# Synthesizer: Execution Flow

This document walks through how Synthesizer processes Ethereum transactions, showing the complete execution flow from input to output with practical examples.

## Transaction Lifecycle Overview

The following diagram shows the complete flow of a transaction through Synthesizer:

┌──────────────────────────────────────────────────────────────────────────────┐  
│ SYNTHESIZER TRANSACTION FLOW │  
└──────────────────────────────────────────────────────────────────────────────┘  
  
 INPUT PROCESSING OUTPUT  
  
┌─────────────┐ ┌─────────────────┐ ┌──────────────┐  
│ │ │ │ │ │  
│ Ethereum │ │ Initialization │ │ │  
│ Transaction │ │ │ │ │  
│ (0x...) │ │ │ │ │  
│ │ │ │ │ │  
│─────────────│ │ │ │ │  
│ │ │ │ │ │  
│ Subcircuit │ │ │ │ │  
│ Library │──────────► └────────┬────────┘ │ │  
│ │ │ │ permutation │  
│ │ ▼ │ .json │  
│─────────────│ ┌─────────────────┐ │ │  
│ │ │ │ │ instance │  
│ RPC Data │ │ EVM + Symbol │ │ .json │  
│ │ │ Execution │ │ │  
│ (On-demand) │ │ │ │ placement │  
│ │ │ │ │ Variables │  
└─────────────┘ │ │ │ .json │  
 │ │ │ │  
 │ │ │ │  
 │ │ │ │  
 │ │ │ │  
 │ │ │ │  
 └────────┬────────┘ │ │  
 │ │ │  
 ▼ │ │  
 ┌─────────────────┐ │ │  
 │ │ │ │  
 │ Finalization │ ───────► │ │  
 │ │ │ │  
 └─────────────────┘ └──────────────┘

**What flows through**:

* **Transaction Hash** → Fetches transaction details and triggers re-execution
* **Subcircuit Library** → Provides circuit templates (.wasm, .ts) used during execution
* **RPC Provider** → Supplies blockchain state (storage, balances, code) on-demand throughout execution

The transaction flows through **6 main steps**, which we’ll explore in detail below.

## Step-by-Step Transaction Processing

### Step 1: Setup & Preparation

Before processing the transaction, Synthesizer prepares its environment:

┌────────────────────────────────────────────────────────┐  
│ 1. Compile Subcircuit Library (One-time setup) │  
└────────────────────────────────────────────────────────┘  
 │  
 │  
 │  
 │  
 ▼  
 Generate .wasm files (subcircuit0.wasm ... subcircuitN.wasm)  
 Generate TypeScript definitions (globalWireList.ts, subcircuitInfo.ts)  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ 2. Configure RPC Provider │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Set up RPC endpoint for Ethereum Mainnet  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ 3. Provide Transaction Hash │  
└────────────────────────────────────────────────────────┘  
 │  
 │ TX: 0x123abc...  
 │  
 ▼  
 Ready to process transaction

**What happens here:**

1. **Subcircuit Library Compilation**: Circom compiles all the fundamental circuits (ALU1, bitify, XOR, etc.) into WebAssembly files. These are the building blocks that Synthesizer will use to construct the transaction-specific circuit.
2. **RPC Connection**: Synthesizer connects to Ethereum Mainnet via RPC to access blockchain state. This connection is essential because Synthesizer needs to:
   * Fetch transaction details (from, to, data, value)
   * Access account states at the transaction’s block height
   * Query storage values on-demand during execution
   * Retrieve block information (number, timestamp, coinbase, etc.)
3. **Transaction Selection**: You provide the transaction hash of an already-executed Ethereum transaction. Synthesizer will re-execute this transaction to generate the circuit.

**Important**: The RPC connection remains active throughout execution, not just during initialization. When the EVM encounters SLOAD, BALANCE, EXTCODESIZE, etc., it queries the blockchain state through RPC in real-time.

### Step 2: Initialization

When you invoke Synthesizer, it creates the execution environment:

┌────────────────────────────────────────────────────────┐  
│ createEVM() is called │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Fetch transaction data from RPC  
 │ Fetch block data from RPC  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ EVM instance created │  
│ - Synthesizer instance attached │  
│ - Opcode handlers registered │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Synthesizer creates internal managers: │  
│ - StateManager (holds Placements map) │  
│ - OperationHandler (arithmetic ops) │  
│ - DataLoader (external data) │  
│ - MemoryManager (memory aliasing) │  
│ - BufferManager (LOAD/RETURN buffers) │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Interpreter created with: │  
│ - Stack (EVM values) │  
│ - StackPt (Synthesizer symbols) │  
│ - Memory (EVM bytes) │  
│ - MemoryPt (Synthesizer symbols with time tracking) │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
 Ready to execute bytecode

**What happens here:**

The EVM is instantiated with an attached [Synthesizer](synthesizer-terminology.md#synthesizer). Think of it as running two virtual machines in parallel:

* **Standard EVM**: Processes the transaction normally, updating stack/memory/storage
* **Synthesizer**: Shadows the EVM execution, tracking everything as mathematical [symbols](synthesizer-terminology.md#symbol-processing)

