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** framework.

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 specifying 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 you need to perform operations like division or non-quadratic multiplication on the signal. For this purpose, you can use the '−>' and '<−' notations. 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.

0.1.4 Theoretical Recap: Using the Learned Concepts

Recall the complex example we analyzed in earlier lectures:

Example.
```
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:

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

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

The inputs can be directly transformed into signals like below:

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

    signal input x1;

    signal input x2;
    signal input x3;
```

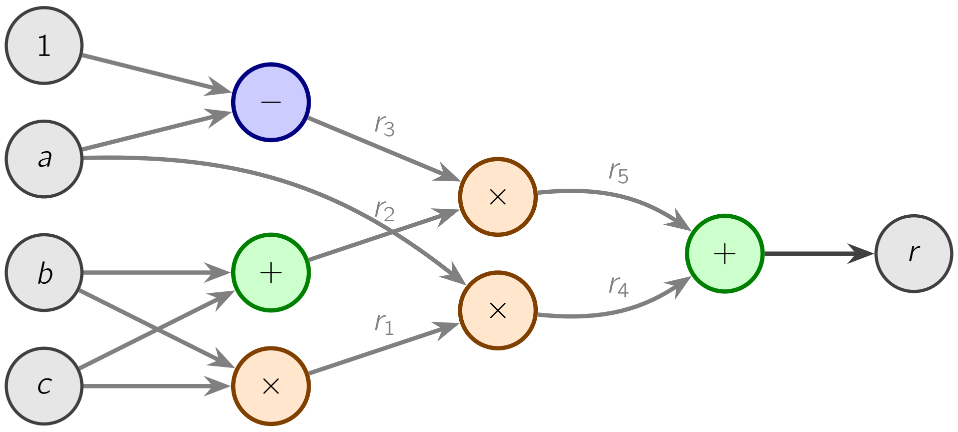

Figure 0.1: Example of a circuit evaluating the `if` statement logic.

```
}
```

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's break down everything that happened. 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.

Let's start with the R1CS file.

In Lecture 8, we defined the following coefficient vectors (in simple constraints) for our task:

$$\begin{aligned}
\mathbf{a}_1 &= (0,0,1,0,0,0,0) & \mathbf{b}_1 &= (0,0,1,0,0,0,0) & \mathbf{c}_1 &= (0,0,1,0,0,0,0)\\
\mathbf{a}_2 &= (0,0,0,1,0,0,0) & \mathbf{b}_2 &= (0,0,0,0,1,0,0) & \mathbf{c}_2 &= (0,0,0,0,0,1,0)\\
\mathbf{a}_3 &= (0,0,1,0,0,0,0) & \mathbf{b}_3 &= (0,0,0,0,0,1,0) & \mathbf{c}_3 &= (0,0,0,0,0,0,1)\\
\mathbf{a}_4 &= (1,0,-1,0,0,0,0) & \mathbf{b}_4 &= (0,0,0,1,1,0,0) & \mathbf{c}_4 &= (0,1,0,0,0,0,-1)
\end{aligned}$$

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

```
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 Lecture 8, 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:

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

Indeed, it aligns with the test from '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 directly corresponds to the theory that $w_0 = 1$ is often used to handle constant terms in R1CS.

0.1.6 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": {
  "pi_a": [
   "4705801711565477046837119510773988173091957417270766918367441244292047980064",
   "1400811599548904237959319989696481634963162026439383059052135976273120564167",
   "1"
  ],
  "pi_b": [
   [
    "12538508168416900299033726521685163817792614632620657244409429354131980454661",
    "10914283679966848917795247355212516197618338956682374874239005506750384424444"
   ],
   [
    "11504632457518572930719312464170675169899321263873993433191427524966381618623",
    "15524163713890313070296837080299781036987071183397727452907670321368057103914"
   ],
   [
    "1",
    "0"
   ]
  ],
  "pi_c": [
   "2609967005332820860840381162476797092857107269468757252635436475859887989998",
   "14278428069254250939292704696175748719031859166075451182707331713513969403299",
   "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 points are used by the verifier:

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

Other constants needed for the verifier are defined in the ‘zkit/artifacts/circuits/Math.circom/Math.vkey
file:

