Circom Language Tutorial Notes

Dr. Cyprian Omukhwaya Sakwa

1 Introduction

`Circom` is a domain-specific language for creating arithmetic circuits used in zero-knowledge proofs, particularly for generating **Rank-1 Constraint Systems (R1CS)**. Although it is considered low-level, its explicit constraint structure makes it ideal for learning the foundations of zk-SNARKs and reasoning about circuit behavior.

> Note
>
> **Version Note:** This guide assumes usage of `Circom 2.1+`, which includes improvements in syntax, component usage, and modular templates.

This guide synthesizes official and community resources on `Circom` and zk-SNARK tooling, curated and clarified by Dr. Cyprian Omukhwaya Sakwa. It walks through core concepts, syntax, tooling, and practical examples.

Disclaimer

This document is under active development. It may contain incomplete sections, updates in progress, or content subject to revision. Please check back regularly for the latest version.

1.1 When to Use Circom

`Circom` is particularly suitable in the following scenarios, where low-level circuit design offers unique advantages:

- **Learning how ZK circuits work under the hood:**
  Because `Circom` exposes constraint-level operations (such as individual multiplications and signal assignments), it serves as an excellent tool for understanding how arithmetic circuits are compiled into R1CS. It helps demystify how high-level operations decompose into low-level constraints that underpin SNARKs.
- **Prototyping small ZK modules:**

`Circom` is ideal for writing small, self-contained gadgets (like comparators, hash preimages, or simple commitments). These gadgets can then be composed into larger systems or used for testing cryptographic primitives in isolation.

- **Gaining precise control over constraint generation:**
  Unlike higher-level languages that abstract away circuit logic, `Circom` allows the developer to explicitly manage each signal and constraint. This level of control is useful for optimizing performance or verifying the exact structure of the underlying R1CS, especially for custom or non-standard primitives.

Note

`Circom` is a low-level circuit language that offers developers fine-grained control over constraint generation. This means that, much like programming in C gives you more control than a high-level language like Python, Circom allows you to work directly with the wires (signals) and constraints of your zero-knowledge circuit.

For example, to enforce the multiplication constraint $z = x \cdot y$, you would write in Circom:

```
signal input x;
signal input y;
signal output z;

z <== x * y;
```

This explicitly instructs the constraint system to create a multiplication gate. In contrast, a high-level ZK language like Noir might let you write:

```
fn main(x: Field, y: Field) -> Field {
 return x * y;
}
```

and automatically handle the underlying constraint generation. This fine-grained control in Circom is powerful for optimization and learning, but requires careful attention to correctness.

Warning

Note on Production Use:
While `Circom` is powerful and flexible, it is not always the best tool for large-scale production systems:

- **Always get a professional audit:**
  ZK circuits can contain subtle bugs that are hard to detect. Circuits written in `Circom`, due to their low-level nature, are especially susceptible to human error. A cryptographic audit is essential for any code handling real assets or sensitive logic.
- **Consider ergonomic, high-level ZK languages for complex projects:**
  Languages like `Noir`, `Leo`, or DSLs based on `Halo2` provide improved syntax, modularity, and safety features. These may be better suited for large applications, team collaboration, and long-term maintenance.

In short, `Circom` is a fantastic tool for understanding, teaching, and prototyping—but caution and tooling maturity must be considered before deploying it at scale.

2 Installing Dependencies

To run `Circom` and its associated tooling, you will need several dependencies installed on your system. These include the `Circom` compiler itself (written in Rust), and Node.js tooling that supports witness generation and further ZKP integrations.

2.1 Core Requirements

- **Rust Toolchain:** `Circom` is implemented in Rust, so the first step is to install the Rust toolchain via `rustup`.
- **Node.js and Package Manager:** Several utilities are published as `npm` packages. Make sure Node.js (version 10 or higher) is available, along with either `npm` or `yarn`.

2.2 Installing Rust

The `circom` compiler is written in Rust. To install Rust, you can use the recommended toolchain manager `rustup`. If you're on Linux or macOS, open a terminal and run:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Follow the prompts to complete the installation. This will install the `rustc`, `cargo` package manager, and other Rust tooling into your system.

Note

Tip: After installation, restart your terminal or run `source $HOME/.cargo/env` to ensure Rust binaries are in your PATH.

