# Synthesizer: Output Files Reference

This document explains how to read and interpret the three output files generated by [Synthesizer](synthesizer-terminology.md#synthesizer): permutation.json, instance.json, and placementVariables.json.

## Overview

After processing a transaction, Synthesizer generates three JSON files that contain all the information needed for proof generation:

| File | Purpose |
| --- | --- |
| permutation.json | Circuit topology (wire connections) |
| instance.json | Public/private I/O witness |
| placementVariables.json | Complete witness for all subcircuits |

## 1. permutation.json

### Purpose

Describes **how** [**wires**](synthesizer-terminology.md#wire) **are connected** between [placements](synthesizer-terminology.md#placement). This defines the circuit’s topology as a Directed Acyclic Graph (DAG).

### Structure

[  
 {  
 "row": 100,  
 "col": 2,  
 "X": 619,  
 "Y": 26  
 },  
 {  
 "row": 619,  
 "col": 26,  
 "X": 619,  
 "Y": 27  
 },  
 {  
 "row": 619,  
 "col": 27,  
 "X": 100,  
 "Y": 2  
 },  
 // ... more entries  
]

### Reading the Format

**Key Pattern**: Entries come in **N-entry cycles** (where N = number of wires sharing the same value):

Example: 3-entry cycle (when 3 wires share the same value)  
Entry 1: (row, col) → (X, Y) // Wire 1 → Wire 2  
Entry 2: (X, Y) → (X', Y') // Wire 2 → Wire 3  
Entry 3: (X', Y') → (row, col) // Wire 3 → Wire 1 (cycle back)

**Important**: The cycle size depends on how many placements use the same wire value:

* 2 wires sharing a value → 2-entry cycle
* 3 wires sharing a value → 3-entry cycle
* N wires sharing a value → N-entry cycle

**Coordinate System**:

* (row, col) = Wire position in format (wireIndex, placementId)
* row = Wire index within a [placement](synthesizer-terminology.md#placement)
* col = Placement ID ([subcircuit](synthesizer-terminology.md#subcircuit) instance number)

**Example: 3-Entry Cycle**

// Wire 100 is shared by 3 placements: (100,2), (619,26), (619,27)  
{ "row": 100, "col": 2, "X": 619, "Y": 26 }, // Wire 1 → Wire 2  
{ "row": 619, "col": 26, "X": 619, "Y": 27 }, // Wire 2 → Wire 3  
{ "row": 619, "col": 27, "X": 100, "Y": 2 } // Wire 3 → Wire 1

Interpretation:

1. **Placement 2** ([PRV\_IN](synthesizer-terminology.md#prv-in-and-prv-out) buffer), [**wire**](synthesizer-terminology.md#wire) **100** outputs a value
2. This value is used by **Placement 26** (wire 619) and **Placement 27** (wire 619)
3. The 3-entry cycle ensures: Placement2[100] == Placement26[619] == Placement27[619]

### How Placements are Grouped (Code-Based)

The grouping of placements into N-entry cycles happens in two main steps during finalization:

**Step 1: Build Permutation Groups** (\_buildPermGroup())

// permutation.ts:441-562  
private \_buildPermGroup(): Map<string, boolean>[] {  
 let permGroup: Map<string, boolean>[] = [];  
  
 // 1. Create groups from output wires (representatives)  
 for (const placeId of this.placements.keys()) {  
 const thisPlacement = this.placements.get(placeId)!;  
 for (let i = 0; i < thisSubcircuitInfo.NOutWires; i++) {  
 const placementWireId = {  
 placementId: placeId, // e.g., 2 (PRV\_IN)  
 globalWireId: globalWireId // e.g., 100  
 };  
 const groupEntry = new Map();  
 groupEntry.set(JSON.stringify(placementWireId), true);  
 permGroup.push(groupEntry); // New group created  
 }  
 }  
  
 // 2. Add input wires to their parent's group  
 for (const thisPlacementId of this.placements.keys()) {  
 const thisPlacement = this.placements.get(thisPlacementId)!;  
 for (let i = 0; i < thisSubcircuitInfo.NInWires; i++) {  
 const thisInPt = thisPlacement.inPts[i];  
  
 if (thisInPt.source !== thisPlacementId) {  
 // Find parent placement  
 const pointedPlacementId = thisInPt.source!; // e.g., 2  
 const pointedOutputId = /\* find matching output wire \*/;  
  
 // Add this wire to parent's group  
 searchInsert(pointedPlacementWireId, thisPlacementWireId, permGroup);  
 }  
 }  
 }  
  
 return permGroup; // Groups of wires with same value  
}

**Result**: Groups like [{placementId:2, wireId:100}, {placementId:26, wireId:619}, {placementId:27, wireId:619}]

**Step 2: Generate N-Entry Cycles** (\_correctPermutation())

// permutation.ts:368-418  
private \_correctPermutation() {  
 let permutationFile = [];  
  
 for (const \_group of this.permGroup) {  
 const group = [...\_group.keys()]; // Array of wire IDs  
 const groupLength = group.length; // N = number of wires sharing same value  
  
 if (groupLength > 1) {  
 // Create N-entry cycles: Wire1 → Wire2 → ... → WireN → Wire1  
 for (let i = 0; i < groupLength; i++) {  
 const element = JSON.parse(group[i]);  
 const nextElement = JSON.parse(group[(i + 1) % groupLength]); // Cycle!  
  
