0.1 Circom Walkthrough

In this final lecture, we bridge the gap between the theoretical concepts presented in previous lectures and their practical realization using the **Circom** DSL.

> **Definition 0.1. Circom** is a domain-specific language for building arithmetic circuits that can be used to produce zk-SNARK proofs.

Throughout this lecture, we will walk through how concepts like R1CS, witness, trusted setup, and verification keys appear in actual code and practice.

0.1.1 Journey Begins

In the previous lectures, we covered a variety of theoretical concepts: zk-SNARKs, trusted setup, arithmetic circuits, constraints, witnesses, and the Rank-1 Constraint System (R1CS) representation. Now, let's see how these appear in practice.

0.1.2 From Theory to Practice: Circom Basics

We learned that a circuit can represent a complex arithmetic computation over a finite field. Circom allows us to write these circuits in a high-level syntax. To begin, consider the arithmetic circuit $r = x \times y$.

It can be represented in Circom syntax as follows:

```
pragma circom 2.1.6;

template Math() {
    signal input x;
    signal input y;

    signal output r <== x * y;
}

component main = Math();
```

Here, we see how easy it is to define a circuit that takes two inputs x, y and outputs their product r. The `template` defines a reusable circuit component, while `signal input` and `signal output` represent inputs and outputs, respectively. Intermediate signals (without `input` or `output`) are internal primitives within the circuit.

Public vs Private Signals:

Output signals are always public. You may also define public inputs by spec-

ifying them in the main component, for example:

```
component main {public [x]} = Math();
```

This means x is a public input and will appear in the verification context. The order of public signals in the final proof verification step follows the order of their definition inside the template, starting with outputs.

For example, if your circuit looks like this:

```
template Circuit() {
    signal x;

    signal output o2;

    signal input c;
    signal input a;

    signal k1;

    signal input b;

    signal output o1;
}

component main {public [a, b, c]} = Circuit();
```

The order of the public signals that should be passed to the verifier is as follows:

$$(o2, o1, c, a, b)$$

0.1.3 Arguments, Functions, and Vars

Sometimes we need to calculate some values as constants for our circuit. For example, if you want your circuit to be a multi-tool that, based on provided arguments, can work with different cases. For this purpose, we can declare *functions* and *vars* inside the circuit, as shown below:

```
function transformNumber(value) {
    return value ** 2;
}

template Math(padding) {
    signal input x;
    signal input y;

    var elementsNumber = transformNumber(padding);
```

```
    signal b <== x * elementsNumber;

    signal output r <== b * y;
}

component main {public [x]} = Math(12);
```

Here, `var elementsNumber` and the function `transformNumber` are evaluated at compile time. Remember that assignments using `var` and functions do not produce constraints by themselves. Only <==, ==>, or === and actual arithmetic on signals produce constraints reflected in R1CS.

Remark. Sometimes, one needs to perform operations like division or non-quadratic multiplication on the signal. For this purpose, you can use the --> and <-- notations to compute the values "out-of-circuit". For example:

```
template Math() {
    signal input x;
    signal input y;

    signal b <-- x / y;

    signal output r <== b * y;
}

component main = Math();
```

In this case, no constraints are generated with the x input, and it does not even participate in the witness directly.
Notice! This is the main difference between Circom and other languages. When you write a function evaluation $y = f(x)$ in any other language (say, Python or Rust), you are specifying the set of instructions to compute y from x (commonly sequentially). In Circom (or in any other R1CS language) you are merely asserting the **correctness** of the computation and therefore of all intermediate computations. This way, if your task would have been to compute $y = \frac{1}{x}$, you could simply ask y to be the result of the division (that can be computed out of circuit) and then asserting that $x \times y = 1$ with $x \neq 0$. This way, you are not writing the division itself, but the constraints that the division should satisfy.

0.1.4 Theoretical Recap: Using the Learned Concepts

Next, let us apply the learned concepts to a more complex examples. We start with the `if` statement logic.

