# Synthesizer: Concepts

### What is Synthesizer?

Synthesizer is a compiler that processes an Ethereum transaction and returns a *wire map.* This wire map serves as preprocessed input for [Tokamak zk-SNARK](https://eprint.iacr.org/2024/507).

|  |
| --- |
| Input and output of Synthesizer |

Input and output of Synthesizer

Input and output of Synthesizer

The prover and verifier of Tokamak zk-SNARK require a zk circuit to initiate an argument. In general, zk circuits translate general programs into a ZKP-compatible language. In our context, Synthesizer translates Ethereum transactions into transaction-specific zk circuits (referred to simply as *circuits*).

Suppose [a predefined library of subcircuits](https://github.com/tokamak-network/Tokamak-zk-EVM/tree/main/packages/frontend/qap-compiler/subcircuits/library) is provided. A circuit is then constructed by placing copies of these subcircuits and connecting their input and output wires. The wire map generated by Synthesizer is in the form of a permutation map, describing the wire connections between these placements. A more rigorous definition of Synthesizer is provided in [the paper](https://eprint.iacr.org/2024/507).

|  |
| --- |
| Figure from the paper |

Figure from the paper

Figure from the paper

|  |
| --- |
| Figure from the paper |

Figure from the paper

Figure from the paper

We anticipate that the subcircuits in our prepared library are sufficient to represent all signal processing performed within the EVM, with the exception of the Keccak function. As a result, the resulting circuit—constructed as a combination of these subcircuits—is expected to handle all kinds of Ethereum transactions.

If we encounter an unexpected transaction that cannot be represented using our current library, new subcircuits can be added as needed by us or external contributors. For instance, this was the case with the EXP instruction, which will be explained later.

However, we have chosen not to implement a Keccak circuit due to efficiency concerns (see the section “[Arithmetic, comparison, and bitwise logic operations](https://www.notion.so/Synthesizer-documentation-164d96a400a3808db0f0f636e20fca24?pvs=21)” for further details).

### Where can I get the input?

* Ethereum transactions

You can retrieve Ethereum transactions from [Etherscan.io](http://etherscan.io). Simply copy a transaction’s hash and provide it to Synthesizer.

* Subcircuit library

We provide the [qap-compiler](https://github.com/tokamak-network/Tokamak-zk-EVM/tree/main/packages/frontend/qap-compiler) that manages a library of fundamental subcircuits. These subcircuits support EVM arithmetic and logical operations for 256-bit words. The subcircuits can be compiled into the R1CS format using [Circom](https://docs.circom.io/), making them compatible with Synthesizer. You don’t need to be familiar with Circom. Simply ensure that Circom is installed on your system and run the following shell script (be sure that you are in the folder “Tokamak-zk-EVM/packages/frontend/qap-compiler”):

./scripts/compile.sh

This script generates the files subcircuit0.wasm through subcircuitN.wasm, as well as globalWireList.ts, and subcircuitInfo.ts. These files are then used as input for the Synthesizer and the [bakcend algorithms](https://github.com/tokamak-network/Tokamak-zk-EVM/tree/main/packages/backend).

### Where does the output go?

Synthesizer behaves as a front-end compiler for Tokamak zk-EVM. The diagrams below illustrate the overall signal flow of Tokamak zk-SNARK:

Front-end algorithms for Tokamak zk-SNARK

The back-end protocol of Tokamak zk-SNARK

## Exploring Synthesizer’s core

Synthesizer works like an add-on to the EVM. In other words, Synthesizer performs additional tasks while still fulfilling the role of the EVM.

### Difference in signal processing from EVM

The difference in signal processing between the original EVM and Synthesizer are illustrated in the figures below.

Signal processing in EVM

In this figure, the EVM takes input values such as environment information, hardcoded data from a bytecode, block information, and data loaded from storage. It then returns output values, including data to be stored in storage, data for logs, and the amount of used gas. In EVM, the relationship between the inputs and outputs is not directly observable.

Signal processing in Synthesizer

As illustrated in this figure, the EVM with Synthesizer treats inputs as *symbols* instead of direct values. Synthesizer produces an intermediate output called Placements, which lists subcircuit placements that manipulates the input symbols. These placements allow the output symbols to be *evaluated* as EVM outputs. In simpler terms, Placements establish a traceable relationship between EVM outputs and inputs by composing subcircuits from the library. The remaining task for Synthesizer is to convert Placements into a permutation map.

Although Synthesizer treats inputs as symbols, it is important to note that this approach is unrelated to [symbolic execution](https://en.wikipedia.org/wiki/Symbolic_execution). When Synthesizer encounters a computation fork during transaction execution, it strictly follows the branch selected by the original EVM, disregarding other possibilities. As zk circuits generally exhibit passive and deterministic properties, the subcircuit composition in Synthesizer follows these principles.