2.3 Installing Node.js and npm

Circom tooling includes a number of npm packages, so you should also install:

- `node.js` (version 10 or higher is recommended)
- A package manager such as `npm` or `yarn`

Recent versions of Node.js support WebAssembly and big integer arithmetic, both of which improve the performance of Circom-related tools.

2.4 Cloning and Compiling Circom

To install `Circom` from source:

1. Clone the repository:

   ```
   git clone https://github.com/iden3/circom.git
   ```

2. Enter the directory:

   ```
   cd circom
   ```

3. Compile the project in release mode:

   ```
   cargo build --release
   ```

   This step may take a few minutes. Once complete, it will produce a binary at `target/release/circom`.

4. Optionally, install the binary globally so you can invoke `circom` from anywhere:

   ```
   cargo install --path circom
   ```

This command installs the binary into `$HOME/.cargo/bin`. To make sure this directory is in your system's PATH (so that you can invoke `circom` globally), add the following line to your shell configuration file (e.g., `.bashrc`, `.zshrc`):

```
export PATH="$HOME/.cargo/bin:$PATH"
```

Then, reload your shell configuration:

```
source ~/.bashrc  # or source ~/.zshrc
```

Now you should be able to verify the installation by running:

```
circom --help
```

2.5 Circom Help Output

Running the help command will display all available options for the Circom compiler:

```
circom compiler 2.2.2
IDEN3
Compiler for the circom programming language

USAGE:
circom [FLAGS] [OPTIONS] [--] [input]

FLAGS:
--r1cs                                  Outputs the constraints in r1cs format
--sym                                   Outputs witness in sym format
--wasm                                  Compiles the circuit to wasm
--json                                  Outputs the constraints in json format
--wat                                   Compiles the circuit to wat
-c, --c                                     Compiles the circuit to C++
--O0                                    No simplification is applied
--O1                                    Signal and constant simplification (default)
--O2                                    Full constraint simplification
--verbose                               Shows logs during compilation
--inspect                               Additional validation over constraints
--constraint_assert_dissabled           Disable assert checks for "===" constraints
--use_old_simplification_heuristics     Use old heuristic simplifications
--simplification_substitution           Outputs simplification substitutions in JSON
--no_asm                                Avoids ASM files in witness generation
--no_init                               Skips zero-initializations in witness code
-h, --help                                  Prints help information
-V, --version                               Prints version information

OPTIONS:
-o, --output <output>                     Output directory [default: .]
-p, --prime <prime>                      Prime field: bn128, bls12377, bls12381, goldilocks,
                                         grumpkin, pallas,
secq256r1, vesta [default: bn128]
-l <link_libraries>...                    Additional library paths
--O2round <simplification_rounds>    Number of simplification passes

ARGS:
<input>    Path to a circuit with a main component [default: ./circuit.circom]
```

2.6 Install snarkjs

`snarkjs` is a command-line utility and JavaScript library that facilitates working with zero-knowledge proofs. It consumes the constraint and witness artifacts generated by `Circom`, and allows you to perform tasks such as trusted setup ceremonies, proof generation, and verification.

Install via `npm`

To install `snarkjs` globally using `npm`, run the following command:

```
npm install -g snarkjs@latest
```

This will make the `snarkjs` CLI available globally from your terminal.

> Note
>
> **Note:** Make sure `node` and `npm` are properly installed before running this command. You can verify the installation with:
>
> ```
> node -v
> npm -v
> ```

After installation, you can check that everything is working by running:

```
snarkjs --help
```

This will display a list of available subcommands such as `groth16 setup`, `plonk prove`, `verify`, and `zkey contribute`, which you'll use in later stages of the zk-SNARK workflow.

You can now start writing and compiling Circom circuits from anywhere on your system.

3 Hello World: Your First ZK Circuit

The "Hello World" of zero-knowledge circuits is a simple multiplication. We start by writing a basic Circom circuit that multiplies two inputs:

Listing 1: multiply.circom

```
pragma circom 2.1.6;

template Multiply() {
        signal input a;
        signal input b;
        signal output out;

        out <== a * b;
```

```
}

component main = Multiply();
```

Explanation:

- `signal input a, b`: Declares the inputs to the circuit.
- `signal output out`: Declares the output.
- `out <== a * b`: Enforces the constraint that `out` is the product of `a` and `b`.
- `component main = Multiply()`: Sets this component as the entry point (main circuit).