Example. Recall the complex example we analyzed in earlier lectures:

```
def r(x1: bool, x2: F, x3: F) -> F:
    return x2 * x3 if x1 else x2 + x3
```

This can be represented as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3).$$

We also had the additional constraint $x_1 \times (1 - x_1) = 0$ to ensure x_1 is binary. The resulting system of constraints was:

$$x_1 \times x_1 = x_1 \tag{1}$$
$$x_2 \times x_3 = \mathsf{mult} \tag{2}$$
$$x_1 \times \mathsf{mult} = \mathsf{selectMult} \tag{3}$$
$$(1 - x_1) \times (x_2 + x_3) = r - \mathsf{selectMult} \tag{4}$$

It took us quite some time to understand and come up with the constraint system, which can be visualized as follows:

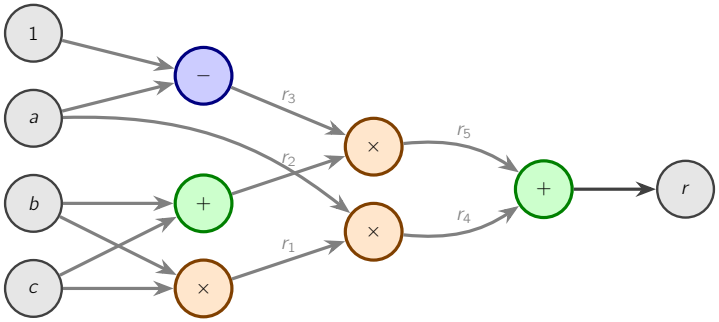

Figure 0.1: Example of a circuit evaluating the `if` statement logic.

The inputs can be directly transformed into signals like below:

```
template Math() {
    signal output r;

    signal input x1;

    signal input x2;
    signal input x3;
```

```
}
```

In our case, we have an additional output signal, so we can "return" it from the circuit. Now, let's compare the mathematical and Circom representations.

Mathematical Constraints:

$x_1 \times x_1 = x_1$

$x_2 \times x_3 = \text{mult}$

$x_1 \times \text{mult} = \text{selectMult}$

$(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult}$

Circom Representation:

```
x1 * x1 === x1;
signal mult <== x2 * x3;
signal selectMult <== x1 * mult;
(1 - x1) * (x2 + x3) + selectMult ==> r;
```

As we can see, the translation from math to Circom is straightforward. We have used *signals* for constraint definitions.

Remark. If you wish to follow along with the explanations in the following chapters:

1. Clone the repository https://github.com/ZKDL-Camp/hardhat-zkit-template.
2. Run `npm install` to install dependencies and `npx hardhat zkit make` to compile Circom circuits and generate the necessary artifacts.

0.1.5 From R1CS to Proof Generation

Now, let us break down everything that is happening during the proof generation process. After compilation, you will find the following files in the `zkit/artifacts/circuits` folder (starting from the project root):

- `.r1cs` file: The Rank-1 Constraint System representation of the circuit.
- `.wasm and *.js` files: The code to compute the witness from the given inputs.
- `.zkey` file: Proving keys after the trusted setup.
- `.sym` file: Symbolic reference for signals.

R1CS File. Let us start with the `.r1cs` file. In **??**, we defined the following coefficient vectors (in simple constraints) for our task:

$$\mathbf{a}_1 = (0,0,1,0,0,0,0) \quad \mathbf{b}_1 = (0,0,1,0,0,0,0) \quad \mathbf{c}_1 = (0,0,1,0,0,0,0)$$
$$\mathbf{a}_2 = (0,0,0,1,0,0,0) \quad \mathbf{b}_2 = (0,0,0,0,1,0,0) \quad \mathbf{c}_2 = (0,0,0,0,0,1,0)$$
$$\mathbf{a}_3 = (0,0,1,0,0,0,0) \quad \mathbf{b}_3 = (0,0,0,0,0,1,0) \quad \mathbf{c}_3 = (0,0,0,0,0,0,1)$$
$$\mathbf{a}_4 = (1,0,-1,0,0,0,0) \quad \mathbf{b}_4 = (0,0,0,1,1,0,0) \quad \mathbf{c}_4 = (0,1,0,0,0,0,-1)$$