 permutationFile.push({  
 row: element.globalWireId - setupParams.l,  
 col: element.placementId,  
 X: nextElement.globalWireId - setupParams.l,  
 Y: nextElement.placementId,  
 });  
 }  
 }  
 }  
  
 return permutationFile; // N-entry cycles written to JSON  
}

**Key Points**:

* **Parent-Child Tracking**: During Phase 3 execution, each [DataPt](synthesizer-terminology.md#datapt-data-point) stores its parent via source and wireIndex fields
* **Grouping**: [Wires](synthesizer-terminology.md#wire) with the same value (parent and all children) are grouped together
* **Cycle Size**: groupLength determines the number of entries (2, 3, 4, … N)
* **Cycle Generation**: (i + 1) % groupLength creates the circular structure (last → first)

**Why N-Entry Cycles?**

The cycle structure is required by the Tokamak zk-SNARK proof system to enforce **wire equality constraints**. Each cycle creates a constraint that all N wire positions in the cycle must have the same value, ensuring correct connections between [placements](synthesizer-terminology.md#placement).

## 2. instance.json

### Purpose

Contains **input/output values** for the circuit, divided into public and private data. This is the “[witness](synthesizer-terminology.md#witness)” for the circuit’s I/O boundaries.

### Structure

{  
 "publicOutputBuffer": {  
 "name": "bufferPubOut",  
 "usage": "Buffer to emit public circuit outputs",  
 "subcircuitId": 0,  
 "inPts": [  
 {  
 "source": 30,  
 "wireIndex": 0,  
 "sourceSize": 32,  
 "valueHex": "0xf805dd4619f94a449a4a798155a05a56"  
 },  
 // ... more wires  
 ],  
 "outPts": [...]  
 },  
 "publicInputBuffer": { ... },  
 "privateOutputBuffer": { ... },  
 "privateInputBuffer": { ... },  
 "a\_pub": [...],  
 "a\_prv": [...]  
}

### Reading the Format

[**Buffer**](synthesizer-terminology.md#buffer-placements) **Sections**:

1. **publicInputBuffer** ([Placement](synthesizer-terminology.md#placement) 0 / [PUB\_IN](synthesizer-terminology.md#pub-in-and-pub-out)):
   * **Purpose**: External data that is publicly revealed and brought INTO the circuit
   * **Examples**: calldata, block.number, msg.sender, Keccak hash **outputs** (results computed externally and fed back into circuit)
   * **Used by**: Both Prover and Verifier
2. **publicOutputBuffer** (Placement 1 / [PUB\_OUT](synthesizer-terminology.md#pub-in-and-pub-out)):
   * **Purpose**: Circuit data that is sent OUT to be processed externally
   * **Examples**: return data, event logs, Keccak hash **inputs** (data to be hashed externally)
   * **Used by**: Both Prover and Verifier
   * **Why Keccak inputs are outputs**: Keccak256 is computed outside the circuit for efficiency. The circuit sends the data to hash (PUB\_OUT), external system computes the hash, and the result comes back (PUB\_IN)
3. **privateInputBuffer** (Placement 2 / [PRV\_IN](synthesizer-terminology.md#prv-in-and-prv-out)):
   * **Purpose**: External data that remains hidden
   * **Examples**: storage values, account state, bytecode constants
   * **Used by**: Prover only
4. **privateOutputBuffer** (Placement 3 / [PRV\_OUT](synthesizer-terminology.md#prv-in-and-prv-out)):
   * **Purpose**: Circuit outputs that remain hidden
   * **Examples**: storage updates, internal state changes
   * **Used by**: Prover only

[**Wire**](synthesizer-terminology.md#wire) **Format**:

{  
 "source": 30, // Placement ID that produced this value  
 "wireIndex": 0, // Wire number within that placement  
 "sourceSize": 32, // Size in bytes (usually 32 for 256-bit EVM words)  
 "valueHex": "0x..." // Actual value in hexadecimal  
}

[**Witness**](synthesizer-terminology.md#witness) **Arrays**:

* a\_pub: Complete public witness (all public intermediate values)
* a\_prv: Complete private witness (all private intermediate values)

These arrays flatten all placement variables into sequential format for the prover.

### Example: SLOAD Operation

// Storage value loaded from blockchain  
{  
 "privateInputBuffer": {  
 "inPts": [  
 {  
 "source": 2, // PRV\_IN buffer itself  
 "wireIndex": 104, // 104th wire in PRV\_IN  
 "sourceSize": 32,  
 "valueHex": "0x64" // storage[key] = 100  
 }  
 ]  
 }  
}

Interpretation:

* Storage value 100 (0x64) was loaded via RPC
* Entered circuit as wire 104 in PRV\_IN buffer (Placement 2)
* Remains private throughout proof (never revealed to verifier)

## 3. placementVariables.json

### Purpose

Contains **complete** [**witness**](synthesizer-terminology.md#witness) for every [placement](synthesizer-terminology.md#placement) ([subcircuit](synthesizer-terminology.md#subcircuit) instance). This includes all internal variables needed to satisfy the subcircuit’s [R1CS](synthesizer-terminology.md#r1cs-rank-1-constraint-system) constraints.