The arithmetic circuits built using `circom` operate on *signals*. First, the `pragma` instruction is used to specify the compiler version. This ensures that the circuit is compatible with the version indicated after the `pragma` statement; otherwise, the compiler will issue a warning. Next, we use the reserved keyword `template` to define the structure of a new circuit, which we call `Multiply`. Inside this template, we declare its `signals`, which represent the inputs and outputs of the circuit.

The shorthand operator `==>` can be used instead of `<==`:

$$a \cdot b \Rightarrow \texttt{out}$$

In every template, we first declare the *signals*, followed by the specification of the *constraints*.

3.1 Save & Compile the Circuit

Save the above code as

```
multiply.circom
```

To compile the circuit and ensure it is syntactically valid, run:

```
circom multiply.circom
```

Expected output:

```
template instances: 1
Everything went okay
```

3.2 Generate R1CS and Symbol Files

Use the following command to generate the constraint system and symbolic names from the circuit:

```
circom multiply.circom --r1cs --sym
```

This produces two files:

- `multiply.r1cs`: The constraint system in binary form.
- `multiply.sym`: Symbolic names for debugging and interpretation.

3.3 Inspecting the R1CS

To examine the actual constraints encoded in the R1CS, use `snarkjs`:

```
snarkjs r1cs print multiply.r1cs
```

Typical output:

```
[ 2188824287...main.a ] * [ main.b ] - [ 2188824287...main.out ] = 0
```

Note: The large number shown is the prime modulus p of the BN254 scalar field:

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

This number is used to perform all arithmetic modulo p. Here, the value:

$$21888242871839275222246405745257275088548364400416034343698204186575808495616 \equiv -1 \mod p$$

So the R1CS constraint:

$$-1 \cdot a \cdot b - (-1 \cdot \text{out}) = 0$$

is algebraically equivalent to:

$$a \cdot b = \text{out}$$

This example demonstrates the basic structure of a Circom circuit, the process to compile it, and how to interpret the generated R1CS constraints. It serves as a foundation for building more complex zero-knowledge proofs.

4 Non-Quadratic Constraints in Circom

4.1 Quadratic Constraint Limitation in R1CS

Circom circuits must compile down to a valid **Rank-1 Constraint System (R1CS)**, where each constraint (i.e., each row of the system) is allowed to have only **one multiplication**. This is known as the **quadratic constraint** requirement.

Any constraint with more than one multiplication is invalid and will not compile.

Consider the following incorrect example:

Listing 2: Invalid: More than one multiplication in one constraint

```
pragma circom 2.1.6;

template Multiply() {
        signal input a;
        signal input b;
        signal input c;
        signal output out;

        out <== a * b * c; //      Invalid
}

component main = Multiply();
```

Running:

```
circom multiply.circom
```

produces the following error:

```
error[T3001]: Non quadratic constraints are not allowed!
 "multiply.circom":9:3

9    out <== a * b * c;
   ^^^^^^^^^^^^^^^^^ found here

= call trace:
->Multiply

previous errors were found
```

Explanation: The `<==` operator in Circom defines a constraint, and this one tries to perform two multiplications ($a \cdot b \cdot c$) in a single constraint, which is not allowed.

Fix: Splitting the Constraint

To resolve this, we break the computation into multiple constraints, each with only one multiplication.

We introduce an intermediate signal `s1` to hold the result of the first multiplication:

Listing 3: Valid: One multiplication per constraint

```
pragma circom 2.1.6;

template Multiply() {
        signal input a;
```

```
        signal input b;
        signal input c;
        signal s1;
        signal output out;

        s1 <== a * b;      // First constraint
        out <== s1 * c;    // Second constraint
}

component main = Multiply();
```

Inspecting the Fixed R1CS

After regenerating the R1CS using:

```
circom multiply.circom --r1cs --sym
snarkjs r1cs print multiply.r1cs
```

We may observe output like:

```
[INFO]  snarkJS: [ 2188824287...main.a ] * [ main.b ] - [ 2188824287...main.s1 ] = 0

[INFO]  snarkJS: [ 2188824287...main.s1 ] * [ main.c ] - [ 2188824287...main.out ] = 0
```

This translates algebraically to:

$$a \cdot b = s_1$$
$$s1 \cdot c = \text{out}$$

This structure enforces that each R1CS row (i.e., each constraint) performs exactly one multiplication, satisfying the requirements of the system and making the circuit valid.