On the other hand, using the test from the `test/Math.witness.test.ts` file and reading the R1CS file, we can see:

test/Math.witness.test.ts

```
expect(constraint1[0]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n,
  ↪ 0n ]);
expect(constraint1[1]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n,
  ↪ 0n ]);
expect(constraint1[2]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n,
  ↪ 0n ]);

expect(constraint2[0]).to.deep.equal([ 0n, 0n, 0n,
  ↪ babyJub.F.negone, 0n, 0n, 0n ]);
expect(constraint2[1]).to.deep.equal([ 0n, 0n, 0n, 0n, 1n, 0n,
  ↪ 0n ]);
expect(constraint2[2]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n,
  ↪ babyJub.F.negone, 0n ]);

expect(constraint3[0]).to.deep.equal([ 0n, 0n, babyJub.F.negone,
  ↪ 0n, 0n, 0n, 0n ]);
expect(constraint3[1]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n, 1n,
  ↪ 0n ]);
expect(constraint3[2]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n, 0n,
  ↪ babyJub.F.negone ]);

expect(constraint4[0]).to.deep.equal([ babyJub.F.negone, 0n, 0n,
  ↪ 0n, 0n, 0n, 0n ]);
expect(constraint4[1]).to.deep.equal([ 0n, 0n, 0n, 1n, 0n, 0n,
  ↪ 0n ]);
expect(constraint4[2]).to.deep.equal([ 0n, babyJub.F.negone, 0n,
  ↪ 0n, 0n, 0n, 0n ]);
```

Mostly, the structure generated by Circom aligns with what we had devised, except for the last constraint. The difference occurs because of Circom's optimization to make proof generation and verification more efficient.

Now, let's take a closer look at how the witness is computed. In **??**, we had:

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \mathsf{mult}, \mathsf{selectMult})$$

Given the inputs: $x_1 = 1, x_2 = 3, x_3 = 4$, we can quickly do the math and find out that the actual witness should look like this: $\mathbf{w} = (1, 12, 1, 3, 4, 12, 12)$ based on:

$$\mathsf{mult} = 3 \times 4 = 12$$
$$\mathsf{selectMult} = 1 \times 12 = 12$$
$$r = 1 \times (3 \times 4) + (1 - 1) \times (3 + 4) = 12 + 0 = 12$$

Indeed, it aligns with the test from `test/Math.witness.test.ts`:

test/Math.witness.test.ts

```
expect(witness[0]).to.equal(1n);
expect(witness[1]).to.equal(12n); // r
expect(witness[2]).to.equal(1n);  // x1
expect(witness[3]).to.equal(3n);  // x2
expect(witness[4]).to.equal(4n);  // x3
expect(witness[5]).to.equal(12n); //
  ↪ mult
expect(witness[6]).to.equal(12n); //
  ↪ selectMult
```

The initial 1 in the witness is a constant to facilitate the usage of constants inside the circuit. This corresponds to the fact that $w_0 = 1$ is often used to handle constant terms in R1CS.

The Circom also provides a named representation of all witness elements, which is stored in the `.sym` file and looks as follows:

.sym file for $x_1 ? x_2 \times x_3 : x_2 + x_3$

```
1,1,0,main.r
2,2,0,main.x1
3,3,0,main.x2
4,4,0,main.x3
5,5,0,main.mult
6,6,0,main.selectMult
```

It not only tells us the names of all signals, but also includes information about the optimized signals.

Recall the previous example where we used division and `<--` to store it in the intermediate signal. The generated `.sym` file would look like this:

.sym file for b <-- $x/y, r$ <== by

```
1,1,0,main.r
2,-1,0,main.x
3,2,0,main.y
4,3,0,main.b
```

As we can see, the -1 was added to the signal x to indicate that it is not used in the witness.

Also, according to the documentation of Circom, there are three levels of optimization.

In the `2.1.9` version of Circom, the default optimization was `O2`, but it was lowered to `O1` in following versions because `O2` was too aggressive leading to vulnerable circuits.

Remark. In addition, all linear constraints are optimized on the O2 optimization.