```
{
  "protocol": "groth16",
  "curve": "bn128",
  "nPublic": 1,
  "vk_alpha_1": [
    "20491192805390485299153009773594534940189261866228447918068658471970481763042",
    "9383485363053290200918347156157836566562967994039712273449902621266178545958",
    "1"
  ],
  "vk_beta_2": [
    [
      "6375614351688725206403948262868962793625744043794305715222011528459656738731",
      "4252822878758300859123897981450591353533073413197771768651442665752259397132"
    ],
    [
      "10505242626370262277552901082094356697409835680220590971873171140371331206856",
      "21847035105528745403288232691147584728191162732299865338377159692350059136679"
    ],
    [
      "1",
      "0"
    ]
  ],
  "vk_gamma_2": [
    [
      "10857046999023057135944570762232829481370756359578518086990519993285655852781",
      "11559732032986387107991004021392285783925812861821192530917403151452391805634"
    ],
    [
      "8495653923123431417604973247489272438418190587263600148770280649306958101930",
      "4082367875863433681332203403145435568316851327593401208105741076214120093531"
    ],
    [
      "1",
      "0"
    ]
  ],
  "vk_delta_2": [
    [
      "10857046999023057135944570762232829481370756359578518086990519993285655852781",
      "11559732032986387107991004021392285783925812861821192530917403151452391805634"
    ],
    [
      "8495653923123431417604973247489272438418190587263600148770280649306958101930",
      "4082367875863433681332203403145435568316851327593401208105741076214120093531"
    ],
    [
      "1",
      "0"
    ]
  ],
  "vk_alphabeta_12": [
    [
      [
        "2029413683389138792403550203267699914886160938906632433982220835551125967885",
        "21072700047562757817161031222997517981543347628379360635925549008442030252106"
      ],
      [
        "5940354580057074848093997050200682056184807770593307860589430076672439820312",
        "12156638873931618554171829126792193045421052652279363021382169897324752428276"
      ],
      [
        "7898200236362823042373859371574133993780991612861777490112507062703164551277",
        "7074218545237549455313236346927434013100842096812539264420499035217050630853"
      ]
    ],
    [
      [
        "7077479683546002997211712695946002074877511277312570035766170199895071832130",
        "10093483419865920389913245021038182291233451549023025229112148274109565435465"
      ],
      [
        "4595479056700221319381530156280926371456704509942304414423590385166031118820",
        "19831328484489333784475432780421641293929726139240675179672856274388269393268"
      ],
      [
        "11934129596455521040620786944827826205713621633706285934057045369193958244500",
        "8037395052364110730298837004334506829870972346962140206007064471173334027475"
      ]
    ]
  ],
  "IC": [
    [
      "4162541565828872643496914921393902054824387648641933177665940781539334781623",
      "4678293780284819015763290392952715769540194300841323348855962545628746384938",
      "1"
    ],
    [
      "7846408072049176620553358542204120795817938985459251067840222635524693287955",
      "17494754572705064819681843056808269434754047134538681907566256240907807975850",
      "1"
    ]
  ]
}
```

Quick reminder about the structure of points in the proof:

- Left inputs to e are of the form $(x, y) \in \mathbb{G}_1$ — a regular curve.
- Right inputs to e are of the form $((x_1, y_1), (x_2, y_2)) \in \mathbb{G}_2$ — a "complex" curve, consisting of two $\mathbb{F}_{p^2}$ coordinates.
- $e(g_1^\alpha, g_2^\beta)$ is of the form $(x_1, \ldots, x_{12}) \in \mathbb{F}_{p^{12}}$.

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
```

This concludes our first journey into learning Circom.

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`