5 Computing the Witness

After writing and compiling your circuit, the next step is to compute the **witness**. The witness contains all intermediate and output values computed by the circuit given a specific input.

Generating the Witness Code

To generate the necessary files for witness generation, run the following terminal command:

```
circom multiply.circom --r1cs --sym --wasm
```

This command:

- Regenerates the `.r1cs` (Rank-1 Constraint System) file.
- Regenerates the `.sym` (symbol mapping) file.
- Creates a directory called `multiply_js/` containing a compiled WebAssembly file and JavaScript code to generate the witness.

Preparing Input

Navigate to the generated folder:

```
cd multiply_js/
```

Inside this directory, create an `input.json` file. This file maps the **input signals** (those declared with `signal input`) to their actual values. Here's an example:

```
{
 "a": "3",
 "b": "5",
 "c": "7"
}
```

Note:

- Only the input signals `a`, `b`, and `c` need to be specified.
- Internal signals like `s1` and `out` are computed automatically.
- The presence of the `.sym` file allows Circom to match names in `input.json` to the correct signal IDs.

Generating the Witness

Once the input file is ready, generate the witness with:

```
node generate_witness.js multiply.wasm input.json
witness.wtns
```

To inspect the contents of the witness in human-readable form, export it to JSON:

```
snarkjs wtns export json witness.wtns
```

Then print the file:

```
cat witness.json
```

You should see output similar to:

```
[
"1",
"105",
"3",
```

```
"5",
"7",
"15"
]
```

Understanding the Output

The witness is an array of field elements (in string form). Each entry corresponds to a signal in the circuit, following the R1CS variable order:

`[1, out, a, b, c, s1]`

In our example:

- `a = 3`, `b = 5`, `c = 7`
- `s1 = a * b = 15`
- `out = s1 * c = 15 * 7 = 105`

Thus, the witness is:

- `"1"` a constant for the R1CS system.
- `"105"` the final output.
- `"3", "5", "7"` the inputs.
- `"15"` the intermediate value of `s1`.

This step proves that the circuit, given the inputs, computes the output and all intermediate values correctly.

6 Public Inputs in Circom

Motivation: Nullifier Schemes

In certain cryptographic protocols, especially those involving zero-knowledge proofs, we may want to make some inputs public. One motivation comes from **nullifier schemes**, which are commonly used in privacy-preserving systems such as mixers or shielded transactions.

What is a Nullifier Scheme?

A nullifier scheme typically works as follows:

- Concatenate two secret numbers.
- Hash the result.
- Later, reveal only one of the original numbers (not both).

The idea is to prove knowledge of a preimage (a secret input to the hash) *without revealing which one* it corresponds to. This prevents double-use of the secret while maintaining privacy.

If the hash were based on just one number, revealing it would disclose the exact preimageand hence which hash it was associated with. On the other hand, revealing nothing at all allows the prover to re-use the same proof multiple times, which is dangerous in scenarios like smart contract withdrawals.

By revealing just one of the two original inputs (say, a nullifier), we achieve a balance:

- The action can't be repeated (because the nullifier is public).
- We dont reveal enough to identify the full preimage.

Public Inputs in Circom

Circom provides a clean way to declare which signals are public. Consider the following circuit:

```
template SomePublic() {

 signal input a;
 signal input b;
 signal input c;
 signal v;
 signal output out;

 v <== a * b;
 out <== c * v;
}

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

Explanation

- Signals `a`, `b`, and `c` are declared as `input`, meaning they are inputs to the circuit.
- `v` is an internal signal used for intermediate computation.
- `out` is the final output.

The key point is:

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

This line declares that only `a` and `c` are **public inputs**, while `b` remains **private**. When this circuit is used to generate a proof, the verifier will have access to the values of `a` and `c`, but not `b`.

Why is this useful?

In the context of nullifiers:

- We can set one of the inputs (e.g., the nullifier) as public.
- The rest (e.g., the secret preimage) stays hidden.
- The circuit verifies a computation involving both, without revealing the full picture.

This pattern supports privacy and prevents replay attacks or double-spending in cryptographic applications.

Inspecting the Fixed R1CS

After regenerating the R1CS using:

```
circom multiply.circom --r1cs --sym
snarkjs r1cs print SomePublic.r1cs
```

We may observe output like:

```
[INFO]  snarkJS: [ 2188824287...main.a ] * [ main.b ] - [ 2188824287...main.v ] = 0

[INFO]  snarkJS: [ 2188824287...main.c ] * [ main.v ] - [ 2188824287...main.out ] = 0
```

This translates algebraically to:

$$a \cdot b = v$$
$$c \cdot v = \text{out}$$

This structure enforces that each R1CS row (i.e., each constraint) performs exactly one multiplication, satisfying the requirements of the system and making the circuit valid.

7 Computing the Witness

After writing and compiling your circuit, the next step is to compute the **witness**. The witness contains all intermediate and output values computed by the circuit given a specific input.

Generating the Witness Code

To generate the necessary files for witness generation, run the following terminal command:

```
circom multiply.circom --r1cs --sym --wasm
```

This command:

- Regenerates the `.r1cs` (Rank-1 Constraint System) file.
- Regenerates the `.sym` (symbol mapping) file.
- Creates a directory called `multiply_js/` containing a compiled WebAssembly file and JavaScript code to generate the witness.

Preparing Input

Navigate to the generated folder:

```
cd multiply_js/
```

Inside this directory, create an `input.json` file. This file maps the **input signals** (those declared with `signal input`) to their actual values. Here's an example:

```
{
 "a": "3",
 "b": "9",
 "c": "7"
}
```

Note:

- Only the input signals `a`, `b`, and `c` need to be specified.
- Internal signals like `s1` and `out` are computed automatically.
- The presence of the `.sym` file allows Circom to match names in `input.json` to the correct signal IDs.

Generating the Witness

Once the input file is ready, generate the witness with:

```
node generate_witness.js SomePublic.wasm input.json witness.wtns
```

To inspect the contents of the witness in human-readable form, export it to JSON:

```
snarkjs wtns export json witness.wtns
```

Then print the file:

```
cat witness.json
```

You should see output similar to:

```
[
"1",
"189",
"3",
"9",
```

```
"7",
"27"
]
```

8 Proving Circuits Overview

After compiling a circuit and running the witness calculator with appropriate inputs, two important files are generated:

- `.wtns`: contains all the computed signals (witness).
- `.r1cs`: contains the Rank-1 Constraint System representation of the circuit.

These files are essential for generating and verifying a zk-SNARK proof.

As an example, consider a circuit called `multiply` that proves knowledge of two factors a and b such that:

$$z = a \cdot b$$

We will use the **Groth16** zk-SNARK protocol to construct and verify this proof.

Trusted Setup for Groth16

Groth16 requires a trusted setup with two phases:

- **Powers of Tau (Phase 1)** circuit-independent.
- **Phase 2** circuit-specific.

Powers of Tau (Phase 1)

Start the Ceremony

```
snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
```

Contribute to the Ceremony

```
snarkjs powersoftau contribute pot12_0000.ptau
 pot12_0001.ptau --name="Type your Name" -v
```

This generates `pot12_0001.ptau`, which will be used in Phase 2.

Phase 2 (Circuit-Specific)

Prepare Phase 2 File

```
snarkjs powersoftau prepare phase2 pot12_0001.ptau
 pot12_final.ptau -v
```

Generate zkey File

```
snarkjs groth16 setup multiply.r1cs pot12_final.ptau
 multiply_0000.zkey
```

Contribute to Phase 2

```
snarkjs zkey contribute multiply_0000.zkey
 multiply_0001.zkey --name="1st Contributor Name" -v
```

Export Verification Key

```
snarkjs zkey export verificationkey multiply_0001.zkey
verification_key.json
```

Create `input.json`

In your project folder (`circom`), create a file named `input.json` with the following content:

Listing 4: input.json

```
{"a": "17","b": "21"}
```

You may use any numbers you'd like to multiply.

Run the Witness Generator

Run the following command in your terminal (from the `circom` folder):