Now, let's examine what the third column in the `.sym` file means.
Consider the following circuit:

```
template BinaryCheck() {
    signal input x1;

    x1 * x1 === x1;
}

template SelectMult() {
    signal input x1;

    signal input x2;
    signal input x3;

    signal mult <== x2 * x3;

    signal output out <== x1 * mult;
}

template Math() {
    signal output r;

    signal input x1;

    signal input x2;
    signal input x3;

    component binCheck = BinaryCheck();
    binCheck.x1 <== x1;

    component selectMult = SelectMult();
    selectMult.x1 <== x1;
    selectMult.x2 <== x2;
    selectMult.x3 <== x3;

    (1 - x1) * (x2 + x3) + selectMult.out ==> r;
}
```

We split the circuit into three parts, and the `.sym` file will look like this:

.sym file for $x_1 ? x_2 \times x_3 : x_2 + x_3$ with templates

```
1,1,2,main.r
2,2,2,main.x1
3,3,2,main.x2
4,4,2,main.x3
5,-1,0,main.binCheck.x1
6,5,1,main.selectMult.out
7,-1,1,main.selectMult.x1
8,-1,1,main.selectMult.x2
9,-1,1,main.selectMult.x3
10,6,1,main.selectMult.mult
```

As we can see, the third column indicates the locality of the signal. Essentially, it tells us which signals are grouped together under the same template.

Another interesting aspect is the order: 0 represents the first component, 1 represents the second component used during computation, and the last component used is the actual 'main'.

Remark. Pay attention to the difference between the modified `Math` circuit's `.sym` file and the original one. Even though we added more signals (i.e., constraints), they were actually optimized by Circom back to the original state.

And the last column is the full name of the constraint, including the path to where it is defined in the code.

0.1.6 Parallel and Custom Keywords

In Circom, there are two special keywords designed to address specific use cases: `custom` and `parallel`.

The `custom` keyword introduces *custom templates* that do not emit R1CS constraints directly. Instead, they delegate logic to `snarkjs` or other libraries at a later stage. Consequently, `custom` templates **cannot** declare subcomponents or add R1CS constraints within their bodies.

The `custom` keyword is used as follows:

```
pragma circom 2.0.6;
pragma custom_templates;

template custom MyCustomGate() {
    // Custom template's code
    // No R1CS constraints or subcomponents can be desclared here
    // Logic will be handled by snarkjs as a PLONK custom gates
}
```

Remark. At the moment of writing the document the `snarkjs` does not support any custom gates (as stated in their documentation). Also, they can be used only in turbo-PLONK or UltraPlonk schemes. Nevertheless, you can find an example of how they have been used here: https://github.com/zkFHE/circomlib-fhe/tree/main.

Meanwhile, the `parallel` keyword (available from Circom 2.0.8 onward) can be applied at either the template or the component instantiation level to parallelize witness generation for independent computations, thereby accelerating large circuits. Parallelism is *only* applied to the C++ witness generator; it does not affect the constraints themselves.

This keyword will be useful in the structures as below:

```
template parallel ParallelExample(n) {
    signal input in[n];
    signal output out[n];

    // Each iteration is independent, so we can parallelize
    for (var i = 0; i < n; i++) {
        out[i] <== in[i] * 2;
    }
}
```

In summary, you should use the `custom` keyword whenever you want to define a template handled **only** by turbo-PLONK or UltraPlonk schemes. You can also find the exact section in the R1CS binary format where custom gates are stored for later processing by the library in the following link:
https://github.com/iden3/r1csfile/blob/master/doc/r1cs_bin_format.md#custom-gates-list-section-plonk

The `parallel` keyword is helpful when dealing with large or repetitive computations, as it can speed up witness generation. However, in small circuits or wherever computation is inherently sequential (i.e., where the output of one part is the input to another), `parallel` has no effect.

You can find additional examples of the `parallel` keyword usage here: https://github.com/zkFHE/circomlib-fhe/tree/main.

With this, we have covered all the important files generated by Circom.

0.1.7 Generating and Verifying Proofs

Now, it is time to look at proof generation and verification. In this chapter, our main focus will be on the code from `test/Math.circuit.ts`.

