# Synthesizer: Data Structures

This document explains the core data structures used in the Tokamak [Synthesizer](synthesizer-terminology.md#synthesizer) for [symbol processing](synthesizer-terminology.md#symbol-processing) and circuit generation.

## On This Page

* [Overview](#overview)
* [DataPt (Data Point)](#datapt-data-point)
* [StackPt (Symbol Stack)](#stackpt-symbol-stack)
* [MemoryPt (Symbol Memory)](#memorypt-symbol-memory)
* [Placement](#placement)

## Overview

The Synthesizer uses **symbol-based processing** instead of value-based computation. This document explains the four core data structures that enable circuit generation:

┌─────────────────────────────────────────────────────────────────────┐  
│ Synthesizer Data Structures: From Values to Circuits │  
├─────────────────────────────────────────────────────────────────────┤  
│ │  
│ [1] DataPt (Data Point) │  
│ ┌────────────────────────────────────┐ │  
│ │ Symbol = Value + Traceability │ │  
│ │ { value: 10n, source: 0, wire: 0 }│ │  
│ └────────────────────────────────────┘ │  
│ ↓ ↓ │  
│ │  
│ [2] StackPt [3] MemoryPt │  
│ ┌──────────────┐ ┌──────────────────┐ │  
│ │ DataPt[] │ │ Map<time, DataPt>│ │  
│ │ (Symbolic │ │ (2D: offset×time)│ │  
│ │ Stack) │ │ (Aliasing track) │ │  
│ └──────────────┘ └──────────────────┘ │  
│ ↓ ↓ │  
│ └───────────┬───────────┘ │  
│ ↓ │  
│ │  
│ [4] Placement (Subcircuit Instance) │  
│ ┌──────────────────────────────────────┐ │  
│ │ name: 'ALU1' │ │  
│ │ usage: 'ADD' │ │  
│ │ inPts: [DataPt, DataPt] ──> Circuit │ │  
│ │ outPts: [DataPt] ───────────────────>│ │  
│ └──────────────────────────────────────┘ │  
│ │ │  
│ └──> Forms DAG (Circuit Graph) │  
│ │  
└─────────────────────────────────────────────────────────────────────┘

### Symbol-Based Processing

Each piece of data is represented as a **DataPt** (Data Point), which tracks:

* **Value**: The actual data (bigint)
* **Source**: Where the data came from ([placement](synthesizer-terminology.md#placement) ID or external source)
* [**Wire Index**](synthesizer-terminology.md#wire-index): Which output [wire](synthesizer-terminology.md#wire) from the source [subcircuit](synthesizer-terminology.md#subcircuit)
* **Metadata**: Type, size, and other context information

// EVM: Value-based processing  
const result = a + b; // Just the final value  
  
// Synthesizer: Symbol-based processing  
const aPt = { source: 0, wireIndex: 0, value: 10n, ... };  
const bPt = { source: 0, wireIndex: 1, value: 20n, ... };  
const resultPt = synthesizer.placeArith('ADD', [aPt, bPt]); // Tracks entire data flow

## DataPt (Data Point)

### Concept

DataPt is a **symbol** that represents data flowing through the circuit. Think of it as a “tracking tag” attached to each piece of data:

┌─────────────────────────────────────────────────────────────┐  
│ DataPt: A Symbol Tracking Data Flow │  
├─────────────────────────────────────────────────────────────┤  
│ │  
│ ┌──────────┐ source=4 ┌────────────┐ │  
│ │ Value │ wireIndex=0 │ Traceability│ │  
│ │ 15n │ ─────────────>│ Information │ │  
│ └──────────┘ └────────────┘ │  
│ ↓ │ │  
│ What data? Where from? │  
│ (actual value) (placement + wire) │  
│ │  
└─────────────────────────────────────────────────────────────┘

**Key Roles**:

1. **Traceability**: Track where data originates (source placement, wire index)
2. **Circuit Generation**: Connect subcircuit outputs to inputs via wire indices
3. **Value Tracking**: Maintain actual values for consistency checks
4. **External Interface**: Bridge between Ethereum state and circuit [symbols](synthesizer-terminology.md#symbol-processing)