As the EVM processes each instruction, Synthesizer performs additional work. Below, we break down how Synthesizer handles each group of [EVM instructions](https://www.evm.codes/).

### Conversion between symbols and evaluations

When Synthesizer starts, Placements is initialized with buffer subcircuit placements. Each buffer subcircuit can take multiple inputs and pass them to outputs. Buffers serve as interfaces for circuits to receive or output external values. They are also used as converters to translate EVM input values into symbols or symbols into EVM output values.

For instance, consider an EVM transaction that retrieves two input values—one from calldata and another from storage—and stores their product in storage. As the result of Synthesizer, thePlacements would look like the figure below:

In this figure, LOAD and RETURN represent the buffer subcircuit placements to produce and evaluate the sybols

,

,

respectively. The other placement, MUL, copied from the library subcircuit for the product, uses these symbols as its inputs and outputs. Here we provide an (simplified) example of the resulting Placements:

Placements = {  
 "0": {  
 "name": "LOAD",  
 "inPts": [  
 {  
 "source": "code: ${address}",  
 "type": "Calldata(User)",  
 "offset": 0,  
 "valueHex": "0x02"  
 },  
 {  
 "source": "storage: ${address}",  
 "key": "0x00",  
 "valueHex": "0x0a"  
 }  
 ],  
 "outPts": [  
 {  
 "source": "placement: 0",  
 "wireIndex": 0,  
 "valueHex": "0x02"  
 },  
 {  
 "source": "placement: 0",  
 "wireIndex": 1,  
 "valueHex": "0x0a"  
 }  
 ]  
 },  
 "1": {  
 "name": "RETURN",  
 "inPts": [  
 {  
 "source": "placement: 2",  
 "wireIndex": 0,  
 "valueHex": "0x14"  
 }  
 ],  
 "outPts": [  
 {  
 "dest": "storage: ${address}",  
 "key": "1",  
 "valueHex": "0x14"  
 }  
 ]  
 },  
 "2": {  
 "name": "MUL",  
 "inPts": [  
 {  
 "source": "placement: 0",  
 "wireIndex": 1,  
 "valueHex": "0x0a"  
 },  
 {  
 "source": "placement: 0",  
 "wireIndex": 0,  
 "valueHex": "0x02"  
 }  
 ],  
 "outPts": [  
 {  
 "source": "placement: 2",  
 "wireIndex": 0,  
 "valueHex": "0x14"  
 }  
 ]  
 }  
}

In Synthesizer, all library’s subcircuits other than buffers are strictly allowed to handle only symbols as both inputs and outputs. In other words, buffer subcircuits are the only placements that can directly take non-symbol values as inputs or outputs. Additionally, there can be only one buffer placement of a given purpose within Placements. For example, in the scenario above, there can be at most one LOAD and one RETURN in Placements.

### Stack, memory, storage, and flow operations

Similar to how the EVM manages the stack and memory, Synthesizer uses its own stack and memory structures, named StackPt and MemoryPt. Unlike the EVM’s stack and memory, which store values, StackPt and MemoryPt store symbols. These structures are synchronized with the EVM’s stack and memory. For instance, whenever an item is added to or removed from the stack, the same operation occurs in StackPt. The same applies to the relationship between MemoryPt and memory. The evaluation of each symbol in StackPt or MemoryPt corresponds to the value in the EVM’s stack or memory.

While both StackPt and the stack share a stack-like structure, MemoryPt and memory differ significantly. Memory is a one-dimensional structure, whereas MemoryPt is two-dimensional. When writing new data, the memory only considers the space where the data is written. In contrast, MemoryPt tracks both the space and time of the data write. This structural feature of MemoryPt allows Synthesizer to detect and track any manipulation of the symbols that occur during memory reads.

For example, suppose that there was data

of length 0x20 at memory offset 0x10, and new data

of length 0x20 is written at location 0x00. In the EVM, reading the memory region from 0x10 to 0x20 would lose information about

. However, MemoryPt allows us to track how

and

are manipulated and related to the read data, as illustrated below.

