Assignment 3. Classical Protocol Engineering Practice

0. 基础信息

Item	Description
Name	yueawang
Discord Account	Yue#1946
Study Group	Team3
Assignment	Classical Protocol Engineering Practice
GitHub Rep	https://github.com/yueawang/miscellaneous

The following function is known, with 3 variables x, y, z.

```
def f(x, y, z):
  if x == 1:
    return y*z
  return 2y - z
}
```

prover owns private variables x, y, z with values (x = 1, y = 2, z = 5), please choose one open source libraries or toolkits of Groth16, Plonk, Stark protocol, and use one of the protocols to complete the verifiable computation of function `f(x,y,z)` as above.

Engineering Assignment Requirements:

1. please briefly describe the use of the protocol code base or framework
2. please explain the implementation idea in steps
3. if there is an output of the calculation step, please give a screenshot of the result
4. please provide the project practice repository, specify the commit hash

1. 基础框架简介

本次作业采用Halo2框架，Halo2 是 ECC 公司在 Halo 的基础上，使用 Plonk 对 Halo 进行升级改造，充分利用了 Plonk 的特性，比如 custom gate，Plonkup 等，使用 Halo2 开发零知识证明电路更加高效和方便。

2. 工程简介

2.1 ZK电路设计

根据被验证函数的特点，将if条件语句转换成表达式：

$$f(x, y, z) \equiv (x == 1) * y * z + (!(x == 1)) * (2 * y - z)$$

进一步分解如下：

$$\begin{aligned} xeq1 &= (x == 1) && (1) \\ xneq1 &= \ !xeq1 && (2) \\ yz &= y * z && (3) \\ 2ysubz &= 2 * y - z && (4) \\ f &= xeq1 * yz + xneq1 * 2ysubz && (5) \end{aligned}$$

为了计算步骤1～5，需要两个子约束函数：

- is_zero()：用于计算x==1，即is_zero(x-1)。

$$is\_zero(a) \equiv 1 - a * a^{-1}$$

- plonk()：基础plonk表达式，用来计算加法/乘法运算，即：

$$sa * a + sb * b + sm * a * b + sq - sc * c \equiv 0$$

-

于是总体思路如下：

$$\begin{aligned} &xsub1 = plonk(x - 1) \\ &xeq1 = is\_zero(xsub1) \\ &xneq1 = plonk(1 - xeq1) \\ &yz = plonk(y * z) \\ &2ysubz = plonk(2 * y - z) \\ &f = plonk(xeq1 * yz + xneq1 * 2ysubz) \end{aligned}$$

2.2 ZK电路实现

实现上基于Halo2的示例代码，实现了两个基本电路操作：

```
trait ConditionInstructions<F: FieldExt>: Chip<F> {
    /// Returns `x == 0`.
    fn is_zero(&self, layouter: impl Layouter<F>, a: Self::Num) ->
Result<Self::Num, Error>;
}

trait CalculateInstructions<F: FieldExt>: Chip<F> {
    /// 2*a - b
    fn calculate_2amb(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    // a - 1
    fn calculate_am1(&self, layouter: impl Layouter<F>, a: Self::Num) ->
Result<Self::Num, Error>;

    // a * b
    fn calculate_ab(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    // a + b
    fn calculate_a_plus_b(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    // !a
    fn calculate_not_a(&self, layouter: impl Layouter<F>, a: Self::Num)
        -> Result<Self::Num, Error>;
}
```

ConditionInstructions计算is_zero，CalculateInstructions则基于Plonk计算加法乘法，由总电路FuncInstructions进行综合操作：

```
impl<F: FieldExt> FuncInstructions<F> for FieldChip<F> {
    fn conditional_calculate(
        &self,
        layouter: &mut impl Layouter<F>,
        a: <Self as FuncInstructions<F>>::Num,
        b: <Self as FuncInstructions<F>>::Num,
        c: <Self as FuncInstructions<F>>::Num,
    ) -> Result<<Self as FuncInstructions<F>>::Num, Error> {
        let x_sub_1 = self.calculate_am1(layouter.namespace(|| "a - 1"), a)?;
        let x_eq_1 = self.is_zero(layouter.namespace(|| "a-1 == 0"),
x_sub_1.clone())?;
        let x_not_eq_1 = self.calculate_not_a(layouter.namespace(|| "a-1 !=
0"), x_eq_1.clone())?;
        let v1 = self.calculate_ab(layouter.namespace(|| "y*z"), b.clone(),
c.clone())?;
```

```
        let v2 = self.calculate_2amb(layouter.namespace(|| "2y-z"), b.clone(),
c.clone())?;
        let v3 = self.calculate_ab(layouter.namespace(|| "(x==1) * y*z"),
x_eq_1.clone(), v1)?;
        let v4 = self.calculate_ab(layouter.namespace(|| "(x!=1) * (2y-z)"),
x_not_eq_1, v2)?;
        self.calculate_a_plus_b(layouter.namespace(|| "f(x, y, z)"), v3, v4)
    }
}
```