```
node multiply_js/multiply_js/generate_witness.js
 multiply_js/multiply_js/multiply.wasm multiply_js/input.json
  witness.wtns
```

Troubleshooting

If you encounter an error like `generate_witness.js doesn't exist`, recompile your `.circom` file using this command:

```
circom multiply.circom --r1cs --wasm --sym -o multiply_js
```

Generating a Proof

After computing the witness and completing the trusted setup:

Generate Proof

```
snarkjs groth16 prove multiply_0001.zkey witness.wtns
proof.json public.json
```

- `proof.json` contains the zk-SNARK proof.
- `public.json` contains the public inputs/outputs.

Verifying a Proof

Verify Proof

```
snarkjs groth16 verify verification_key.json public.json
 proof.json
```

If valid, the tool outputs `OK`. This confirms:

- The proof is valid for the circuit.
- The public inputs/outputs match those in `public.json`.

The following files are produced during the Groth16 zero-knowledge proof generation process:

File	Description
`verification_key.json`	Exported from the final `.zkey` file using: `snarkjs zkey export verificationkey multiply_0001.zkey verification_key.json` Used to verify the proof either off-chain or on-chain.
`proof.json`	Contains the actual zero-knowledge proof. Generated with: `snarkjs groth16 prove multiply_0001.zkey witness.wtns proof.json public.json`
`public.json`	Contains public inputs to the circuit (e.g., $a \cdot b$) used in verification.
`witness.wtns`	Intermediate binary file containing all intermediate values from the circuit evaluation. Required only during proof generation.

Next Steps After Generation

- Use `verification_key.json`, `proof.json`, and `public.json` to verify the proof locally:

  ```
  snarkjs groth16 verify verification_key.json public.json proof.json
  ```

- The verification key can also be used to generate a Solidity verifier contract:

  ```
  snarkjs zkey export solidityverifier multiply_0001.zkey verifier.sol
  ```

  This contract allows the proof to be verified on-chain in Ethereum-compatible environments.

- `witness.wtns` is not needed after proving is complete; it can be deleted unless debugging or re-running proofs.

On-Chain Verification (Solidity Verifier)

To verify the proof on Ethereum:

Generate Solidity Verifier Contract

Export Solidity Verifier

```
snarkjs zkey export solidityverifier multiplier2_0001.zkey
verifier.sol
```

This creates a file `verifier.sol` containing two contracts:

- `Pairing`
- `Verifier` only this needs to be deployed.

Use Remix for Deployment

You can paste the Solidity code into Remix and deploy the `Verifier` contract. Use a testnet like `Sepolia` or `Goerli`, or the JavaScript VM in Remix.

Calling `verifyProof`

To generate calldata for the `verifyProof` function:

Generate Calldata

```
snarkjs generatecall
```

Paste the generated parameters into the Remix interface. The function returns `TRUE` only if the proof is valid. Changing even a single bit in the parameters causes verification to fail.

9 Arrays in Circom

In this section, we demonstrate how to compute the first n powers of an input using Circom. Instead of manually defining each intermediate output, Circom allows the use of **signal arrays** and **template parameters** to generalize behavior.

Example Code

Listing 5: Computing powers of input using signal arrays

```
pragma circom 2.1.6;

template Powers(n) {
        signal input a;
        signal output powers[n];
```

```
        powers[0] <== a;
        for (var i = 1; i < n; i++) {
            powers[i] <== powers[i - 1] * a;
        }
}

component main = Powers(6);
```

Explanation and Concepts

- **Signal Arrays:**
  - The line signal output powers[n]; declares an array of output signals of size n.
  - This is more scalable than declaring each output manually.
- **Initialization and Loop:**
  - We initialize the first element as powers[0] <== a;.
  - A for loop iteratively computes the next powers using previous values: powers[i] <== powers[i − 1] ∗ a;.
- **Template Parameters:**
  - template Powers(n) makes the circuit reusable with different sizes.
  - The actual size (e.g., Powers(6)) must be hardcoded when instantiating the component.
  - This is because Circom generates a *static* constraint system, which cannot be resized dynamically.
- **R1CS Immutability:**
  - Rank 1 Constraint Systems (R1CS) must have a fixed shape.
  - Thus, the parameter n must be known at compile time and cannot be changed during execution.

Conclusion

Arrays in Circom provide a clean and scalable way to model repeated structures like powers, sums, and accumulations. Template parameters make components reusable and flexible while still satisfying the immutability constraints of zero-knowledge proof systems like R1CS.

Circom Variables

To better understand the role of variables in Circom, consider the equivalent version of the `Powers` component written without a loop:

Listing 6: Unrolled version without variables

