Lecture 08: Ethereum Virtual Machine (EVM)

Akash Khosla
Nick Zoghb

BLOCKCHAIN
AT BERKELEY

LECTURE OUTLINE

1. ETHEREUM TXN RECAP
2. EXECUTION ON THE EVM
3. EVM OVERVIEW
4. EVM 2.0

1

ETHEREUM TXN RECAP

TRANSACTION COMPONENTS

STATE CHANGERS

- **nonce**: number of transactions sent by address of txn sender
- **gasPrice**: amount of Wei sender is willing to pay per unit of gas required to execute the transaction
- **gasLimit**: max amount of gas the sender is willing to pay for executing this transaction, set before any computation is done
- **to**: address of the recipient
- **value**: the amount of Wei to be transferred from the sender to the recipient
- **v, r, s**: used to generate the signature that identifies the sender of the transaction
- **init** (only exists for contract-creating transactions): An EVM code fragment that is used to initialize the new contract account
- **data** (optional field that only exists for message calls): the input data (i.e. parameters) of the message call

GAS AND PAYMENT

FEES

- Every computation that occurs as a result of a transaction on the Ethereum network incurs a fee called **gas**
- **Gas** is the unit used to measure the fees for a particular computation
- **Gas price** is the amount of Ether you are willing to spend on every unit of gas
    - Measured in “gwei” - 1E18 wei = 1 ETH, 1 gwei = 1,000,000,000 wei
- With every transaction, a sender sets a **gas limit** and a **gas price**
    - **gas price** * **gas limit** = max **amount** of wei **sender is willing to pay** for transaction

WHAT IS A TRANSACTION

STATE CHANGERS

- Transactions move the state of an account within the global state - one state to the next
- **Formal Definition:** A transaction is a cryptographically signed piece of instruction that is generated by an externally owned account, serialized, and then submitted to a blockchain
- Two types: **Message calls and contract creations**

MESSAGE CALLS

TRANSACTIONS

- Both **message calls** and **contract creating** transactions are always initiated by **externally owned accounts**
  - Transactions bridge the external world to the internal state of Ethereum
- Contracts that exist within the global scope of Ethereum can talk to other contracts using **messages** (internal transactions) to other contracts
- We can think of messages as being similar to transactions, except they are not generated by externally owned accounts, only by contracts
  - Virtual objects within the Ethereum execution environment

HOW ETHEREUM WORKS

STATE TRANSITION PERSPECTIVE

```
/* Ethereum blockchain update algorithm */
def APPLY(state, TX):
  if not TX.is_valid(): return ERROR # check for well formed txn
  if not TX.check_sig(): return ERROR # check for valid signature
  if TX.sender.nonce != TX.nonce: return ERROR # acct nonce == txn nonce
  fee = txn.gas * txn.gasprice
  decrement(tx.sender.balance, fee) # Returns error if not enough balance
  tx.sender.nonce += 1
  gas = txn.gas - (tx.bytes * miner.gas_per_byte)
  transfer(txn.value, txn.sender, txn.receiver)
"""
transfer: create receiver if non existant. if receiver is contract, run the code to completion or until gas is out.
if failed because not enough money or ran out of gas, revert all state except fees and add the fees to miner
account. If success, refund fees for all remaining gas to the sender and send fees paid for gas consumed to miner.
"""
```

INTRINSIC VALIDITY

TRANSACTIONS

- Initial test before a transaction is executed
  - The transactions follows well-formed Recursive Length Prefix
  - The signature on transaction is valid
  - The nonce on the transaction is valid
    - Same as sender account's current nonce
  - The **gasLimit** is greater than or equal to the **intrinsic gas** used by the transaction
  - The sender's account balance contains the cost required in up-front payment

The **intrinsic gas** for a transaction is the amount of gas that the transaction uses before any code runs. It is a constant "transaction fee" (currently 21000 gas) plus a fee for every byte of data supplied with the transaction (4 gas for a zero byte, 68 gas for non-zeros). These constants are all currently defined for geth in params/protocol_params.go. Presumably they are also coded into the source for the other node/client implementations as well.