Data aliasing in the memory

Data aliasing is solvable with MemoryPt

#### Key instructions

* PUSHs: In the EVM, the PUSH instructions add hardcoded values from a bytecode to the stack. In Synthesizer, these value are symbolized (e.g., as
* through the LOAD buffer interface. The symbol is then added to the top of StackPt. The wires of LOAD used for the symbolization are recorded in Placements.
* MSTORE8: In the EVM, the MSTORE8 instruction writes only the last 8 bits of a stack item to the memory. In Synthesizer, this process is emulated by masking the symbol with a library subcircuit for the AND operation. The resulting symbol is added to MemoryPt. The LOAD wires for symbolizing a masker value and the new AND placement are recorded in Placements.
* MSTORE: In the EVM, the MSTORE instruction writes a stack item to memory. In Synthesizer, the MSTORE instruction adds a StackPt symbol to MemoryPt. No additional subcircuit placement is required.
* MSTORE8: In the EVM, the MSTORE8 instruction writes only the last 8 bits of a stack item to the memory. In Synthesizer, this process is emulated by masking the symbol with a library subcircuit for the AND operation. The resulting symbol is added to MemoryPt. The LOAD wires for symbolizing a masker value and the new AND placement are recorded in Placements.
* MLOAD: In the EVM, the MLOAD instruction reads a specific memory region and adds the result to the stack. In Synthesizer, any possible aliasing of overwritten symbols in the memory region is reproduced using subcircuits (e.g., the library’s subcircuits for ADD, SHL, SHR, AND operations). The reproduced symbol is finally added to StackPt. All involved LOAD wires and new placements are recorded in Placements.
* SSTORE: In the EVM, the SSTORE instruction moves a stack item to storage. We can regard this item as one of the actual EVM outputs. In Synthesizer, a StackPt symbol is fed to the interface buffer, RETURN, so that we can desymbolize or evaluate it. The wires of RETURN involved in the evaluation are recorded in Placements.
* SLOAD: In the EVM, the SLOAD instruction retrieves a storage value and adds it to the stack. In Synthesizer, this value is symbolized through LOAD and added to StackPt. The LOAD for the symbolization are recorded in Placements.
* MCOPY: In the EVM, the MCOPY instruction copies a memory region of flexible size to another offset. In Synthesizer, truncation and shifting of symbols are reproduced before adding them to MemoryPt. All involved LOAD wires and new placements are recorded in Placements.
* Reproduction of the newly added symbols
* JUMP/JUMPI: Synthesizer performs no additional work beyond managing StackPt.

### Arithmetic, comparison, and bitwise logic operations

When the EVM interpreter executes instructions for arithmetic, comparison, or bitwise logic operations, it pops one to three operands from the stack, applies the operation, and then pushes the resulting value back onto the stack. Synthesizer follows the same process using StackPt. Additionally, Synthesizer records the relationship between input and output symbols in Placements.

|  |
| --- |
| Signal processing in Synthesizer for arithmetic, comparison, and bitwise logic operations (e.g., ADD instruction). |

Signal processing in Synthesizer for arithmetic, comparison, and bitwise logic operations (e.g., ADD instruction).

Signal processing in Synthesizer for arithmetic, comparison, and bitwise logic operations (e.g., ADD instruction).

The placements that track symbol relationships resulting from these instructions are often directly represented using built-in subcircuits from the subcircuit library. However, for certain operations, such placements must be derived indirectly by combining multiple built-in subcircuits. Currently, Synthesizer derives placements for the EXP and KECCAK256 instructions indirectly. In the future, more instructions may be considered to adopt indirect placement derivation to further improve the efficiency of Tokamak zk-SNARK.

#### Instructions that need indirect placements

* EXP: Since the number of multiplications for a single exponentiation can only be determined when the input value is given, it is challenging to implement it as an optimized circuit. The output symbol manipulated by the EXP instruction can be reproduced using the placements shown in the figure below. This process utilizes two library subcircuits, Bitify and SubEXP.
  + Bitify: It has one input
  + , and
  + outputs
  + , such that
  + .
  + SubEXP: It has three inputs
  + , and two outputs
  + , such that
  + ,
  + , and
  + .
* Reproduction of the newly added symbols
* KECCAK256: Implementing Keccak hashing directly in a circuit is computationally inefficient. Both direct subcircuit implementation, such as [Keccak256-circom](https://github.com/vocdoni/keccak256-circom), and indirect derivation of placements would result in complexities of approximately 151k constraints. This is significantly higher compared to the complexity of thousands for the library subcircuits. Therefore, adopting a KECCAK256 subcircuit would severely impact the performance of the subsequent proving algorithm of the Tokamak zk-SNARK.
* As a result, Synthesizer does not reproduce the output symbols of KECCAK256. Instead, it uses buffer subcircuits to emit the KECCAK256 input values from the circuit and reintroduce the KECCAK256 output values back into the circuit. Outside the circuit, the Keccak hash computation can be run by the verifier of the Tokamak-zk SNARK.
* In the EVM, the KECCAK256 instruction first loads a value from a specific memory region of variable length. This length can exceed the EVM’s word size of 32 bytes, and the value is then passed to the Keccak hash function. In Synthesizer, reproducing aliasing of symbols that during memory loading is handled similarly to the MLOAD instruction, with adjustments to account for the dynamic length of the memory region:
  + If the length of the memory region does not exceed 32 bytes, aliasing is reproduced in the same manner as described for MLOAD.
  + If the length of the memory region exceeds 32 bytes, Synthesizer divides the memory region into multiple 32-byte chunks and reproduces aliasing for each chunk individually.

