# Synthesizer: Transaction Processing Flow

This document provides a detailed walkthrough of how Synthesizer processes Ethereum transactions, from initialization to finalization, with code-level details at each step.

## Overview

The transaction processing follows four main phases:

1. **Initialization** - Create EVM and Synthesizer instances
2. **Execution Setup** - Prepare Interpreter with dual state
3. **Bytecode Execution** - Process each opcode with dual handlers
4. **Finalization** - Generate output files

## Code Execution Flow Overview

This diagram shows the key function calls and code paths from transaction input to circuit output:

┌─────────────────────────────────────────────────────────────────────────┐  
│ PHASE 1: INITIALIZATION │  
└─────────────────────────────────────────────────────────────────────────┘  
  
 createEVM()  
 ├─► new EVM()  
 │ └─► this.synthesizer = new Synthesizer()  
 │ └─► new StateManager()  
 │ └─► \_initializePlacements() // IDs 0-3  
 └─► new RPCStateManager()  
  
┌─────────────────────────────────────────────────────────────────────────┐  
│ PHASE 2: EXECUTION SETUP │  
└─────────────────────────────────────────────────────────────────────────┘  
  
 EVM.runCall(txData)  
 └─► new Interpreter(evm, stateManager, ..., synthesizer)  
 └─► this.\_runState = {  
 stack, stackPt, // Dual execution state  
 memory, memoryPt,  
 synthesizer  
 }  
  
┌─────────────────────────────────────────────────────────────────────────┐  
│ PHASE 3: BYTECODE EXECUTION (Loop for each opcode) │  
└─────────────────────────────────────────────────────────────────────────┘  
  
 Interpreter.runStep()  
 ├─► opFn(runState, common) // Unified handler  
 │ ├─► EVM: stack.push(result)  
 │ └─► Synthesizer: synthesizerArith() / loadStorage() / placeMemoryToStack()  
 │ └─► StateManager.placements.set(id, placement)  
 │  
 └─► Consistency check: stack[i] === stackPt[i].value  
  
┌─────────────────────────────────────────────────────────────────────────┐  
│ PHASE 4: FINALIZATION │  
└─────────────────────────────────────────────────────────────────────────┘  
  
 Finalizer.exec()  
 ├─► PlacementRefactor.refactor()  
 ├─► new Permutation(placements)  
 │ ├─► \_buildPermGroup() // Group wires by parent-child  
 │ └─► \_correctPermutation() // Generate 3-entry cycles  
 ├─► outputPlacementVariables() // Calculate witness via WASM  
 └─► outputInstance() // Extract buffer values  
  
 Output: permutation.json, instance.json, placementVariables.json

## Detailed Phase-by-Phase Breakdown

### Phase 1: Initialization

**What happens:**

* EVM and Synthesizer instances created
* StateManager initializes buffer placements (0-3)
* RPC connection established for on-demand state queries

**Detailed Flow:**

User Call  
 │  
 ▼  
createEVM() [constructors.ts:19]  
 │  
 ├─► Create EVM instance [evm.ts:74]  
 │ └─► new Synthesizer() [evm.ts:271]  
 │ └─► new StateManager() [synthesizer/index.ts:37]  
 │ ├─► initializeState()  
 │ ├─► initializeSubcircuitInfo()  
 │ └─► initializePlacements() (IDs 0-3: Buffers)  
 │  
 └─► Create RPCStateManager [constructors.ts:30]  
 └─► Fetch transaction & block data from RPC

**Key Code:**

// constructors.ts:19 - Entry point  
export async function createEVM(opts?: EVMOpts) {  
 const evm = new EVM(opts);  
 return evm;  
}  
  
// evm.ts:271 - Synthesizer instantiation  
constructor(opts: EVMOpts) {  
 // ... original EthereumJS initialization  
 this.synthesizer = new Synthesizer(); // Tokamak addition  
}  
  
// synthesizer/index.ts:37 - StateManager initialization  
constructor() {  
 this.\_state = new StateManager();  
 this.\_state.\_initializePlacements(); // Creates buffer placements 0-3  
}

[**Buffer Placements**](synthesizer-terminology.md#buffer-placements) **(IDs 0-3):**

* **Placement 0 (**[**PUB\_IN**](synthesizer-terminology.md#pub-in-and-pub-out)**)**: Public input buffer (calldata, block info, msg.sender)
* **Placement 1 (**[**PUB\_OUT**](synthesizer-terminology.md#pub-in-and-pub-out)**)**: Public output buffer (return data, logs)
* **Placement 2 (**[**PRV\_IN**](synthesizer-terminology.md#prv-in-and-prv-out)**)**: Private input buffer (storage, account state)
* **Placement 3 (**[**PRV\_OUT**](synthesizer-terminology.md#prv-in-and-prv-out)**)**: Private output buffer (storage updates)

### Phase 2: Execution Setup

**What happens:**

* Interpreter created with dual state (Stack/StackPt, Memory/MemoryPt)
* Message wraps transaction data
* RunState prepared with all necessary references