STEP OF A TRANSACTION

TRANSACTIONS

1. Transaction is formed by user and sent to EVM to execute
2. Transaction has to undergo **intrinsic validity test**
3. Assuming it passes, then the intrinsic gas is taken from the user's account and nonce of user +=1
4. Then, transaction is passed into the transition function with machine state, system state, etc...
5. Transition function is called iteratively and changes(such as remaining_gas -= gas_used), at each loop, there are 3 options:
   a. If exception happens, just end and handle the halt correctly
   b. If more loops, then start again at step 4
   c. Else: come to halt successfully
6. Generates **resultant_state**, **remaining_gas**, **accrued_substate**, and **resultant_output**
7. Generates **transaction receipt** with the result of a transaction's execution, the **log events** of the execution, the **amount of gas** used

2 EXECUTION ON THE EVM

EVM STATE IS 8 TUPLE

ETHEREUM VIRTUAL MACHINE (EVM)

```
{
        block_state,  // also references storage
        transaction,  // current transaction
        message,      // current message
        code,         // current contract's code
        memory,       // memory byte array
        stack,        // words on the stack
        pc,           // program counter → code[pc]
        gas           // gas left to run tx
}
```

EVM STATE INSIDE CONTRACT

ETHEREUM VIRTUAL MACHINE (EVM)

```
Invariant per Contract:
        block_state, // also references storage
        transaction, // current transaction
        message,     // current message
        code         // current contract’s code
Contract State:
{
        pc,          // program counter → code[pc]
        gas,         // gas left to run tx
        stack,       // words on the stack
        memory,      // memory byte array
        storage      // K/V store of words
}
```

NAME REGISTRY DOWN TO EVM ASSEMBLY

ETHEREUM VIRTUAL MACHINE (EVM)

```
# nameregistry.se

# calldata(0)  - domain name - up to 32 bytes string
# calldata(32) - IP address - padded to LSB

if not self.storage[calldataload(0)]:
  self.storage[calldataload(0)] = calldataload(32)
```

Compiled to EVM assembly:

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI
STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0
CALLDATALOAD SSTORE
```

0x35 **CALLDATALOAD** - Get input data of current environment

NAME REGISTRY EXAMPLE

ETHEREUM VIRTUAL MACHINE (EVM)

We register a domain “54” with IP “20202020”

- Send Tx:
  - From: “Zvi 160-bit address”
  - To: “NameRegistry” contract’s address
  - Value: 0 ether
  - Data: [54, 20202020]
  - GasLimit: 2000 gas
  - GasPrice: 1.0 (1 gas == 1 wei)

NAME REGISTRY EXAMPLE

ETHEREUM VIRTUAL MACHINE (EVM)

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI
STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0
CALLDATALOAD SSTORE
```

Calldata [54, 20202020] is 2 words of 32 bytes = 64 bytes

- GasLimit * GasPrice = 2000 * 1 = 2000 wei

Tx costs:

- 500 + 5*TXDATALEN = 500 + 5*64 bytes = 820 gas

1150 gas consumer by Tx execution

- 2000 gas - 1150 gas = 850 gas refund

If we were setting GasLimit to less that 1150, the Tx would be failing in the middle and there would be no refund.

PC	OPCODE	FEE	GAS	STACK	MEM	STORAGE
0	PUSH1 0	-1	-820	[]	[]	{}
2	CALLDATALOAD	-1	-821	[0]	[]	{}
3	SLOAD	-20	-822	[54]	[]	{}
4	NOT	-1	-842	[0]	[]	{}
5	PUSH1 9	-1	-843	[1]	[]	{}
7	JUMPI	-1	-844	[1, 9]	[]	{}
8	STOP					
9	JUMPDEST	-1	-845	[]	[]	{}
10	PUSH1 32	-1	-846	[]	[]	{}
12	CALLDATALOAD	-1	-847	[32]	[]	{}
13	PUSH1 0	-1	-848	[2020202020]	[]	{}
15	CALLDATALOAD	-1	-849	[2020202020, 0]	[]	{}
16	SSTORE	-300	-850	[2020202020, 54]	[]	{}
			-1150	[]	[]	{54: 2020202020}