很明显就是分别约束上面分解后的步骤1～5。

具体实现上就是简单翻译设计思路，比如is_zero的约束是这样的。

```
meta.create_gate("a == 0", |meta| {
  let a = meta.query_advice(advice[0], Rotation::cur());
  let a_inv = meta.query_advice(advice[1], Rotation::cur());
  let a_is_zero = meta.query_advice(advice[0], Rotation::next());
  let s_cond = meta.query_selector(s_cond);

  let one = Expression::Constant(F::one());

  let is_zero_expression = one - a.clone() * a_inv.clone();

  // This checks `value_inv ≡ value.invert()` when `value` is not
  // zero: value ⋅ (1 - value ⋅ value_inv)
  let poly1 = a * is_zero_expression.clone();
  // This checks `value_inv ≡ 0` when `value` is zero:
  // value_inv ⋅ (1 - value ⋅ value_inv)
  let poly2 = a_inv * is_zero_expression.clone();

  vec![
    s_cond.clone() * poly1,
    s_cond.clone() * poly2,
    s_cond.clone() * (a_is_zero - is_zero_expression),
  ]
});
```

而Plonk的约束已知是固定的，各种加减乘法都只是对Plonk系数的调整，比如计算乘法$c = a * b$，翻译到Plonk约束：

$$0 * a + 0 * b + 1 * ab + 0 - c = 0$$

```
fn calculate_ab(
  &self,
  mut layouter: impl Layouter<F>,
  a: Self::Num,
  b: Self::Num,
) -> Result<Self::Num, Error> {
  let config = self.config();

  layouter.assign_region(
    || "a*b",
    |mut region| {
      a.0.copy_advice(|| "lhs", &mut region, config.a, 0)?;
      b.0.copy_advice(|| "rhs", &mut region, config.b, 0)?;

      let value = a.0.value().and_then(|a| b.0.value().map(|b| *a * *b));

      let out = region.assign_advice(
        || "out",
        self.config.c,
        0,
        || value,
      )?;

      region.assign_fixed(|| "sa", self.config.sa, 0, ||
Value::known(F::zero()))?;
      region.assign_fixed(|| "sb", self.config.sb, 0, ||
Value::known(F::zero()))?;
      region.assign_fixed(|| "sc", self.config.sc, 0, ||
Value::known(F::one()))?;
      region.assign_fixed(|| "sm", self.config.sm, 0, ||
Value::known(F::one()))?;

      Ok(Number(out))
    },
  )
}
```

其他操作如$(a+b)$，$(a-1)$，$(2*a-b)$等进行类似调整即可。

2.3 测试用例

测试还是基于Halo2的示例代码，使用MockProver进行验证：

```
fn main() {
    use halo2_proofs::dev::MockProver;
    use halo2_proofs::pasta::Fp;

    // ANCHOR: test-circuit
    // The number of rows in our circuit cannot exceed 2^k. Since our example
    // circuit is very small, we can pick a very small value here.
    let k = 4;

    // Prepare the private and public inputs to the circuit!
    let a = 1;
    let b = 2;
    let c = 5;
    let d = foo(a, b, c); // should be 10

    // Instantiate the circuit with the private inputs.
    let circuit = MyCircuit {
        x: Some(Fp::from(a)),
        y: Some(Fp::from(b)),
        z: Some(Fp::from(c)),
    };

    // Arrange the public input. We expose the function result in row 0
    // of the instance column, so we position it there in our public inputs.
    let mut public_inputs = vec![Fp::from(d)];

    // Given the correct public input, our circuit will verify.
    let prover = MockProver::run(k, &circuit, vec!
[public_inputs.clone()]).unwrap();
    assert_eq!(prover.verify(), Ok(()));

    // If we try some other public input, the proof will fail!
    public_inputs[0] += Fp::one();
    let prover = MockProver::run(k, &circuit, vec![public_inputs]).unwrap();
    assert!(prover.verify().is_err());
    // ANCHOR_END: test-circuit
}
```

如果需要生成proof，直接调用Halo2的API即可。

3. 总结

工程代码位于：https://github.com/yueawang/halo2/commit/cbdc9f383e7fbdd98e898920d98bce8948f1a334

运行环境：

```
> Executing task: cargo run --package halo2_proofs --example week2-assignment-
q3 <

    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/examples/week2-assignment-q3`

  Test Passed!
```

本作业工程参考Halo2以及其他zk工程实现了对特定函数计算过程的验证，进一步了解了电路的设计和实现模式，通过测试用例加深了对ZK证明过程的理解。当然不足之处也很多，实现上是简单的2个子电路拼接，没有考虑其他更优化的实现方式，另外可能有潜在的问题没有发现等等。

4. 参考代码

1. https://github.com/zcash/halo2
2. https://github.com/privacy-scaling-explorations/zkevm-circuits