### Visual: DataPt Journey

External Value DataPt Creation Circuit Usage  
────────────── ─────────────── ─────────────  
  
calldata[0] = 5 ─┐  
 ├──> DataPt { ┌──> Used as input  
 │ value: 5n, │ to ALU1  
 │ source: 0, ─────┤  
 │ wireIndex: 0, │  
Block info ─┤ type: 'Calldata' └──> Tracked in  
 │ } circuit graph  
Storage value ─┘

### Definition

**Source**: [types/synthesizer.ts:48-69](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/types/synthesizer.ts#L48-L69)

export interface CreateDataPointParams {  
 // Placement index at which the dataPt comes from  
 source: number;  
  
 // Wire index at which the dataPt comes from (within the source placement)  
 wireIndex?: number;  
  
 // Actual value of the data  
 value: bigint;  
  
 // Size of the data source (in bytes)  
 sourceSize: number;  
  
 // === External Source Information (if applicable) ===  
 // Address/identifier if data comes from external source  
 extSource?: string;  
  
 // Address/identifier if data goes to external destination  
 extDest?: string;  
  
 // Type of external data (e.g., 'CALLDATA', 'BLOCKHASH', 'Storage')  
 type?: string;  
  
 // Key if the external data comes from or goes to a DB (e.g., storage key)  
 key?: string;  
  
 // Offset if the external data comes from memory  
 offset?: number;  
}  
  
export type DataPt = CreateDataPointParams & { valueHex: string };

### Creation

**Source**: [pointers/dataPointFactory.ts:6-16](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/pointers/dataPointFactory.ts#L6-L16)

export class DataPointFactory {  
 public static create(params: CreateDataPointParams): DataPt {  
 SynthesizerValidator.validateValue(params.value);  
 const hex = params.value.toString(16);  
 const paddedHex = hex.length % 2 === 1 ? '0' + hex : hex;  
 const valueHex = '0x' + paddedHex;  
  
 return {  
 ...params,  
 valueHex,  
 };  
 }  
}

### Example: DataPt Lifecycle

// Step 1: Load external data as DataPt  
const calldataPt = synthesizer.loadEnvInf(  
 address,  
 'Calldata',  
 0x05n, // value  
 0, // offset  
 1 // size  
);  
// Result: { source: 0, wireIndex: 0, value: 5n, type: 'Calldata', ... }  
  
// Step 2: Use DataPt in arithmetic operation  
const constantPt = synthesizer.loadAuxin(10n);  
// Result: { source: 0, wireIndex: 1, value: 10n, ... }  
  
// Step 3: Generate circuit placement  
const resultPt = synthesizer.placeArith('ADD', [calldataPt, constantPt]);  
// Result: { source: 4, wireIndex: 0, value: 15n, ... }  
// source=4 means this comes from placement #4 (the ADD subcircuit)  
  
// Step 4: Store to external state  
synthesizer.storePrvOut(address, 'Storage', resultPt, storageKey);

### Key Fields Explained

| Field | Type | Purpose | Example |
| --- | --- | --- | --- |
| source | number | Placement ID where data originates | 4 (from placement #4) |
| wireIndex | number? | Output wire index from source | 0 (first output) |
| value | bigint | Actual data value | 15n |
| valueHex | string | Hex representation | "0x0f" |
| sourceSize | number | Data size in bytes | 32 |
| extSource | string? | External source address | "0x123...abc" |
| type | string? | External data type | "Calldata", "Storage" |
| key | string? | Storage key (if applicable) | "0x00...01" |
| offset | number? | Memory offset (if applicable) | 64 |

## StackPt (Symbol Stack)

### Concept

[StackPt](synthesizer-terminology.md#stackpt) is the **symbolic equivalent** of the EVM stack. While the EVM stack holds actual values for computation, StackPt holds DataPt symbols for tracking data flow:

EVM Stack StackPt (Symbol Stack)  
───────────────── ──────────────────────────────  
  