EVM EXECUTION EXAMPLE I

SIMPLE PARSING

```
pragma solidity ^0.4.11;

contract C {
   uint256 a;

   function C() {
     a = 1;
   }
}
```

```
60 01
60 00
81
90
55
50
```

```
tag_1:
 // 60 01
 0x1
 // 60 00
 0x0
 // 81
 dup2
 // 90
 swap1
 // 55
 sstore
 // 50
 pop
```

EVM EXECUTION EXAMPLE I

SIMPLE PARSING

```
// 60 01: pushes 1 onto
stack
0x1
 stack: [0x1]

// 60 00: pushes 0 onto
stack
0x0
 stack: [0x0 0x1]
```

```
// 81: duplicate the second
item on the stack
dup2
 stack: [0x1 0x0 0x1]

// 90: swap the top two
items
swap1
 stack: [0x0 0x1 0x1]
```

```
// 55: store the value 0x1
at position 0x0
// This instruction consumes
the top 2 items
sstore
 stack: [0x1]
 store: { 0x0 => 0x1 }

// 50: pop (throw away the
top item)
pop
 stack: []
 store: { 0x0 => 0x1 }
```

EVM EXECUTION EXAMPLE I

SIMPLE PARSING

```
// No need for dup2, swap1, pop

// a = 1
sstore(0x0, 0x1)

// All we need are the following three instructions
0x1
0x0
sstore
```

EVM EXECUTION EXAMPLE II

SIMPLE PARSING

```
contract C {
   uint256 a;
   uint256 b;
   function C() {
      a = 1;
      b = 2;
   }
}
```

```
// Pseudocode

// a = 1
sstore(0x0, 0x1)
// b = 2
sstore(0x1, 0x2)
```

EVM EXECUTION EXAMPLE III

STORAGE PACKING

```
// Each slot storage can store 32 bytes

contract C {
   uint128 a;
   uint128 b;
   function C() {
     a = 1;
     b = 2;
   }
}
```

- `sstore` costs 20000 gas for first write to a new position
- `sstore` costs 5000 gas for subsequent writes to an existing position. Why?
- `sload` costs 500 gas
- Most instructions costs 3~10 gases

EVM EXECUTION EXAMPLE III

STORAGE PACKING (OPTIMIZED)

```
$ solc --bin --asm --optimize c3.sol
```

```
tag_1:
   /* "c3.sol":95:96  a */
 0x0
   /* "c3.sol":95:100  a = 1 */
 dup1
 sload
   /* "c3.sol":108:113  b = 2 */
 0x200000000000000000000000000000000
 not(sub(exp(0x2, 0x80), 0x1))
...
```

```
...
  /* "c3.sol":95:100  a = 1 */
swap1
swap2
and
  /* "c3.sol":99:100  1 */
0x1
   /* "c3.sol":95:100  a = 1 */
or
sub(exp(0x2, 0x80), 0x1)
   /* "c3.sol":108:113  b = 2 */
and
or
swap1
sstore
```

3

EVM

BLOCKCHAIN
AT BERKELEY

WHAT IS THE EVM

ETHEREUM VIRTUAL MACHINE (EVM)

- Solidity as a **high-level language** creates an abstract of a machine, known as a **virtual machine**
- The virtual machine executes instructions in a way that guarantees every transaction will execute the same way on every Ethereum node in the network
- The **Ethereum Virtual Machine (EVM)** handles the actual processing of transactions that are proposed within a block
- It's turing complete, except the EVM is bounded by gas
  - total amount of computation that can be done is limited by the amount of gas provided
  - but even if I provide a ton of gas, I could theoretically hit the block gas limit
- **Stack based VM architecture**

VIRTUAL MACHINES

ETHEREUM VIRTUAL MACHINE (EVM)

- VMs are commonly used to distribute programs in an **architecture-neutral format**, which can easily be interpreted or compiled
  - Important for all the different architectures the Ethereum nodes probably need to run on, all of them should be able to **interpret** and **compile** Solidity to EVM bytecode!
- Some VMs like the Java VM uses a **virtual stack architecture**, rather than the **register architecture** that dominates in real processors (think of different languages for machines)
  - These two architectures exist because they are very simple, there's a sequence of instructions, some state, and semantics for every instruction
  - The languages are very minimal, syntactic sugar is all compiled away for high level languages

