# **Assembly**

**Benefits of using inline assembly:**

* **Reduced gas cost.**

Solidity’s assembly generates a lot of redundant opcodes and odd jumps.

Using inline assembly can optimize the code significantly.

The compiled binaries can be quite smaller than their Solidity equivalent.

* **Processing entire words at once.**

Inline assembly allows to read entire words of 32 bytes in a single operation.

String comparisons for example can become very fast because you can compare them in chunks of 32 bytes instead of byte by byte.

(This wasn’t necessary for this assignment.)

* **More exposure of operations.**

Some operations aren’t exposed in Solidity. For instance, the keccak opcode takes a byte range in memory to hash, while the Solidity function takes a string.

Hashing parts of a string in Solidity would require costly string copying operations.

With inline assembly, only the needed bytes can be passed to hash.

* **Faster**

Because the opcodes are optimized and entire words can be processed at once, the execution of a function will be faster than equivalent Solidity code.

* **Educational**

You really need to think about how Solidity works internally.

The layout of storage variables and their types are important.

EVM uses more operations to store elements smaller than 32 bytes.

To optimize this, the order of storage variables needs to be such that they can be packed tightly. For example: uint128, uint128, uint256 needs two slots of storage while uint128, uint256, uint128 needs 3.

**Inline assembly in this assignment:**

We used inline assembly for 3 functions:

* ***getAmountOfDocumentsFromAuthor***
  + Input: bytes32 authorName
  + Output: uint256 nrOfDocuments

The use of assembly isn’t really necessary here.

It was done for demonstrational purposes.

* ***registerDocumentAsm***
  + Input: bytes32 docHash, bytes32 title, bytes32 releaseDate, bytes32 authorName, bytes32 authorMail
  + Output: bool succes

Inline assembly would mostly benefit here.

The use of storage variables are costly. For this assignment it was needed to use 2 structs and 2 mappings. Inline assembly allowed to access those storage variables in a more efficient manner.

We didn’t really see a significant reduced gas cost. This is probably because of the CMC setup.

* ***getDocumentListFromAuthorAsm***
  + Input: bytes32 authorName, uint256 startIndex, uint256 endIndex
  + Output: bytes32 authorMail, bytes32[pageSize] memory titles, bytes32[pageSize] memory releaseDates

Because this function is constant, gas reduction isn’t needed.

Inline assembly does execute this function faster.

Note: Dynamic arrays can’t be passed between contracts. So fixed sized arrays are used.

**Inline assembly in this assignment – explanation:**

We have following storage variables:

struct *Author* { bytes32 name; bytes32 mail; }

struct *Document* { bytes32 title; bytes32 releaseDate; Author author; bool isEntity; }

Mappings:

*documentsOfAuthor[authorName]* = array of docHashes

*documents[docHash]* = Document

To get the location of documentsOfAuthor[key] in storage:

* Get first free memory slot with mload(0x40).
* Store the key into that location, which is 32 bytes.
* Store the slot of storage it uses in the next 32 bytes.
* Keccak (SHA3) those 64 bytes. Output is 32 bytes.
* The length of the dynamic array is stored at that location.

To get element documentsOfAuthor[authorName][i]:

* For element 0: Keccak the first 32 bytes of previous output.
* Add i tot he new output of 32 bytes and load value stored at that location

It’s the same principle for documents[docHash]:

* Store 32 byte docHash in memory.
* Store slot of storage in next 32 bytes.
* Keccak those 64 bytes.
* That location contains the struct. To access a variable in the struct, just add its position to the calculated output.