┌──────────────┐ ┌───────────────────────────┐  
│ 30 │ ← value │ DataPt { value: 30n, │  
├──────────────┤ │ source: 4, │  
│ 20 │ │ wireIndex: 0 } │  
├──────────────┤ ├───────────────────────────┤  
│ 10 │ │ DataPt { value: 20n, │  
└──────────────┘ │ source: 0, │  
 │ wireIndex: 1 } │  
 Stores values ├───────────────────────────┤  
 for execution │ DataPt { value: 10n, │  
 │ source: 0, │  
 │ wireIndex: 0 } │  
 └───────────────────────────┘  
  
 Stores symbols  
 for circuit generation

### Visual: Parallel Processing

Opcode: 0x01 ADD  
  
┌─────────────────────────────┐ ┌──────────────────────────────────┐  
│ EVM Stack (functions.ts) │ │ StackPt (handlers.ts) │  
├─────────────────────────────┤ ├──────────────────────────────────┤  
│ │ │ │  
│ pop: [10, 20] │ │ pop: [DataPt{10}, DataPt{20}] │  
│ ↓ │ │ ↓ │  
│ compute: 10 + 20 = 30 │ │ place: ALU1(selector, 10, 20) │  
│ ↓ │ │ ↓ │  
│ push: 30 │ │ push: DataPt{30, source:4} │  
│ │ │ │  
│ Result: Value only │ │ Result: Value + Traceability │  
└─────────────────────────────┘ └──────────────────────────────────┘

### Comparison

| Aspect | EVM Stack | StackPt |
| --- | --- | --- |
| **Data Type** | bigint[] (values) | DataPt[] (symbols) |
| **Purpose** | Execute operations | Track data flow |
| **Push** | Pushes actual value | Pushes symbol reference |
| **Pop** | Returns value for computation | Returns symbol for circuit generation |
| **Operation** | Performs arithmetic | Records circuit connections |

### Definition

**Source**: [pointers/stackPt.ts:1-56](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/pointers/stackPt.ts#L1-L56)

export type TStackPt = DataPt[];  
  
export class StackPt {  
 private \_stack: TStackPt;  
 private \_len: number;  
 private \_maxHeight: number;  
  
 constructor(maxHeight = 1024) {  
 this.\_stack = [];  
 this.\_len = 0;  
 this.\_maxHeight = maxHeight;  
 }  
  
 public push(dataPt: DataPt): void;  
 public pop(): DataPt;  
 public popN(num: number): DataPt[];  
 public swap(position: number): void;  
 public dup(position: number): void;  
 // ... more methods  
}

### Key Differences from EVM Stack

**Source Comment**: [pointers/stackPt.ts:6-26](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/pointers/stackPt.ts#L6-L26)

/\*\*  
 \* Key differences between Stack and StackPt classes  
 \*  
 \* 1. Data Type  
 \* - Stack: bigint[] (stores actual values)  
 \* - StackPt: DataPt[] (stores data pointers)  
 \*  
 \* 2. Purpose  
 \* - Stack: Used in actual EVM execution  
 \* - StackPt: Used for symbolic execution  
 \*  
 \* 3. Operation Handling  
 \* - Stack: Performs operations on actual values (e.g., actual addition)  
 \* - StackPt: Manages pointers for data flow tracking  
 \*  
 \* 4. Usage  
 \* - Stack: Actual transaction processing, contract execution  
 \* - StackPt: Program analysis, optimization, circuit generation  
 \*/

### Example: Parallel Stack Processing

// EVM Stack (functions.ts) - Opcode 0x01: ADD  
const [a, b] = runState.stack.popN(2);  
const result = (a + b) % TWO\_POW256;  
runState.stack.push(result);  
// Stack: [10, 20] → [30]  
  
// StackPt (handlers.ts) - Opcode 0x01: ADD  
const [aPt, bPt] = runState.stackPt.popN(2);  
const resultPt = synthesizer.placeArith('ADD', [aPt, bPt]);  
runState.stackPt.push(resultPt);  
// StackPt: [DataPt{10}, DataPt{20}] → [DataPt{30, source:4}]