### System operations

Instructions in the system operations group manages EVM [contexts](https://www.evm.codes/about). These instructions enable the EVM to either enter a new child context or return to the existing parent context. Each context operates independently, in the perspective of managing its own stack and memory. Transferring memory data between parent and child contexts is exclusively handled by instructions in this group.

However, one context—whether parent or child—cannot directly manipulate the stack or memory of another context. When system operation instructions such as CALL or RETURN are executed, the EVM creates a virtual memory space (a buffer) to mediate these interactions. Each context can only access its own buffer. When a parent context initiates a child context, or when a child context terminates, a specified region of the memory from the current context is copied to the buffer. This buffer is then delivered as environmental information to the receiving context (child or parent).

In Synthesizer, StackPt and MemoryPt are maintained for each context, mirroring the behavior of the original EVM stack and memory. A virtual memory structure, similar to MemoryPt, is also implemented to facilitate memory data transfers between contexts. This identical structure enables Synthesizer to reproduce data aliasing that during memory transfers in the same way as with the MCOPY instruction.

To track symbol manipulations across all contexts, Placements is shared and jointly managed.

#### Key instructions

* CALL, CALLCODE, DELEGATECALL, STATICCALL (in a parent context): In the EVM, these instructions first copy a specific memory region from the parent context into a buffer. This buffer is then passed to the child context as environment information.
* In the Synthesizer, the data transfer from the memory to the buffer is handled in the same way as with the MCOPY instruction.
* CALLDATALOAD, CALLDATACOPY (in a child context): In the child context, these instructions retrieve the buffer content. Any data aliasing during this process is reproduced in the same way as with the MLOAD or MCOPY instructions.
* Any instruction execution (in a child context): All symbol manipulations are recorded in Placements, regardless of the context in which the EVM interpreter operators.
* RETURN (in a child context): In the EVM, this instruction transfers a specified memory region from the child context back to the parent context’s memory: The child context’s memory region of interest is first copied into a buffer, and when control returns to the parent context, the buffer content is written to a predefined memory region in the parent context.
* In Synthesizer, data aliasing during this composite transfer (from memory to buffer and then back to memory) is reproduced in the same way as with the MCOPY instruction.

### Environmental and block information

The EVM uses instructions from the environmental and block information group to load an account’s state, the EVM interpreter’s state, and block information onto the stack. Most of the data loaded by these instructions can be treated as external inputs to the EVM, similar to storage data.

In Synthesizer, loading the external inputs to the EVM is handled in the same way as loading storage data. Specifically, when instructions in this group are executed, Synthesizer converts the external input values into symbols using the LOAD placement. For more details, refer to the explanation of SLOAD in the section “Stack, memory, storage, and flow operations”. Signal processing for each individual instruction in this group is omitted in this document.

#### Special cases: Child contexts

There are exceptions to this process, specifically for instructions like CALLDATALOAD, CALLDATACOPY, and RETURNCOPY when executed in child contexts:

* Main context: When these instructions are executed in the main context, the data loaded is treated as external inputs to the EVM, following the standard process similar to the case of SLOAD as described above.
* Child contexts: When executed in a child context, these instructions are processed as described in the section “System operations”, where they handle data transfer between contexts via buffers and produce aliasing if necessary.

### Log instructions

The EVM uses Log instructions to emit meaningful data manipulated during execution to external systems. This emitted data typically consists of memory data from a specific region, with optional stack data included as “topics”. The data sent by Log instructions is treated as part of the EVM’s output, similar to storage data.

In Synthesizer, emitting the EVM output is handled similarly to emitting storage data. Specifically, when a Log instruction is executed, Synthesizer uses the RETURN placement to convert the symbols in StackPt or MemoryPt into values.

However, since some of these symbols are retrieved from specific memory regions, aliasing may occur during this process. In such cases, Synthesizer reproduces the aliasing using the same manner applied for the MLOAD instruction and records the process in Placements.

## Features not yet implemented

The following features or instructions are not currently supported in the existing version of Synthesizer. They will be implemented in the future.

### Batch transaction execution

Currently, Synthesizer processes single transactions. Support for batch transaction execution will be added in future versions. This feature is essential for Layer 2 scaling solutions, particularly for state channel implementations where multiple off-chain transactions need to be batched and proven together when opening or closing channels.