# Synthesizer: Class Structure

This document provides detailed information about [Synthesizer](synthesizer-terminology.md#synthesizer)’s class structure, relationships, and implementation details.

## Synthesizer Class Hierarchy

┌─────────────────────────────────────────────────────────┐  
│ Synthesizer │  
│ ┌────────────────────────────────────────────────────┐ │  
│ │ StateManager │ │  
│ │ - placements: Map<number, PlacementEntry> │ │  
│ │ - auxin: Auxin (auxiliary inputs) │ │  
│ │ - storagePt, logPt, keccakPt, etc. │ │  
│ │ - placementIndex: number │ │  
│ └────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────┐ │  
│ │ OperationHandler │ │  
│ │ - placeArith(op, inputs) │ │  
│ │ - placeExp(base, exponent) │ │  
│ └────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────┐ │  
│ │ DataLoader │ │  
│ │ - loadStorage(addr, key) │ │  
│ │ - storeStorage(addr, key, value) │ │  
│ │ - loadEnvInf/loadBlkInf │ │  
│ └────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────┐ │  
│ │ MemoryManager │ │  
│ │ - placeMemoryToStack(aliasInfos) │ │  
│ └────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────┐ │  
│ │ BufferManager │ │  
│ │ - addWireToInBuffer(val, placementId) │ │  
│ │ - addWireToOutBuffer(sym, val, placementId) │ │  
│ └────────────────────────────────────────────────────┘ │  
└─────────────────────────────────────────────────────────┘

## Detailed Class Breakdown

### 1. Synthesizer Class

**Location**: src/tokamak/core/synthesizer/index.ts:27-181

**Role**: Central coordinator using Facade pattern

**Architecture**:

export class Synthesizer  
 implements ISynthesizerProvider, IDataLoaderProvider, IMemoryManagerProvider  
{  
 private \_state: StateManager; // Line 30  
 private operationHandler: OperationHandler; // Line 31  
 private dataLoader: DataLoader; // Line 32  
 private memoryManager: MemoryManager; // Line 33  
 private bufferManager: BufferManager; // Line 34  
  
 constructor() {  
 this.\_state = new StateManager();  
 this.operationHandler = new OperationHandler(this, this.\_state);  
 this.dataLoader = new DataLoader(this, this.\_state);  
 this.memoryManager = new MemoryManager(this, this.\_state);  
 this.bufferManager = new BufferManager(this, this.\_state);  
 }  
  
 public get state(): StateManager {  
 return this.\_state;  
 }  
  
 // Delegate to handlers  
 public placeArith(name: ArithmeticOperator, inPts: DataPt[]): DataPt[] {  
 return this.operationHandler.placeArith(name, inPts);  
 }  
  
 public loadStorage(codeAddress: string, key: bigint, value: bigint): DataPt {  
 return this.dataLoader.loadStorage(codeAddress, key, value);  
 }  
  
 // ... more delegation methods  
}

**Design**: Facade pattern delegates to specialized handlers

### 2. StateManager Class

**Location**: src/tokamak/core/handlers/stateManager.ts:24-102

**Role**: Central state repository

**Key Data Structures**:

export class StateManager {  
 public placements!: Placements; // All placement instances  
 public auxin!: Auxin; // Auxiliary inputs  
 public envInf!: Map<string, {...}>; // Environment info (CALLER, etc.)  
 public blkInf!: Map<string, {...}>; // Block info (NUMBER, etc.)  
 public storagePt!: Map<string, DataPt>; // Storage symbols  
 public logPt!: {...}[]; // Log data  
 public keccakPt!: {...}[]; // Keccak inputs/outputs  
 public TStoragePt!: Map<...>; // Transient storage  
 public placementIndex!: number; // Sequential counter  
 public subcircuitInfoByName!: SubcircuitInfoByName;  
 public subcircuitNames!: SubcircuitNames[];  
  
 constructor() {  
 this.\_initializeState(); // Reset all state  
 this.\_initializeSubcircuitInfo(); // Load subcircuit metadata  
 this.\_initializePlacements(); // Create buffer placements (0-3)  
 this.placementIndex = INITIAL\_PLACEMENT\_INDEX; // Start from 4  
 }  
  
 public getNextPlacementIndex(): number {  
 return this.placementIndex++; // Atomic increment  
 }  
}

**Key Points**:

* Single source of truth for all Synthesizer state
* [Placements](synthesizer-terminology.md#placement) 0-3 reserved for [buffers](synthesizer-terminology.md#buffer-placements)
* Placement IDs start from 4

### 3. OperationHandler Class

**Location**: src/tokamak/core/handlers/operationHandler.ts

**Role**: Create [placements](synthesizer-terminology.md#placement) for arithmetic/logic operations

**Key Method**:

public placeArith(name: ArithmeticOperator, inPts: DataPt[]): DataPt[] {  
 // 1. Map operation to subcircuit  
 const [subcircuitName, selector] = SUBCIRCUIT\_MAPPING[name];  
  