### Memory Management Optimization

/\*\*  
 \* 2. Memory Management  
 \* - Once allocated, array size never decreases  
 \* - During pop operations, \_len is decreased instead of actually deleting items  
 \* - This is an optimization strategy to reduce memory reallocation costs  
 \*/  
  
// Implementation  
public pop(): DataPt {  
 if (this.\_len < 1) {  
 throw new EvmError(ERROR.STACK\_UNDERFLOW);  
 }  
 const dataPt = this.\_stack[this.\_len - 1];  
 this.\_len--; // Decrease length, don't delete item  
 return dataPt;  
}

## MemoryPt (Symbol Memory)

### Concept

MemoryPt is a **2D data structure** (offset × time) that solves the [**data aliasing problem**](synthesizer-terminology.md#data-aliasing) in memory. Unlike EVM memory which only keeps the latest value, MemoryPt tracks all overlapping writes:

EVM Memory (1D) MemoryPt (2D: offset × time)  
─────────────── ────────────────────────────  
  
Memory Array: Timestamp Map:  
┌────┬────┬────┬────┐  
│ 00 │ 10 │ 20 │ 30 │ ← offset timestamp=0: { offset: 0x00, size: 32, dataPt1 }  
└────┴────┴────┴────┘ timestamp=1: { offset: 0x10, size: 32, dataPt2 }  
 │ timestamp=2: { offset: 0x08, size: 16, dataPt3 }  
 └─> Only latest value  
 (old values lost) ↑  
 All writes preserved with timestamps!

### Visual: The Data Aliasing Problem

Time ─────────────────────────────────────────>  
  
t=0: MSTORE(0x00, dataPt1) Write 32 bytes at 0x00-0x20  
 ┌────────────────────────────────┐  
 │ dataPt1 (value=100) │  
 └────────────────────────────────┘  
 0x00 0x20  
  
t=1: MSTORE(0x10, dataPt2) Write 32 bytes at 0x10-0x30 (overlaps!)  
 ┌────────────────────────────────┐  
 │ dataPt2 (value=200) │  
 └────────────────────────────────┘  
 0x10 0x30  
  
t=2: MLOAD(0x00, 32) Load 32 bytes at 0x00-0x20  
  
 ┌──────────────┬──────────────┐  
 │ dataPt1 │ dataPt2 │ ← Need BOTH!  
 │ (bytes 0-15)│ (bytes 0-15)│  
 └──────────────┴──────────────┘  
 0x00 0x10 0x20  
  
EVM Result: Only sees final state (loses history)  
MemoryPt: Returns DataAliasInfos for circuit generation

### How MemoryPt Solves This

| Aspect | EVM Memory | MemoryPt |
| --- | --- | --- |
| **Data Structure** | Uint8Array (continuous bytes) | Map<timestamp, MemoryPtEntry> |
| **Write** | Overwrites bytes directly | Records timestamped data points |
| **Read** | Returns current byte values | Returns data alias information |
| **Aliasing** | Lost (only latest value) | Tracked (all overlapping writes) |

### Definition

**Source**: [pointers/memoryPt.ts:163-436](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/pointers/memoryPt.ts#L163-L436)

export type TMemoryPt = Map<number, MemoryPtEntry>;  
  
export interface MemoryPtEntry {  
 memOffset: number; // Memory start offset  
 containerSize: number; // Size of data in bytes  
 dataPt: DataPt; // Data point reference  
}  
  
export class MemoryPt {  
 \_storePt: TMemoryPt;  
 private \_timeStamp: number;  
  
 constructor() {  
 this.\_storePt = new Map();  
 this.\_timeStamp = 0;  
 }  
  
 public write(offset: number, size: number, dataPt: DataPt): void;  
 public getDataAlias(offset: number, size: number): DataAliasInfos;  
 // ... more methods  
}

### Implementation Details

**Source Comment**: [pointers/memoryPt.ts:135-161](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/pointers/memoryPt.ts#L135-L161)

