What is different

It is worth emphasizing that there are slight differences in the behaviour of Titan compared to Ethereum. When building applications on Titan, it is important to be aware of these discrepancies and consider them during development.

Opcode Differences

Modified Opcodes

Opcode	Solidity equivalent	Behaviour
`COINBASE`	`block.coinbase`	Value is set by the sequencer. Currently returns the OVM_SequencerFeeVault address (0x420...011).
`DIFFICULTY`	`block.difficulty`	Always returns zero.
`BASEFEE`	`block.basefee`	Currently unsupported.
`ORIGIN`	`tx.origin`	If the transaction is an L1 ⇒ L2 transaction, then tx.origin is set to the https://www.notion.so/ENG-75342b0432fc478dac546787fa49c2cf?pvs=21 of the address that triggered the L1 ⇒ L2 transaction. Otherwise, this opcode behaves normally.

Added Opcodes

Opcode	Behavior
`L1BLOCKNUMBER`	Returns the block number of the last L1 block known by the L2 system. Typically, this block number will lag up to 15 minutes behind the latest L1 block number.

Block Numbers and Timestamps

Block production is not constant

- On Ethereum, the `NUMBER` opcode (`block.number` in Solidity) corresponds to the current Ethereum block number. Similarly, in Titan, `block.number` corresponds to the current L2 block number. However, each transaction on L2 is placed in a separate block and blocks are NOT produced at a constant rate.
- This is important because it means that `block.number` is currently NOT a reliable source of timing information. If you want access to the current time, you should use `block.timestamp` (the `TIMESTAMP` opcode) instead.

Timestamps

- The `TIMESTAMP` opcode (`block.timestamp` in Solidity) uses the timestamp of the transaction itself.

Using ETH in Contracts

- The process of using ETH on L2 is identical to the process of using ETH in Ethereum.

- For tooling developers and infrastructure providers, please note that ETH is still represented internally as an ERC20 token at the address. `0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000`
- As a result, user balances will always be zero inside the state trie, and the user's actual balance will be stored in the aforementioned token's storage.

It is impossible to call this contract directly; it will throw an error.

Address Aliasing

- Because of the behaviour of the `CREATE` opcode, a user can create a contract on L1 and on L2 that share the same address but have different bytecode. This can break trust assumptions because one contract may be trusted and another be untrusted (see below).
- To prevent this problem, the behaviour of the `ORIGIN` and `CALLER` opcodes (`tx.origin` and `msg.sender`) differs slightly between L1 and L2.
- The value of `tx.origin` is determined as follows:

Call source	tx.origin
L2 user (Externally Owned Account)	The user's address (same as in Ethereum)
L1 user (Externally Owned Account)	The user's address (same as in Ethereum)
L1 contract (using CanonicalTransactionChain.enqueue)	L1_contract_address + `0x1111000000000000000000000000000000001111`

- The value of `msg.sender` at the top-level (the very first contract being called) is always equal to `tx.origin` . Therefore, if the value of `tx.origin` is affected by the rules defined above, the top-level value of `msg.sender` will also be impacted.

In general, `tx.origin` should *not* be used for authorization (https://docs.soliditylang.org/en/latest/security-considerations.html#tx-origin). However, that is a separate issue from address aliasing because address aliasing also affects `msg.sender` .