 // 2. Create selector DataPt  
 const selectorPt = DataPointFactory.create({  
 source: 'literal',  
 value: selector,  
 // ...  
 });  
  
 // 3. Create output DataPt  
 const outPt = DataPointFactory.create({  
 source: this.state.getNextPlacementIndex(), // New placement ID  
 wireIndex: outWireIndex,  
 value: computedValue,  
 // ...  
 });  
  
 // 4. Call Synthesizer.place()  
 this.provider.place(  
 subcircuitName,  
 [selectorPt, ...inPts],  
 [outPt],  
 name  
 );  
  
 return [outPt];  
}

### 4. DataLoader Class

**Location**: src/tokamak/core/handlers/dataLoader.ts

**Role**: Handle external data (storage, environment, block info)

**Key Methods**:

public loadStorage(codeAddress: string, key: bigint, value: bigint): DataPt  
public storeStorage(codeAddress: string, key: bigint, inPt: DataPt): void  
public loadEnvInf(name: EnvInfNames, value: bigint): DataPt  
public loadBlkInf(name: BlkInfNames, value: bigint): DataPt  
public storeLog(valPts: DataPt[], topicPts: DataPt[]): void  
public loadAndStoreKeccak(inPts: DataPt[], outValue: bigint, length: bigint): DataPt

**Example**: loadStorage()

public loadStorage(codeAddress: string, key: bigint, value: bigint): DataPt {  
 const keyString = `${codeAddress}\_${key.toString()}`;  
  
 // Check if already loaded (warm access)  
 if (this.state.storagePt.has(keyString)) {  
 return this.state.storagePt.get(keyString)!;  
 }  
  
 // Cold access: load from PRV\_IN buffer  
 const inPt = DataPointFactory.create({ value, ... });  
 const outPt = this.provider.addWireToInBuffer(inPt, PRV\_IN\_PLACEMENT\_INDEX);  
  
 // Cache for future accesses  
 this.state.storagePt.set(keyString, outPt);  
  
 return outPt;  
}

### 5. MemoryManager Class

**Location**: src/tokamak/core/handlers/memoryManager.ts

**Role**: Resolve [memory aliasing](synthesizer-terminology.md#data-aliasing)

**Key Method**:

public placeMemoryToStack(dataAliasInfos: DataAliasInfos): DataPt {  
 // Generate subcircuits to reconstruct overlapping memory  
 // Uses SHR, SHL, AND, OR to combine fragments  
 // Returns reconstructed symbol  
}

**Used by**: MLOAD, CALLDATACOPY, KECCAK256, LOG, etc.

### 6. BufferManager Class

**Location**: src/tokamak/core/handlers/bufferManager.ts

**Role**: Manage LOAD and RETURN [buffer placements](synthesizer-terminology.md#buffer-placements)

**Key Methods**:

public addWireToInBuffer(inPt: DataPt, placementId: number): DataPt {  
 // Placement 0 (PUB\_IN) or 2 (PRV\_IN)  
 // External value → Symbol conversion  
  
 const outPt = DataPointFactory.create({  
 source: placementId,  
 wireIndex: nextIndex,  
 value: inPt.value,  
 // ...  
 });  
  
 this.state.placements.get(placementId)!.inPts.push(inPt);  
 this.state.placements.get(placementId)!.outPts.push(outPt);  
  
 return outPt; // Symbol for circuit  
}  
  
public addWireToOutBuffer(inPt: DataPt, outPt: DataPt, placementId: number): void {  
 // Placement 1 (PUB\_OUT) or 3 (PRV\_OUT)  
 // Symbol → External value conversion  
  
 this.state.placements.get(placementId)!.inPts.push(inPt);  
 this.state.placements.get(placementId)!.outPts.push(outPt);  
}

### 7. Finalizer Class

**Location**: src/tokamak/core/finalizer/index.ts:5-26

**Role**: Generate output files

**Execution Flow**:

export class Finalizer {  
 private state: StateManager;  
  
 constructor(stateManager: StateManager) {  
 this.state = stateManager;  
 }  
  
 public async exec(\_path?: string, writeToFS: boolean = true): Promise<Permutation> {  
 // 1. Refactor placements (optimize wire sizes)  
 const placementRefactor = new PlacementRefactor(this.state);  
 const refactoriedPlacements = placementRefactor.refactor();  
  
 // 2. Generate permutation and witness  
 const permutation = new Permutation(refactoriedPlacements, \_path);  
 permutation.placementVariables = await permutation.outputPlacementVariables(  
 refactoriedPlacements,  
 \_path,  
 );  
  
 // 3. Write permutation.json  
 permutation.outputPermutation(\_path);  
  
 return permutation;  
 }  
}