/\*\*  
 \* 3. Read/Write Operations  
 \* - Memory: Direct read/write to actual memory  
 \* - MemoryPt:  
 \* - Write: Creates new data pointers and manages overlapping regions  
 \* - Read: Returns data alias information through getDataAlias  
 \*  
 \* 5. Characteristics  
 \* - MemoryPt:  
 \* - Timestamp-based data management  
 \* - Memory region conflict detection  
 \* - Data alias information generation  
 \*/

### Example: Overlapping Memory Writes

// Scenario: Overlapping MSTORE operations  
1. MSTORE(0x00, dataPt1) // Write 32 bytes at 0x00-0x20, timestamp=0  
2. MSTORE(0x10, dataPt2) // Write 32 bytes at 0x10-0x30, timestamp=1  
3. MLOAD(0x00) // Load 32 bytes at 0x00-0x20  
  
// MemoryPt tracks both writes:  
\_storePt = {  
 0: { memOffset: 0x00, containerSize: 32, dataPt: dataPt1 },  
 1: { memOffset: 0x10, containerSize: 32, dataPt: dataPt2 }  
}  
  
// MLOAD(0x00) generates DataAliasInfos:  
[  
 {  
 dataPt: dataPt1,  
 shift: 0, // Use bytes 0-15 from dataPt1  
 masker: 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000000000000000000000000000  
 },  
 {  
 dataPt: dataPt2,  
 shift: 128, // Use bytes 0-15 from dataPt2 (shifted left by 128 bits)  
 masker: 0x00000000000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
 }  
]  
  
// Circuit generation:  
// result = (dataPt1 & mask1) | ((dataPt2 >> 128) & mask2)

### Data Alias Information

export interface DataAliasInfo {  
 dataPt: DataPt; // Source data point  
 shift: number; // Bit shift for alignment  
 masker: bigint; // Mask for extracting relevant bits  
}  
  
export type DataAliasInfos = DataAliasInfo[];

The getDataAlias() method returns information needed to reconstruct memory data from overlapping writes, which is then used to generate circuits (using DecToBit, Accumulator, and bitwise operations).

## Placement

### Concept

A **Placement** is an **instance** of a [subcircuit](synthesizer-terminology.md#subcircuit) with specific input/output data. Think of it like an object created from a class:

Subcircuit (Template/Class) Placement (Instance/Object)  
─────────────────────────── ──────────────────────────  
  
┌─────────────────────┐ ┌────────────────────────┐  
│ ALU1.circom │ ─────> │ Placement ID: 4 │  
│ │ │ name: 'ALU1' │  
│ - Defines circuit │ │ usage: 'ADD' │  
│ - Has 803 │ │ inPts: [sel, a, b] │  
│ constraints │ │ outPts: [result] │  
│ - Can do ADD, MUL, │ └────────────────────────┘  
│ SUB, etc. │  
└─────────────────────┘ ┌────────────────────────┐  
 ─────> │ Placement ID: 5 │  
 │ name: 'ALU1' │  
 │ usage: 'MUL' │  
 │ inPts: [sel, x, y] │  
 │ outPts: [product] │  
 └────────────────────────┘  
  
One template, multiple instances with different data!

### Visual: Placement Creation Flow

Opcode Execution Placement Creation Circuit Graph  
──────────────── ────────────────── ─────────────  
  
ADD instruction ─> 1. Select subcircuit ─> Placement #4  
 (ALU1, selector=2) ┌──────────┐  
Pop stack: │ ALU1 │  
 a = DataPt{10} ─> 2. Gather inputs ─> │ ADD │  
 b = DataPt{20} [sel, aPt, bPt] │ │  
 │ in: [2, │  
Compute: ─> 3. Create output ─> │ 10, 20]│  
 result = 30 DataPt{30, source:4} │ out: [30]│  
 └──────────┘  
Push result ─> 4. Store placement │  
 placements.set(4, ...) └─> Used by  
 next op