To generate a proof, we need to call the `generateProof` method on the circuit object:

```
const proof = await circuit.generateProof(inputs);
```

The actual proof looks like this:

proof.json


```
{
 "proof": {
  "pi_a": [
   "47058017115654770468371195107739881730919574172 70
    76691836744124429204798006 4",
   "14008115995489042379593199896964816349631620264 39
    38305905213597627312056416 7",
   "1"
  ],
  "pi_b": [
   [
    "12538508168416900299033726521685163817792614632 62
     06572440942935413198045466 1",
    "10914283679966848917795247355212516197618338956 68
     23748742390055067503844244 44"
   ],
   [
    "11504632457518572930719312464170675169899321263 87
     39934331914275249663816186 23",
    "15524163713890313070296837080299781036987071183 39
     77274529076703213680571039 14"
   ],
   [
    "1",
    "0"
   ]
  ],
  "pi_c": [
   "26099670053328208608403811624767970928571072694 687
    57252635436475859887989 98",
   "14278428069254250939292704696175748719031859166 075
    45118270733171351396940329 9",
   "1"
  ],
  "protocol": "groth16",
  "curve": "bn128"
 },
 "publicSignals": {
  "r": "18"
 }
}
```


Also, at the end of the proof, we have the public signals.

Remark. Usually, public signals are represented by an array of elements, but when using the `hardhat-zkit` plugin, they are typed, and we have actual names for them.

The third element of each program does not participate in any computations; it is needed as additional metadata for the library that implements Groth16 verification.

Remark. When submitting the proof, we have to swap elements inside the arrays of the b point, so that the proof can be verified correctly.

These three points π_L, π_R, π_O are used by the verifier to check the equality:

$$e(\pi_L, \pi_R) = e(g_1^\alpha, g_2^\beta) e(\pi_{\text{io}}, g_2^\gamma) e(\pi_O, g_2^\delta).$$

Other constants needed for the verifier (for example, points g_1^α or g_2^β) are defined in the following file: `zkit/artifacts/circuits/Math.circom/Math.vkey.json`. We will not show this file due to its size, but in fact, it is similar in formatting to the `proof.json` file shown before.

Quick reminder about the structure of points in the proof for BN254 curve:

- Each point is either from the regular curve $\mathbb{G}_1 : y^2 = x^3 + b$ over $\mathbb{F}_p$ or from the quadratic extension curve $\mathbb{G}_2 : y'^2 = x'^3 + b'$ over $\mathbb{F}_{p^2}$. For BN254, the quadratic extension is defined as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1$. Curve coefficients are $b = 3 \in \mathbb{F}_p$ and $b' = \frac{3}{9+i} \in \mathbb{F}_{p^2}$.
- Left inputs to the pairing function e are the points on the regular curve $\mathbb{G}_1$. They are specified in the form of two field elements $(x, y) \in \mathbb{G}_1$, where $x, y \in \mathbb{F}_p$ are the coordinates.
- Right inputs to the pairing function e are the points over the quadratic extension curve $\mathbb{G}_2$. They are specified of the form of four prime field elements $(x_1, y_1, x_2, y_2) \in \mathbb{G}_2$, where the coordinates are $x_1 + iy_1$, $x_2 + iy_2 \in \mathbb{F}_{p^2}$.
- $e(g_1^\alpha, g_2^\beta)$ is the element from the multiplicative group $\mathbb{F}_{p^{12}}^\times$. Therefore, we need 12 prime field elements to represent it.

Remark (On representing $\mathbb{F}_{p^{12}}$ element)**.** One might wonder: why is the element from $\mathbb{F}_{p^{12}}$ is represented as a pair of two arrays, each consisting of three pairs of prime field elements? The primary reason is that the most convenient way to construct $\mathbb{F}_{p^{12}}$ element is to use the so-called **tower of extensions**: we represent an element from $\mathbb{F}_{p^{12}}$ as a pair of two $\mathbb{F}_{p^6}$ elements, while each $\mathbb{F}_{p^6}$ consists of a triplet of $\mathbb{F}_{p^2}$ elements. For more details, see **??**

Thus, we have covered all the information about the internal structure of the Circom files needed for proof generation and verification.

Finally, we verify the proof in the code:

```
expect(await math.verifyProof(proof)).to.be.true
```

Remark. Here is a set of links that can be used for a deeper dive into the Circom ecosystem:

1. Circom Documentation: https://docs.circom.io/
2. Circom Libraries (like circomlib): https://github.com/iden3/circomlib