```
pragma circom 2.1.6;

template Powers() {
        signal input a;
        signal output powers[6];

        powers[0] <== a;
        powers[1] <== powers[0] * a;
        powers[2] <== powers[1] * a;
        powers[3] <== powers[2] * a;
        powers[4] <== powers[3] * a;
        powers[5] <== powers[4] * a;
}

component main = Powers();
```

What This Teaches Us

- Although the circuit is written differently, the resulting **Rank-1 Constraint System (R1CS)** is identical to the version that uses a `for` loop with a `var` variable.
- This emphasizes an important point:

  Key Insight

  Variables in Circom (e.g., `var i`) exist only at circuit construction time. They do not translate into constraints.

- Variables are purely a tool to help describe how the circuit should be constructed. They help automate and abstract the generation of constraints, but do not become part of the circuit's witness or its constraint system.

Why Use Variables?

- Without variables and loops, circuits quickly become verbose and harder to maintain.
- The looped version is cleaner, more scalable, and easier to parameterize with a template argument like `n`.

- Variables improve circuit readability without compromising performance or constraint integrity.

Conclusion

Circom variables help in **generating constraints**, but they are not themselves part of the circuit. Think of them as *compile-time helpers*, not runtime data.

Signals vs Variables

Key Differences

- **Signals** are *immutable* and correspond directly to entries in the **witness vector** of the R1CS. Each signal can be seen as a column in the constraint system.
- **Variables** are only used during circuit constructionthey do **not** appear in the R1CS and exist purely at compile time for convenience.

Why Signals Are Immutable

Signals represent witness entries. Once a value is chosen for the witness, it must remain fixed to maintain a valid proof. Mutating signal values would invalidate the R1CS.

Incorrect Examples (Not Allowed)

Listing 7: Invalid Signal and Variable Assignments

```
signal a;
a = 2;          //      Invalid: Cannot assign to a signal using '

var v;
v <— a + b;     //      Invalid: Cannot assign to a variable using
```

Operator Summary

- `<--` and `<==` perform assignments for signals.
- `===` adds a constraint without performing assignment.
- For variables, normal C-style operators apply: `=`, `==`, `!=`, `++`, `--`, etc.

`<==` vs `===`

Listing 8: Semantically Equivalent Circuits

```
pragma circom 2.1.6;
```

```
template Multiply() {
        signal input a;
        signal input b;
        signal output c;

        c <-- a * b;
        c === a * b;
}

template MultiplySame() {
        signal input a;
        signal input b;
        signal output c;

        c <== a * b;
}
```

- `<==` both assigns and enforces a constraint.
- `===` only enforces a constraint, assuming a value has already been assigned using `<--`.

Constraint-Only Example

Sometimes, the prover is responsible for supplying both input and output values. In such cases, you only need a constraint:

Listing 9: Constraint Without Output Signal

```
pragma circom 2.1.6;

template Multiply() {
        signal input a;
        signal input b;
        signal input c;

        c === a * b;
}

component main {public [c]} = Multiply();
```

Note: Circom does not require an `output` signal; the notion of output is syntactic sugar. Internally, everything is part of the witness vectorthere is no technical distinction between input and output in the constraint system.

Acknowledgments

I would like to acknowledge the invaluable resources and communities that contributed to the development of these notes on `circom`:

- The official `circom` documentation for providing comprehensive and authoritative references.
- The `snarkjs` repository for tools and examples that complement `circom` circuits.
- Tutorials and learning materials from `iden3`, the creators of `circom`, for guiding best practices in zero-knowledge circuit design.
- The broader zero-knowledge proof community, whose discussions and open-source contributions continue to inspire and refine practical implementations of zk-SNARKs.

Special thanks to my students and colleagues at `Web3Clubs Foundation Limited` for their curiosity, insights, and collaboration in exploring zk-SNARKs with `circom`.