### Purpose

* **Template vs Instance**: Subcircuit is a template (e.g., ALU1.circom), Placement is an instance
* **ID Assignment**: Each placement gets a unique sequential ID (starting from 4)
* **Wire Connections**: outPts from one placement become inPts to another
* **Circuit Building**: Placements form a Directed Acyclic Graph (DAG) representing the entire computation

### Definition

**Source**: [types/synthesizer.ts:71-76](https://github.com/tokamak-network/Tokamak-zk-EVM/blob/main/packages/frontend/synthesizer/src/tokamak/types/synthesizer.ts#L71-L76)

export type PlacementEntry = {  
 name: SubcircuitNames; // Type of subcircuit (e.g., 'ALU1', 'AND')  
 usage: ArithmeticOperator; // Specific operation (e.g., 'ADD', 'MUL')  
 subcircuitId: SubcircuitId; // Numeric ID from QAP Compiler  
 inPts: DataPt[]; // Input data points  
 outPts: DataPt[]; // Output data points  
};  
  
export type Placements = Map<number, PlacementEntry>;

### Example: Placement Creation

// When synthesizerArith('ADD', [aPt, bPt], resultValue, runState) is called:  
  
// 1. Select subcircuit and create selector  
const [subcircuitName, selectorValue] = SUBCIRCUIT\_MAPPING['ADD']; // ['ALU1', 2n]  
const selectorPt = synthesizer.loadAuxin(selectorValue);  
  
// 2. Create output DataPt  
const resultPt = DataPointFactory.create({  
 source: placementId, // ID of this placement (e.g., 4)  
 wireIndex: 0, // First output wire  
 value: resultValue, // Actual result (15n)  
 sourceSize: 32  
});  
  
// 3. Create and store placement  
const placement: PlacementEntry = {  
 name: 'ALU1',  
 usage: 'ADD',  
 subcircuitId: 4,  
 inPts: [selectorPt, aPt, bPt],  
 outPts: [resultPt]  
};  
  
synthesizer.state.placements.set(placementId, placement);

### Placement IDs 0-3: Buffer Placements

Special placements act as **interfaces** between the external world and the circuit:

Buffer Placements: The Bridge Between Worlds  
────────────────────────────────────────────  
  
External World Buffer Layer Circuit World  
────────────── ──────────── ─────────────  
  
Calldata ┐  
Block info ├──> PUB\_IN (ID=0) ──> Public symbols ──┐  
msg.sender ┘ │  
 ├──> Circuit  
Storage ┐ │ operations  
Account state ├──> PRV\_IN (ID=2) ──> Private symbols ──┘  
Private data ┘  
 ┌──> Return data  
 │ Logs  
Circuit result ──> PUB\_OUT (ID=1) ──> Public output ────┘  
  
Circuit result ──> PRV\_OUT (ID=3) ──> Storage updates  
 State changes

| ID | Name | Purpose |
| --- | --- | --- |
| 0 | PUB\_IN | Public input buffer (calldata, block info, msg.sender) |
| 1 | PUB\_OUT | Public output buffer (return data, logs) |
| 2 | PRV\_IN | Private input buffer (storage, account state) |
| 3 | PRV\_OUT | Private output buffer (storage updates) |

### Visualizing Complete Data Flow

External World Synthesizer Circuit External World  
────────────── ─────────────────── ──────────────  
  
calldata[0]=5 ──┐  
block.number ──┤──> PUB\_IN (ID=0) ──> DataPt{5, source:0, wire:0} ──┐  
msg.sender ──┘ │  
 ▼  
storage[key]=10 ──┐ ADD (ID=4)  
 ├──> PRV\_IN (ID=2) ──> DataPt{10, source:2, wire:0} ──┤  
account.balance ──┘ │  
 ▼  
 DataPt{15, source:4, wire:0}  
 │  
returnData <───────── PUB\_OUT (ID=1) <───────────────────────────────────┤  
logs <──┘ │  
 │  
storage[key]=15 <───── PRV\_OUT (ID=3) <────────────────────────────────────┘