VIRTUAL MACHINES

ETHEREUM VIRTUAL MACHINE (EVM)

- Since a VM needs to emulate the operations carried out by a physical CPU, it should ideally encompass the following:
  - Compilation of **source language** into **VM specific bytecode**
  - **Data structures** to contain **instructions** and **operands** (the data the instructions process)
  - A **call stack** for function call operations
  - An **'Instruction Pointer' (IP)** pointing to the next instruction to execute
  - A **virtual 'CPU'** – the instruction dispatcher that
    - Fetches the next instruction (addressed by the instruction pointer)
    - Decodes the operands
    - Executes the instruction

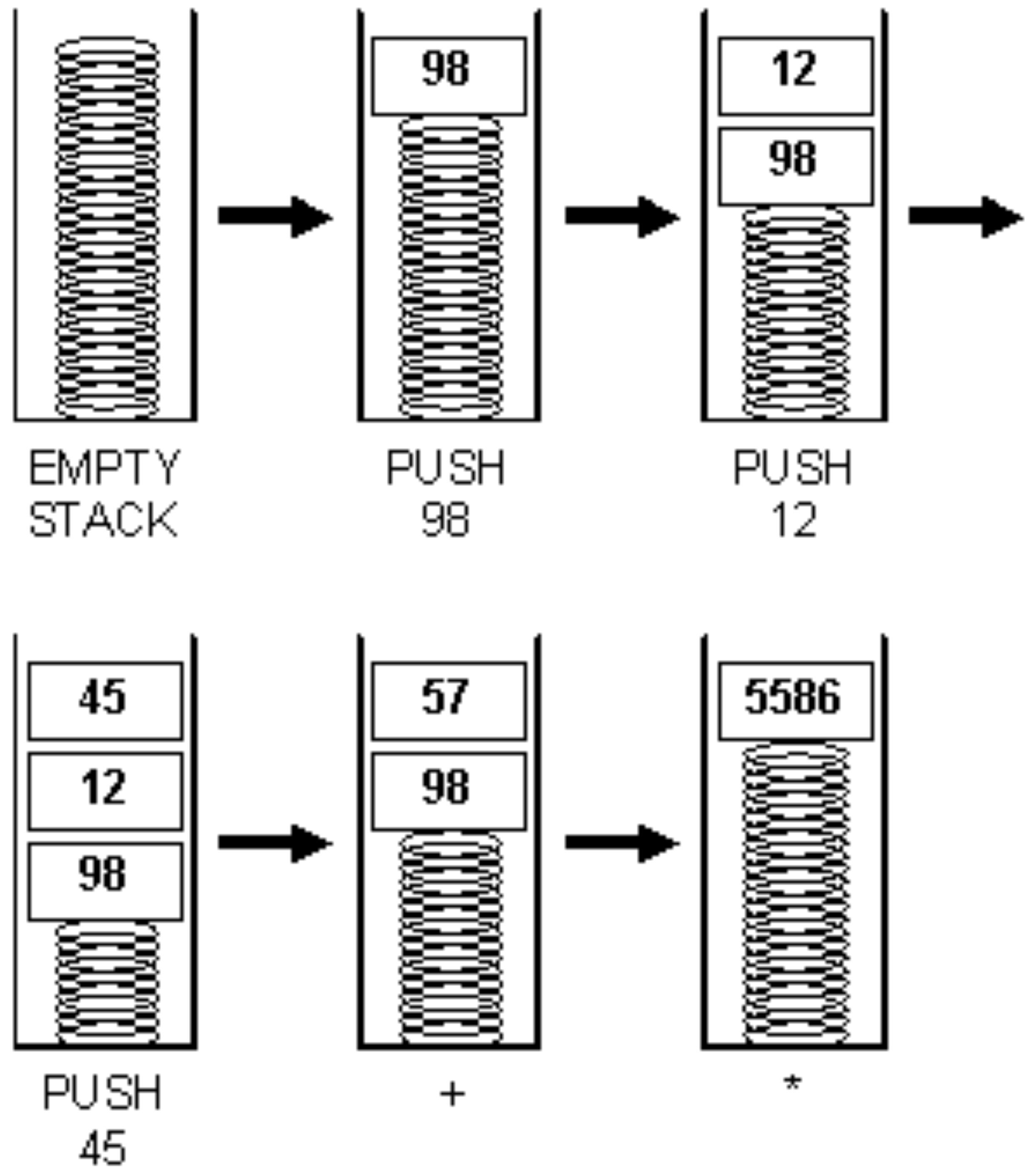
3.1 STACK VS REGISTER VMs

BLOCKCHAIN AT BERKELEY

STACK BASED ARCHITECTURE

ETHEREUM VIRTUAL MACHINE (EVM)