### Structure

[  
 {  
 "subcircuitId": 1,  
 "variables": [  
 "0x01", // Variable 0  
 "0x58b5bbeb7719f6739471b5cb1b119a0d", // Variable 1  
 "0xe34ae175aa5b73392e7b87f4fefe45d6", // Variable 2  
 // ... all internal circuit variables  
 ]  
 },  
 {  
 "subcircuitId": 0,  
 "variables": [...]  
 }  
]

### Reading the Format

**Structure**:

* Array of placement records
* Each record contains:
  + subcircuitId: Which [Circom](synthesizer-terminology.md#circom) subcircuit this placement uses
  + variables: All [wire](synthesizer-terminology.md#wire) values for this placement instance

**Variable Ordering**:

The variables array follows Circom’s internal witness ordering:

variables[0] = constant 1 (always 0x01)  
variables[1..N\_out] = output signals  
variables[N\_out+1..N\_out+N\_in] = input signals  
variables[N\_out+N\_in+1..] = internal signals

**Important**: Outputs come **before** inputs in the witness array (Circom convention).

### Example: ALU1 Subcircuit

template ALU1\_() {  
 signal input in[7]; // selector + 3x 256-bit inputs (2 limbs each)  
 signal output out[4]; // 2x 256-bit outputs (2 limbs each)  
 // ... internal signals  
}

**Real Data Example**:

{  
 "subcircuitId": 4,  
 "variables": [  
 "0x01", // [0] constant 1  
 "0x01", // [1] out[0] - first output limb  
 "0x00", // [2] out[1] - second output limb  
 "0x00", // [3] out[2] - (unused)  
 "0x00", // [4] out[3] - (unused)  
 "0x200000", // [5] in[0] - selector (2^21 = ISZERO opcode)  
 "0x00", // [6] in[1] - first input, lower limb  
 "0x00", // [7] in[2] - first input, upper limb  
 "0x00", // [8] in[3] - second input, lower limb  
 "0x00", // [9] in[4] - second input, upper limb  
 "0x00", // [10] in[5] - third input, lower limb  
 "0x00", // [11] in[6] - third input, upper limb  
 // [12+] hundreds of internal variables...  
 ]  
}

Interpretation:

* Placement uses ALU1 subcircuit (ID 4)
* Performs ISZERO operation ([selector](synthesizer-terminology.md#selector) = 0x200000 = 2^21)
  + For selector value mappings, see [Appendix: Subcircuit Mapping Table](synthesizer-opcodes.md#appendix-subcircuit-mapping-table)
* Input: 0x00 (256-bit zero, represented as two 0x00 [limbs](synthesizer-terminology.md#limb))
  + **Why 2 limbs?** Circom uses a 254-bit finite field, but Ethereum uses 256-bit numbers. To handle this, 256-bit values are split into two 128-bit limbs (lower and upper) to avoid field overflow.
* Output: 0x01 (256-bit one, represented as 0x01 lower limb, 0x00 upper limb)
* Logic: ISZERO(0) = 1 (true, input is zero)
* Variables [12+] contain intermediate calculation steps (bitify, comparisons, etc.)

### Subcircuit IDs

Complete subcircuit list (from [qap-compiler/subcircuits/library/subcircuitInfo.ts](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/qap-compiler/subcircuits/library/subcircuitInfo.ts)):

| ID | Name | Description | Inputs | Outputs |
| --- | --- | --- | --- | --- |
| 0 | bufferPubOut | Public output buffer (RETURN data, event logs) | 40 | 40 |
| 1 | bufferPubIn | Public input buffer (calldata, tx data) | 20 | 20 |
| 2 | bufferPrvOut | Private output buffer (storage updates) | 40 | 40 |
| 3 | bufferPrvIn | Private input buffer (storage values, bytecode) | 512 | 512 |
| 4 | ALU1 | ADD, MUL, SUB, EQ, ISZERO, NOT, SubEXP | 7 | 4 |
| 5 | ALU2 | DIV, SDIV, MOD, SMOD, ADDMOD, MULMOD | 7 | 2 |
| 6 | ALU3 | SHL, SHR, SAR (bit shifts) | 7 | 2 |
| 7 | ALU4 | LT, GT, SLT, SGT (comparisons) | 7 | 2 |
| 8 | ALU5 | SIGNEXTEND, BYTE | 7 | 2 |
| 9 | OR | Bitwise OR | 4 | 2 |
| 10 | XOR | Bitwise XOR | 4 | 2 |
| 11 | AND | Bitwise AND | 4 | 2 |
| 12 | DecToBit | Number to bit array conversion (for memory operations) | 2 | 256 |
| 13 | Accumulator | Bit array to number conversion (for memory reconstruction) | 64 | 2 |