At this point:

* The [Placements](synthesizer-terminology.md#placement) map is empty (will be populated during execution)
* [Buffer placements](synthesizer-terminology.md#buffer-placements) (IDs 0-3) are pre-initialized for LOAD and RETURN operations
* Both Stack and [StackPt](synthesizer-terminology.md#stackpt) are empty
* Both Memory and [MemoryPt](synthesizer-terminology.md#memorypt) are empty

### Step 3: Bytecode Execution (Dual Processing)

Now the interpreter begins executing the transaction bytecode. For **every single opcode**, both the EVM and Synthesizer process it in parallel:

┌────────────────────────────────────────────────────────────────────┐  
│ For each opcode in transaction bytecode: │  
└────────────────────────────────────────────────────────────────────┘  
 │  
 ┌───────────────┴───────────────┐  
 │ │  
 ▼ ▼  
 ┌─────────────────────────┐ ┌─────────────────────────┐  
 │ EVM Handler executes │ │ Synthesizer Handler │  
 │ │ │ executes │  
 │ • Pop from Stack │ │ • Pop from StackPt │  
 │ • Compute result │ │ • Create placement │  
 │ • Push to Stack │ │ with output symbol │  
 │ • Update Memory/Storage│ │ • Push to StackPt │  
 │ │ │ │  
 └─────────────┬───────────┘ └───────────┬─────────────┘  
 │ │  
 └──────────────┬──────────────┘  
 │  
 ▼  
 ┌─────────────────────────┐  
 │ Consistency Check │  
 │ Stack == StackPt ? │  
 │ If not → Error │  
 └────────────┬────────────┘  
 │  
 ▼  
 Continue to next opcode

**What happens here:**

This is the core of Synthesizer. For example, when processing ADD:

**EVM side:**

1. Pops two values: a = 10, b = 5
2. Computes: result = 15
3. Pushes 15 to Stack

**Synthesizer side:**

1. Pops two symbols: x, y (where x.value = 10, y.value = 5)
2. Creates a new [placement](synthesizer-terminology.md#placement): ADD\_placement = ALU1(x, y)
3. Creates output symbol: z (where z.value = 15, z.source = ADD\_placement)
4. Pushes z to [StackPt](synthesizer-terminology.md#stackpt)
5. Records: Placements[4] = { name: "ALU1", usage: "ADD", subcircuitId: 4, inPts: [x, y], outPts: [z] }

After every opcode, Synthesizer verifies that Stack[i].value == StackPt[i].value for all elements. This ensures the symbolic execution matches the actual execution.

**Key insight**: Synthesizer is not simulating the EVM—it’s **shadowing** it. The EVM computes the actual values, while Synthesizer builds a mathematical proof of how those values were derived.

### Step 4: Symbol Loading & Returning

Throughout execution, Synthesizer needs to convert between external values and internal symbols:

┌────────────────────────────────────────────────────────┐  
│ Loading External Data (LOAD Buffer) │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Examples: CALLDATALOAD, SLOAD, BLOCKHASH  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ External value → Symbol conversion │  
│ │  
│ calldata[0] = 0x05 → x (symbol) │  
│ storage[key] = 0x0a → y (symbol) │  
│ block.number = 19000 → z (symbol) │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Symbols flow through circuit  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Symbol undergoes transformations │  
│ │  
│ x' = ADD(x, y) │  
│ x'' = MUL(x', constant) │  
│ x''' = AND(x'', mask) │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Examples: SSTORE, LOG  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Returning to External World (RETURN Buffer) │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Symbol → External value conversion │  
│ │  
│ x''' (symbol) → storage[key] = 0x14 │  
│ y'' (symbol) → log.data = 0x... │  
└────────────────────────────────────────────────────────┘

**What happens here:**

Buffers act as the **interface** between the external world (Ethereum state) and the internal circuit world (symbols):

* **LOAD Buffer** ([Placement](synthesizer-terminology.md#placement) IDs 0, 2): Takes concrete values and produces symbols
  + Public inputs: calldata, block info, msg.sender ([PUB\_IN](synthesizer-terminology.md#pub-in-and-pub-out) - Placement 0)
  + Private inputs: storage values, account states ([PRV\_IN](synthesizer-terminology.md#prv-in-and-prv-out) - Placement 2)
* **RETURN Buffer** (Placement IDs 1, 3): Takes symbols and produces concrete outputs
  + Public outputs: logs, return data ([PUB\_OUT](synthesizer-terminology.md#pub-in-and-pub-out) - Placement 1)
  + Private outputs: storage updates ([PRV\_OUT](synthesizer-terminology.md#prv-in-and-prv-out) - Placement 3)

This is crucial for zero-knowledge proofs: public inputs/outputs are revealed, while private inputs/outputs remain hidden.