- Essentially a computer that uses a last-in, first out stack to hold temporary values
- Size of each stack item in the EVM is 256 bits
- Stack has a max size of 1024



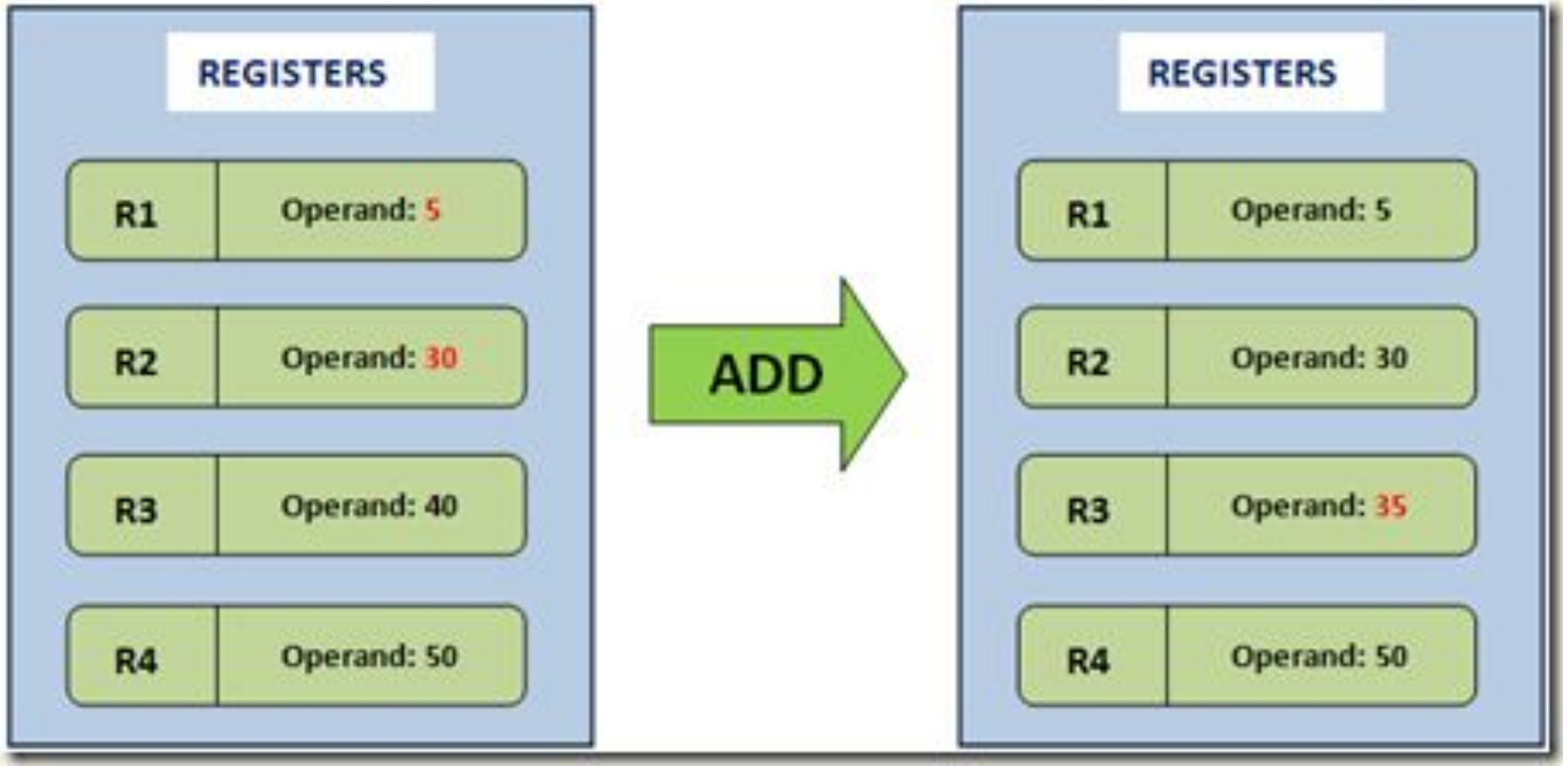
REGISTER BASED ARCHITECTURE

ETHEREUM VIRTUAL MACHINE (EVM)

