  %arg2[0] back  2 : <1, mutable>
 %10 = zirgen.add  %8 : <default>,  %9 : <default>
 %11 = zirgen.sub  %3 : <default>,  %10 : <default>
 %12 = zirgen.and eqz  %1, %11 : <default>
 %13 = zirgen.and cond  %6, %7 : <default>,  %12
 %14 = zirgen.get  %arg0[2] back  0 : <3, constant >
 %15 = zirgen.get global  %arg1[0] : <1, global>
 %16 = zirgen.sub  %15 : <default>,  %3 : <default>
 %17 = zirgen.and eqz  %1, %16 : <default>
 %18 = zirgen.and cond  %13, %14 : <default>,  %17
 return  %18 : !zirgen.constraint
}
```

```
func.func @fib(%arg0: !zirgen.buffer<3, constant>,
               %arg1: !zirgen.buffer<1, global>,
               %arg2: !zirgen.buffer<1, mutable>) ->
               !zirgen.constraint attributes {deg = 2} {
 %0 = zirgen.const 1 {deg = 0}
 %1 = zirgen.true {deg = 0}
 %2 = zirgen.get %arg0[0] back 0 : <3, constant> {deg = 1}
 %3 = zirgen.get %arg2[0] back 0 : <1, mutable> {deg = 1}
 %4 = zirgen.sub %3 : <default>, %0 : <default> {deg = 1}
 %5 = zirgen.and_eqz %1, %4 : <default> {deg = 1}
 %6 = zirgen.and_cond %1, %2 : <default>, %5 {deg = 2}
 %7 = zirgen.get %arg0[1] back 0 : <3, constant> {deg = 1}
 %8 = zirgen.get %arg2[0] back 1 : <1, mutable> {deg = 1}
 %9 = zirgen.get %arg2[0] back 2 : <1, mutable> {deg = 1}
 %10 = zirgen.add %8 : <default>, %9 : <default> {deg = 1}
 %11 = zirgen.sub %3 : <default>, %10 : <default> {deg = 1}
 %12 = zirgen.and_eqz %1, %11 : <default> {deg = 1}
 %13 = zirgen.and_cond %6, %7 : <default>, %12 {deg = 2}
 %14 = zirgen.get %arg0[2] back 0 : <3, constant> {deg = 1}
 %15 = zirgen.get_global %arg1[0] : <1, global> {deg = 0}
 %16 = zirgen.sub %15 : <default>, %3 : <default> {deg = 1}
 %17 = zirgen.and_eqz %1, %16 : <default> {deg = 1}
 %18 = zirgen.and_cond %13, %14 : <default>, %17 {deg = 2}
 return {deg = 2} %18 : !zirgen.constraint
}
```

Codegen: C++

```
Fp step_exec(size_t steps, size_t cycle, Fp** args) {
 size_t mask = steps - 1;
 Fp x0(1);
 auto x1 = args[0][0 * steps + ((cycle - 0) & mask)];
 if (x1 != 0) {
   auto& reg = args[2][0 * steps + cycle];
   reg = x0;
 }
 auto x2 = args[0][1 * steps + ((cycle - 0) & mask)];
 if (x2 != 0) {
   auto x3 = args[2][0 * steps + ((cycle - 1) & mask)];
   auto x4 = args[2][0 * steps + ((cycle - 2) & mask)];
   auto x5 = x3 + x4;
   auto& reg = args[2][0 * steps + cycle];
   reg = x5;
 }
 auto x6 = args[0][2 * steps + ((cycle - 0) & mask)];
 if (x6 != 0) {
   auto x7 = args[2][0 * steps + ((cycle - 0) & mask)];
   args[1][0] = x7;
 }
 auto x8 = x0 - x6;
 return x8;
}
```

Performance

- CPU: 25 Khz (cycles/sec)
- GPU: 50 Khz (cycles/sec)
- EOY target: 500 Khz (cycles/sec)
- Recursion: 37s on NVIDIA RTX A5000

Questions?

frank@risczero.com

We are:

- Hiring
  - https://risczero.com/careers
- Hacking
  - https://github.com/risc0
- Hanging out
  - @risczero on Twitter
  - https://discord.gg/risczero

MLIR: Operation