### Step 5: Memory Aliasing Resolution

One of Synthesizer’s most complex tasks is tracking overlapping memory writes:

┌────────────────────────────────────────────────────────┐  
│ Time 0: MSTORE at offset 0x00 │  
│ Store symbol x (32 bytes) │  
└────────────────────────────────────────────────────────┘  
 │  
 │ MemoryPt[0x00-0x20] = { time: 0, symbol: x }  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Time 1: MSTORE at offset 0x10 │  
│ Store symbol y (32 bytes) │  
└────────────────────────────────────────────────────────┘  
 │  
 │ MemoryPt[0x10-0x30] = { time: 1, symbol: y }  
 │ (Overlaps with x at 0x10-0x20!)  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Time 2: MLOAD at offset 0x00-0x20 │  
│ Need to reconstruct the value! │  
└────────────────────────────────────────────────────────┘  
 │  
 │ Memory region 0x00-0x20 now contains:  
 │ - Bytes 0x00-0x0F: from x (unchanged)  
 │ - Bytes 0x10-0x1F: from y (overwrote x)  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Synthesizer creates reconstruction circuit: │  
│ │  
│ 1. Extract first 16 bytes of x │  
│ x\_low = SHR(x, 128) & 0xFFFF... │  
│ │  
│ 2. Extract first 16 bytes of y │  
│ y\_low = SHR(y, 128) & 0xFFFF... │  
│ │  
│ 3. Combine them │  
│ result = SHL(x\_low, 128) | y\_low │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ New placement added for reconstruction │  
│ StackPt.push(result symbol) │  
└────────────────────────────────────────────────────────┘

**What happens here:**

Traditional EVM simply overwrites memory and returns the latest value. But Synthesizer must prove **how** that value was computed from the original symbols.

The 2D structure of [MemoryPt](synthesizer-terminology.md#memorypt) (offset × time) allows Synthesizer to:

1. Track all writes to each memory location
2. Detect overlaps when reading
3. Generate [subcircuits](synthesizer-terminology.md#subcircuit) (using SHR, SHL, AND, OR) to reconstruct the correct value
4. Prove the reconstruction is correct

This is why memory operations can generate multiple [placements](synthesizer-terminology.md#placement)—they need to prove [data aliasing](synthesizer-terminology.md#data-aliasing).

### Step 6: Finalization & Output Generation

After bytecode execution completes, Synthesizer generates the final output files:

┌────────────────────────────────────────────────────────┐  
│ Bytecode execution finished │  
│ - All opcodes processed │  
│ - Placements map populated │  
│ - Symbol graph complete │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Finalizer analyzes Placements │  
│ │  
│ For each placement: │  
│ - Extract input wire indices │  
│ - Extract output wire indices │  
│ - Track wire connections between placements │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Generate permutation.json │  
│ │  
│ Wire connection map: │  
│ [ │  
│ { row: 13, col: 1, X: 14, Y: 3 }, │  
│ { row: 27, col: 2, X: 8, Y: 5 }, │  
│ ... │  
│ ] │  
│ │  
│ Meaning: Wire 13 of Placement 1 connects to │  
│ Wire 14 of Placement 3 │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Generate instance.json │  
│ │  
│ Public/Private witness values: │  
│ { │  
│ "publicInputBuffer": [...], // From PUB\_IN │  
│ "publicOutputBuffer": [...], // From PUB\_OUT │  
│ "privateInputBuffer": [...], // From PRV\_IN │  
│ "privateOutputBuffer": [...],// From PRV\_OUT │  
│ "a\_pub": [...], // Public witness array │  
│ "a\_prv": [...] // Private witness array │  
│ } │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Generate placementVariables.json │  
│ │  
│ Complete witness for each placement: │  
│ [ │  
│ { │  
│ "subcircuitId": 4, │  
│ "variables": ["0x01", "0x04", ...] │  
│ }, │  
│ ... │  
│ ] │  
└────────────────────────────────────────────────────────┘  
 │  
 ▼  
┌────────────────────────────────────────────────────────┐  
│ Output files ready for backend prover │  
│ - permutation.json (circuit topology) │  
│ - instance.json (public/private I/O) │  
│ - placementVariables.json (complete witness) │  
└────────────────────────────────────────────────────────┘

**What happens here:**

The [Finalizer](synthesizer-terminology.md#finalizer) converts the Placements map into three critical files:

1. **permutation.json**: Describes the circuit topology
   * How subcircuit [wires](synthesizer-terminology.md#wire) are connected
   * PLONK-style [Permutation](synthesizer-terminology.md#permutation) Argument
   * Used by Prove, Verify stages
2. **instance.json**: Contains the actual input/output values
   * Public values are revealed (anyone can see)
   * Private values remain hidden (only prover knows)
   * Contains both buffer data and complete [witness](synthesizer-terminology.md#witness) arrays
3. **placementVariables.json**: Full [witness](synthesizer-terminology.md#witness) for proof generation
   * All intermediate values for each [subcircuit](synthesizer-terminology.md#subcircuit)
   * Needed by the prover to satisfy constraints
   * Maps to [R1CS](synthesizer-terminology.md#r1cs) format used by Tokamak zk-SNARK

These files are then passed to the backend Rust prover, which generates the actual zero-knowledge proof.