- Not used by the EVM!
- Data structure where the operands are stored is based on the registers on the CPU (I need to specify where my variables are)
  - Whereas the stack based architecture had a stack pointer that points of operands
- No PUSH or POP operations, meaning less overhead
- The instructions in a register-based VM execute faster within the instruction dispatch loop
- Another advantage of this model are optimizations -
  - i.e. when there are common sub expressions in code, the register model can calculate it once and store the result in a register for future use when the sub expression comes up again
- A stack pointer results in shorter instructions, whereas explicit operand locations result in longer

REGISTER BASED ARCHITECTURE

ETHEREUM VIRTUAL MACHINE (EVM)



ADD R1, R2, R3 ; # Add contents of R1 and R2, store result in R3

STACK BASED ARCHITECTURE

ETHEREUM VIRTUAL MACHINE (EVM)

- Using a stack based virtual machine model has the advantage that it can be easily transferred to both register and stack machines, while the opposite is not necessarily true
- A register-based VM would need to make assumptions about the number of registers, the size of the registers etc
- With a stack machine, no such assumptions are necessary
- But stack machines are not what CPUs are optimized for
  - Think of registers like a cache for the top 5 or 6 (or whatever) items on the top of the stack
  - If the top of the stack is being accessed much, much more than the bottom (which is true in a lot of programs), then having this cache will speed things up