**Detailed Flow:**

EVM.runCall() [evm.ts:858]  
 │  
 ├─► Create Message [message.ts:48]  
 │  
 ├─► Create Interpreter [interpreter.ts:152]  
 │ └─► Initialize RunState [interpreter.ts:217]  
 │ ├─► Stack (EVM)  
 │ ├─► StackPt (Synthesizer)  
 │ ├─► Memory (EVM)  
 │ ├─► MemoryPt (Synthesizer)  
 │ └─► synthesizer reference  
 │  
 └─► Interpreter.run() [interpreter.ts:300]

**Key Code:**

// evm.ts:858 - Create Interpreter with Synthesizer  
async runCall(opts: EVMRunCallOpts): Promise<EVMResult> {  
 const interpreter = new Interpreter(  
 this,  
 this.stateManager,  
 // ... other params  
 this.synthesizer // Pass Synthesizer to interpreter  
 );  
 return interpreter.run(message);  
}  
  
// interpreter.ts:217 - RunState with dual structures  
this.\_runState = {  
 // EVM state  
 stack: new Stack(),  
 memory: new Memory(),  
  
 // Synthesizer state (parallel processing)  
 stackPt: new StackPt(),  
 memoryPt: new MemoryPt(),  
 synthesizer: synthesizer,  
};

**Dual State Structure:**

* **EVM State**: Stack, Memory - Track actual execution values
* [**Synthesizer**](synthesizer-terminology.md#synthesizer) **State**: [StackPt](synthesizer-terminology.md#stackpt), [MemoryPt](synthesizer-terminology.md#memorypt) - Track [symbolic](synthesizer-terminology.md#symbol-processing) representations
* Both states are maintained in parallel and verified for consistency

### Phase 3: Bytecode Execution

**What happens:**

* Each opcode triggers both EVM and Synthesizer handlers
* Arithmetic ops → OperationHandler → Create placements
* Storage ops → DataLoader → Buffer management
* Memory ops → MemoryManager → Aliasing resolution
* Consistency checks ensure EVM and Synthesizer stay synchronized

**Detailed Flow:**

Interpreter.runStep() [interpreter.ts:384]  
 │  
 ├─► Parse opcode from bytecode  
 │  
 ├─► Execute Unified Handler [opcodes/functions.ts]  
 │ │ (Contains both EVM + Synthesizer logic)  
 │ │  
 │ ├─► 1. EVM Logic: Update Stack, Memory, Storage  
 │ │  
 │ └─► 2. Synthesizer Logic: Create placements/symbols  
 │ │  
 │ ├─► Arithmetic ops  
 │ │ └─► OperationHandler.placeArith()  
 │ │ └─► Create ALU placement  
 │ │  
 │ ├─► Storage ops  
 │ │ └─► DataLoader.loadStorage()  
 │ │ └─► Add to PRV\_IN buffer  
 │ │  
 │ └─► Memory ops  
 │ └─► MemoryManager.placeMemoryToStack()  
 │ └─► Create reconstruction circuit  
 │  
 └─► Consistency Check [interpreter.ts:441-449]  
 └─► Verify Stack values == StackPt values

**Key Code:**

// interpreter.ts:384-449 - Opcode execution  
async runStep(opcodeObj?: OpcodeMapEntry): Promise<void> {  
 const opEntry = opcodeObj ?? this.lookupOpInfo(this.\_runState.opCode);  
 const opInfo = opEntry.opcodeInfo;  
  
 // ... gas calculation and program counter advance ...  
  
 // Execute opcode handler (contains both EVM and Synthesizer logic)  
 const opFn = opEntry.opHandler;  
  
 if (opInfo.isAsync) {  
 await (opFn as AsyncOpHandler).apply(null, [this.\_runState, this.common]);  
 } else {  
 opFn.apply(null, [this.\_runState, this.common]);  
 }  
  
 // Verify consistency between EVM and Synthesizer  
 const stackVals = this.\_runState.stack.getStack();  
 const stackPtVals = this.\_runState.stackPt.getStack().map(dataPt => dataPt.value);  
 if (!(stackVals.length === stackPtVals.length &&  
 stackVals.every((val, index) => val === stackPtVals[index]))) {  
 console.log(`Instruction: ${opInfo.name}`);  
 console.log(`Stack values(right-newest): ${stackVals}`);  
 console.log(`StackPt values(right-newest): ${stackPtVals}`);  
 throw new Error('Synthesizer: Stack mismatch between EVM and Synthesizer');  
 }  
}  
  
// opcodes/functions.ts:95 - Handler definition (unified EVM + Synthesizer)  
export const handlers: Map<number, OpHandler> = new Map([  
 // 0x01: ADD  
 [  
 0x01,  
 function (runState) {  
 // 1. EVM execution (original EthereumJS logic)  
 const [a, b] = runState.stack.popN(2);  
 const r = mod(a + b, TWO\_POW256);  
 runState.stack.push(r);  
  
 // 2. Synthesizer execution (Tokamak addition)  
 synthesizerArith('ADD', [a, b], r, runState);  
 },  
 ],  
 // ... more opcodes  
]);  
  
