作业2. 256比特长度的加法电路设计

0. 基础信息

Item	Description
Name	yueawang
Discord Account	Yue#1946
Study Group	Team3
Assignment	256比特长度的加法电路设计
GitHub Rep	https://github.com/yueawang/miscellaneous

作业2. 256比特长度的加法电路设计

Q1. 现有两个字符串A，B，其中A, B最长为256个字符，每个字符可为0或1。请使用逻辑门(形如：z = x op y)设计一个电路，实现字符串加法，A+B = C。请简述设计思路。（语言不限，建议使用circom、aleo）

Q2. 对Q1中实现的电路，请使用zk开发库进行该电路的可验证计算，即prover创建proof，verifier验证。

> 建议：参考circomlib进行电路设计，使用snarkjs完成科研中计算

作业要求：

1. 请简述设计思路
2. 请提供实现代码仓库和commit hash
3. 计算步骤有输出，请给出结果截图

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

二进制字符串加法主要是带进位的bit加法。伪代码形式如下：

```
def bin_add(a[256], b[256]):
o[256]
cr = 0
for i in range(0, 256):
  o[i] = (a[i] + b[i] + cr) & 0x1
  cr = (a[i] + b[i] + cr) > 1
assert cr == 0 #last cr should be zero.
return o
```

假设输入的字符串已经是二进制0/1。那么有个问题在于如何实现`o[i] = a[i] + b[i]`，在电路里，a和b实际是Field Number。如果a=1，b=1，那么a+b==2，输出到o[i]==0的话需要右移或者触法等操作，这里需要设计一个门电路用于bit计算。比较直接的想法是使用异或xor门，由于电路本身不支持xor操作，因此将xor转化成表达式：

$$XOR(a,b) = (a+b) - 2ab$$

因此o[i]的计算没有问题了，只是需要两次XOR，即

```
tmp = XOR(a[i], b[i])
o[i] = XOR(tmp, cr)
```

下面看如何计算cr，重新设计一个电路用来计算 `(a + b + cr) > 1` 是可以的，这里为了复用XOR，将cr写成如下表达式：

```
cr = a*b ^ b*c ^ c*a
```

需要用到一个乘法和XOR，细化一下：

```
tmp0 = a*b
tmp1 = a*cr
tmp2 = b*cr
tmp3 = XOR(tmp0, tmp1)
cr = XOR(tmp2, tmp3)
```

因此为了计算二进制，需要两个子约束函数：

- a_mul_b()：用于计算a*b。
- a_xor_b()：用于计算a^b.

幸运的是这两者都很容易用基础plonk表达式表达，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：

```
bin_add(a[256], b[256]):
  cr = 0
```

```
  o[256] = 0
  for i in 0..256 {
    tmp = a_xor_b(a[i], b[i])
    o[i] = a_xor_b(tmp, cr)

    tmp0 = a*b
    tmp1 = a*cr
    tmp2 = b*cr
    tmp3 = XOR(tmp0, tmp1)
    cr = XOR(tmp2, tmp3)
  }
  return o
```

2.2 ZK电路实现

实现上基于Halo2的示例代码，用plonk实现两个基本电路操作，已知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，比如乘法$c = a * b$，翻译到Plonk约束：

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

即`c==a*b`，然后xor：

$$1 * a + 1 * b - 2 * ab + 0 - c = 0$$

即`c==(a+b)-2ab`。具体配置代码如下：

```
impl<F: FieldExt> CalculateInstructions<F> for BasicPlonkChip<F> {
    type Num = Number<F>;

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

        layouter.assign_region(
            || "calc 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(|| "sq", self.config.sq, 0, ||
Value::known(F::zero()))?;
                region.assign_fixed(|| "s(a * b)", self.config.sm, 0, ||
Value::known(F::one()))?;

                Ok(Number(out))
            },
        )
    }

    /// XOR(a, b) == a + b - 2ab
    fn calculate_a_xor_b(
        &self,
        mut layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error> {
        let config = self.config();

        layouter.assign_region(
            || "calc 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 - (F::one() + F::one()) * *a * *b)
                });

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

                region.assign_fixed(|| "sa", self.config.sa, 0, ||
Value::known(F::one()))?;
                region.assign_fixed(|| "sb", self.config.sb, 0, ||
Value::known(F::one()))?;
                region.assign_fixed(|| "sc", self.config.sc, 0, ||
Value::known(F::one()))?;
                region.assign_fixed(|| "sq", self.config.sq, 0, ||
Value::known(F::zero()))?;
                region.assign_fixed(
                    || "s(a * b)",
```

```
                    self.config.sm,
                    0,
                    || Value::known(-(F::one() + F::one())),
                )?;

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

有了乘法和异或两个组件，最后由总电路FuncInstructions进行综合操作：