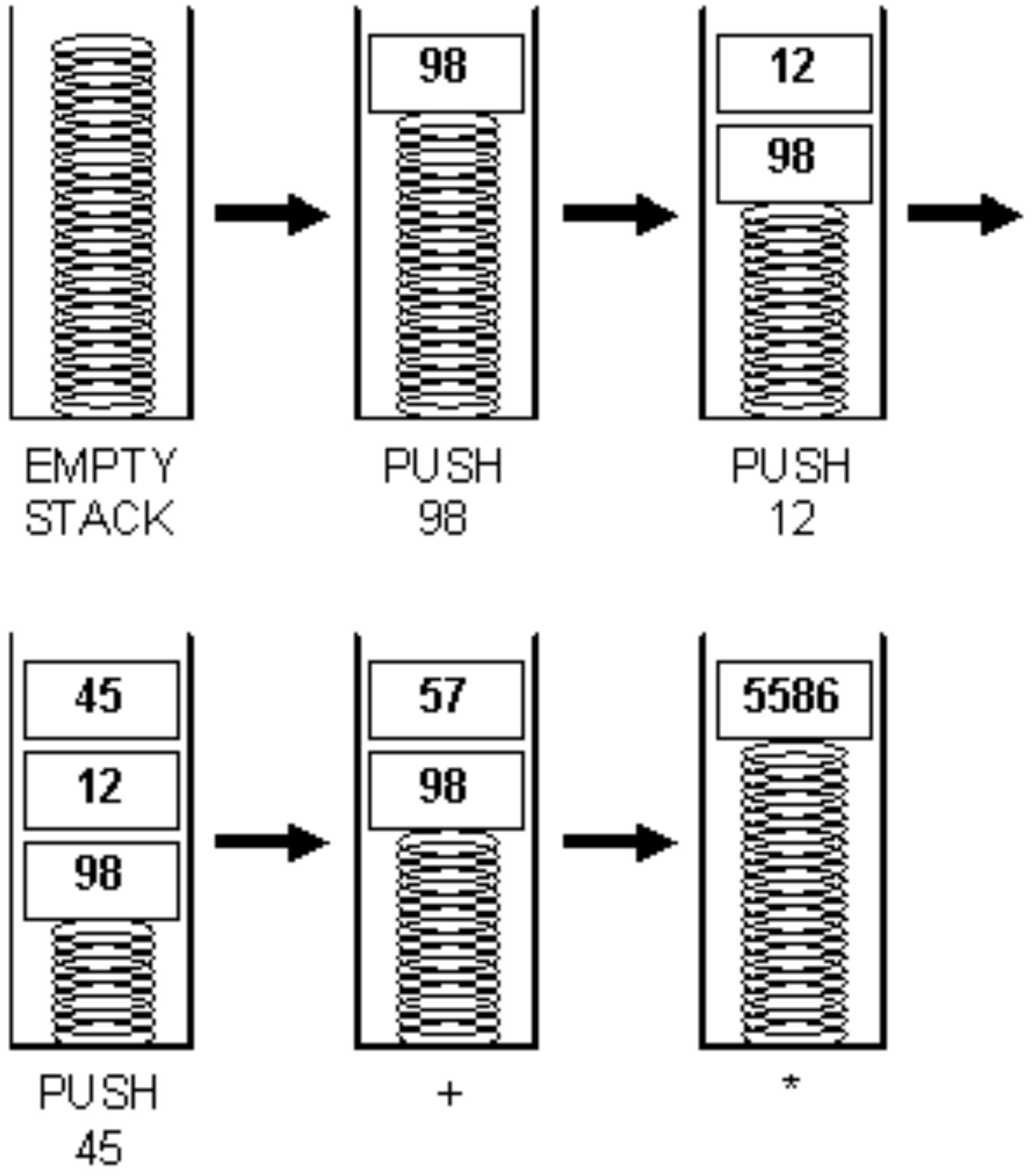
3.2 EVM PROPERTIES

BLOCKCHAIN
AT BERKELEY

STACK BASED ARCHITECTURE

ETHEREUM VIRTUAL MACHINE (EVM)

- Essentially a computer that uses a last-in, first out stack to hold temporary values
- Size of each stack item in the EVM is 256 bits (32B words)
- Stack has a max size of 1024



EVM STORAGE

ETHEREUM VIRTUAL MACHINE (EVM)

- The EVM has **memory**, where items are stored as word-addressed byte arrays
  - This memory is volatile, so it is not permanent
- The EVM also has key/value **storage**
  - This is the only VM which uses an associative array (or dictionary) for its address space
  - Storage is not volatile and is stored as part of the system state
- The EVM stores program **code** separately in a virtual ROM (Read-only memory) that can be accessed via special instructions
  - The EVM also has its own language, EVM Bytecode, compiled from Solidity