// Example: ADD operation creates a placement  
// operationHandler.ts:80  
public placeArith(name: ArithmeticOperator, inPts: DataPt[]): DataPt[] {  
 const [subcircuitName, selector] = SUBCIRCUIT\_MAPPING[name]; // 'ADD' → ['ALU1', 2n]  
 const outPt = this.createOutput(name, inPts);  
  
 // Record placement in circuit  
 this.provider.place(subcircuitName, [selectorPt, ...inPts], [outPt], name);  
 return [outPt];  
}  
  
// See also: [DataPt](synthesizer-terminology.md#datapt-data-point), [Subcircuit](synthesizer-terminology.md#subcircuit), [Selector](synthesizer-terminology.md#selector)

**Opcode Processing Examples:**

For detailed code walkthroughs of opcode processing, see the following examples:

* [**Example 1: Arithmetic Operation (ADD)**](./synthesizer-code-examples.md#1-arithmetic-operation-add) - How arithmetic operations create placements
* [**Example 2: Storage Load (SLOAD)**](./synthesizer-code-examples.md#2-storage-load-sload) - Buffer management and external data loading
* [**Example 3: Memory Load with Aliasing (MLOAD)**](./synthesizer-code-examples.md#3-memory-load-with-aliasing-mload) - Memory aliasing resolution with reconstruction circuits

### Phase 4: Finalization

**What happens:**

* Placements map converted to output files
* Witness calculated for each placement using WASM
* Three JSON files generated for backend prover

**Detailed Flow:**

Finalizer.exec() [finalizer/index.ts:12]  
 │  
 ├─► PlacementRefactor.refactor() [placementRefactor.ts:30]  
 │ └─► Optimize wire sizes  
 │  
 ├─► new Permutation() [permutation.ts:84]  
 │ │  
 │ ├─► \_buildPermGroup() [permutation.ts:441]  
 │ │ ├─► Group wires by value  
 │ │ └─► Create parent-child relationships  
 │ │  
 │ └─► \_correctPermutation() [permutation.ts:368]  
 │ └─► Generate 3-entry cycles  
 │ └─► Write permutation.json  
 │  
 ├─► outputPlacementVariables() [permutation.ts:123]  
 │ ├─► For each placement:  
 │ │ ├─► Load subcircuitN.wasm  
 │ │ ├─► generateSubcircuitWitness() [permutation.ts:613]  
 │ │ │ └─► witnessCalculator.calculateWitness()  
 │ │ └─► Validate outputs  
 │ └─► Write placementVariables.json  
 │  
 └─► outputInstance() [instance.ts]  
 └─► Write instance.json

**Key Code:**

// finalizer/index.ts:12  
public async exec(\_path?: string): Promise<Permutation> {  
 // 1. Optimize placements  
 const refactored = new PlacementRefactor(this.state).refactor();  
  
 // 2. Generate wire connections and witness  
 const permutation = new Permutation(refactored, \_path);  
 permutation.placementVariables = await permutation.outputPlacementVariables(  
 refactored,  
 \_path,  
 );  
  
 // 3. Write output files  
 permutation.outputPermutation(\_path); // → permutation.json  
 outputInstance(\_path); // → instance.json  
 // → placementVariables.json  
  
 return permutation;  
}  
  
// permutation.ts:613 - Calculate witness for each placement  
async generateSubcircuitWitness(placement: PlacementEntry): Promise<bigint[]> {  
 const wasmPath = `./subcircuit${placement.subcircuitId}.wasm`;  
 const witness = await witnessCalculator.calculateWitness(inputs);  
 return witness; // All internal circuit values  
}

**Output Files:**

1. **permutation.json** - Circuit topology ([wire](synthesizer-terminology.md#wire) connections)
   * Describes how wires between [placements](synthesizer-terminology.md#placement) are connected
   * Uses N-entry cycle structure for equality constraints ([Permutation](synthesizer-terminology.md#permutation))
   * Used by Setup, Prove, Verify stages
   * Example: { row: 13, col: 1, X: 14, Y: 3 } means wire 13 in Placement 1 connects to wire 14 in Placement 3
2. **instance.json** - Public/Private I/O values ([Instance](synthesizer-terminology.md#instance))
   * Public input/output from [PUB\_IN](synthesizer-terminology.md#pub-in-and-pub-out)/[PUB\_OUT](synthesizer-terminology.md#pub-in-and-pub-out) buffers
   * Private input/output from [PRV\_IN](synthesizer-terminology.md#prv-in-and-prv-out)/[PRV\_OUT](synthesizer-terminology.md#prv-in-and-prv-out) buffers
   * Complete [witness](synthesizer-terminology.md#witness) arrays (a\_pub, a\_prv)
   * Used by Prove stage
3. **placementVariables.json** - Complete [witness](synthesizer-terminology.md#witness) for all [placements](synthesizer-terminology.md#placement)
   * All intermediate values for each [subcircuit](synthesizer-terminology.md#subcircuit) instance
   * Needed by prover to satisfy [R1CS](synthesizer-terminology.md#r1cs) constraints
   * Maps to Tokamak zk-SNARK format

**For detailed information on output file formats, see** [**Output Files Reference**](./synthesizer-output-files.md)