```
fn bit_calculate(
        &self,
        layouter: &mut impl Layouter<F>,
        a: &Vec<<Self as FuncInstructions<F>>::Num>,
        b: &Vec<<Self as FuncInstructions<F>>::Num>,
    ) -> Result<Vec<<Self as FuncInstructions<F>>::Num>, Error> {
        let mut output = vec![];

        let mut a_bit = &a[0];
        let mut b_bit = &b[0];
        let mut a_plus_b = self.calculate_a_xor_b(
            layouter.namespace(|| "c0 = a0 xor b0"),
            a_bit.clone(),
            b_bit.clone(),
        )?;
        output.push(a_plus_b);
        let mut cr = self.calculate_a_mul_b(
            layouter.namespace(|| "cr0 = a0*b0"),
            a_bit.clone(),
            b_bit.clone(),
        )?;

        for i in 1..self.config.size {
            a_bit = &a[i];
            b_bit = &b[i];

            a_plus_b = self.calculate_a_xor_b(
                layouter.namespace(|| "a xor b"),
                a_bit.clone(),
                b_bit.clone(),
            )?;
            let a_b_cr = self.calculate_a_xor_b(
                layouter.namespace(|| "o[i] = a[i]+b[i]+cr[i-1]"),
                a_plus_b.clone(),
                cr.clone(),
```

```
            )?;
            output.push(a_b_cr.clone());

            let _ab =
                self.calculate_a_mul_b(layouter.namespace(|| "a*b"),
a_bit.clone(), b_bit.clone())?;
            let _ac =
                self.calculate_a_mul_b(layouter.namespace(|| "a*cr"),
a_bit.clone(), cr.clone())?;
            let _bc =
                self.calculate_a_mul_b(layouter.namespace(|| "b*cr"),
cr.clone(), b_bit.clone())?;

            let cr_0 = self.calculate_a_xor_b(
                layouter.namespace(|| "a*b ^ a*cr"),
                _ab.clone(),
                _ac.clone(),
            )?;
            cr = self.calculate_a_xor_b(
                layouter.namespace(|| "cr = a*b ^ a*cr ^ b*cr"),
                cr_0.clone(),
                _bc.clone(),
            )?;
        }
        Ok(output)
    }
```

很明显就是总体思路的具体实现。

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;

    fn i_to_b(x: [u64; 4]) -> Vec<Fp> {
        let mut o = vec![];
        for i in 0..64 {
            o.push(Fp::from((x[0] & (1u64 << i) != 0) as u64));
        }

        for i in 0..64 {
            o.push(Fp::from((x[1] & (1u64 << i) != 0) as u64));
        }

        for i in 0..64 {
            o.push(Fp::from((x[2] & (1u64 << i) != 0) as u64));
```

```
        }

        for i in 0..64 {
            o.push(Fp::from((x[3] & (1u64 << i) != 0) as u64));
        }

        o
    }

    // 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 = 20;

    // Prepare the private and public inputs to the circuit!
    let a = 7;
    let b = 1;

    // Instantiate the circuit with the private inputs.
    let circuit = MyCircuit {
        a: Some(i_to_b([a, 0, 0, 0])),
        b: Some(i_to_b([b, 0, 0, 0])),
        // c: Some(i_to_b(a + b)),
    };

    // 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 = i_to_b([a + b, 0, 0, 0]);
    println!("{:?}", public_inputs);

    // Given the correct public input, our circuit will verify.
    let prover = MockProver::run(k, &circuit, vec![public_inputs]).unwrap();
    let err = prover.verify();
    assert_eq!(err, 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/7eeb9a8d6bdf4ebbc828415869985e3e57c90ff8

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，由于时间比较仓促，来不及熟悉circomlib，使用了更熟悉的Halo2作为基础库，同时省略了很多步骤，有很多前置假设没有在电路实现，能够想到的有一下几点：

1. 字符串输入变量的bit(0/1)检查
2. 最终的cr必须为0的溢出检查。
3. 没有考虑其他优化，基于plonk用乘法和异或门是个比较省事的办法，采用更灵活的自定义门效果应该会更好一些。
4. Testcase覆盖不全。

后续有机会再补一下。

4. 参考代码

1. https://github.com/zcash/halo2