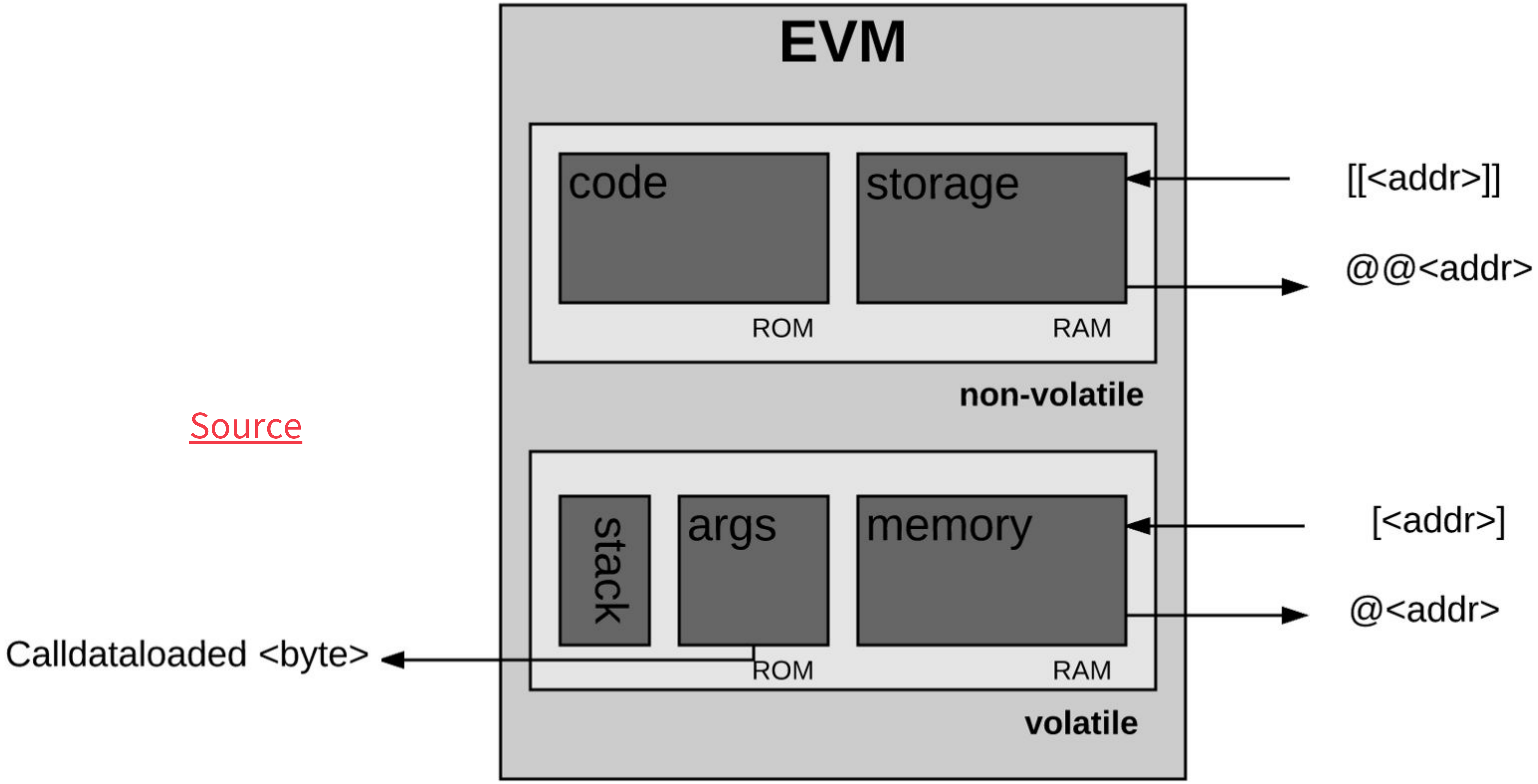
WHY 256 BIT WORDS

ETHEREUM VIRTUAL MACHINE (EVM)

- Crypto Primitives:
  - SHA256 (SHA3)
  - Public key is 256-bit uint
  - Private key uses secp256k1/EDCSA with 2 256-bit uints (r, s)
- 160-bit account addresses fit into 256-bit
- 256-bit SIMD [Single Instruction, Multiple Data] Instruction Set Architectures (SSE, AVX) on modern CPUs
  - (Look at SIMD Intrinsics for CPUs, easy to exploit data level parallelism.)
  - SIMD parallelism is conceptually similar to the Map-Reduce paradigm in that the same operation can be carried out on data distributed over multiple processors

EVM STORAGE

ETHEREUM VIRTUAL MACHINE (EVM)



Source

0x35 **CALLDATALOAD** - Get input data of current environment

EVM SECURITY

ETHEREUM VIRTUAL MACHINE (EVM)

- The EVM is a security oriented virtual machine, designed to permit untrusted code to be executed by a global network of computers. To do so securely, it imposes the following restrictions:
  - Every **computational step** taken in a program's execution must be **paid for up front**, thereby preventing **Denial-of-Service** attacks
  - Programs may only **interact** with each other **by transmitting a single arbitrary-length byte array**; they do not have access to each other's state
  - **Program execution is sandboxed**; an EVM program may access and modify its own internal state and may trigger the execution of other EVM programs, but nothing else
  - Program **execution** is fully deterministic and **produces identical state transitions** for any conforming implementation beginning in an identical state (important for consensus)

3.3

EVM COMPILER THEORY

SOLC COMPILER

ETHEREUM VIRTUAL MACHINE (EVM)

- How do we go from Solidity to Ethereum bytecode?
    - Using the Solc compiler, which is on all Ethereum clients
- Can break down into 5 steps:
    - Parsing
    - Syntax Checking
    - Type Checking
    - Static Analysis
    - Code Generation

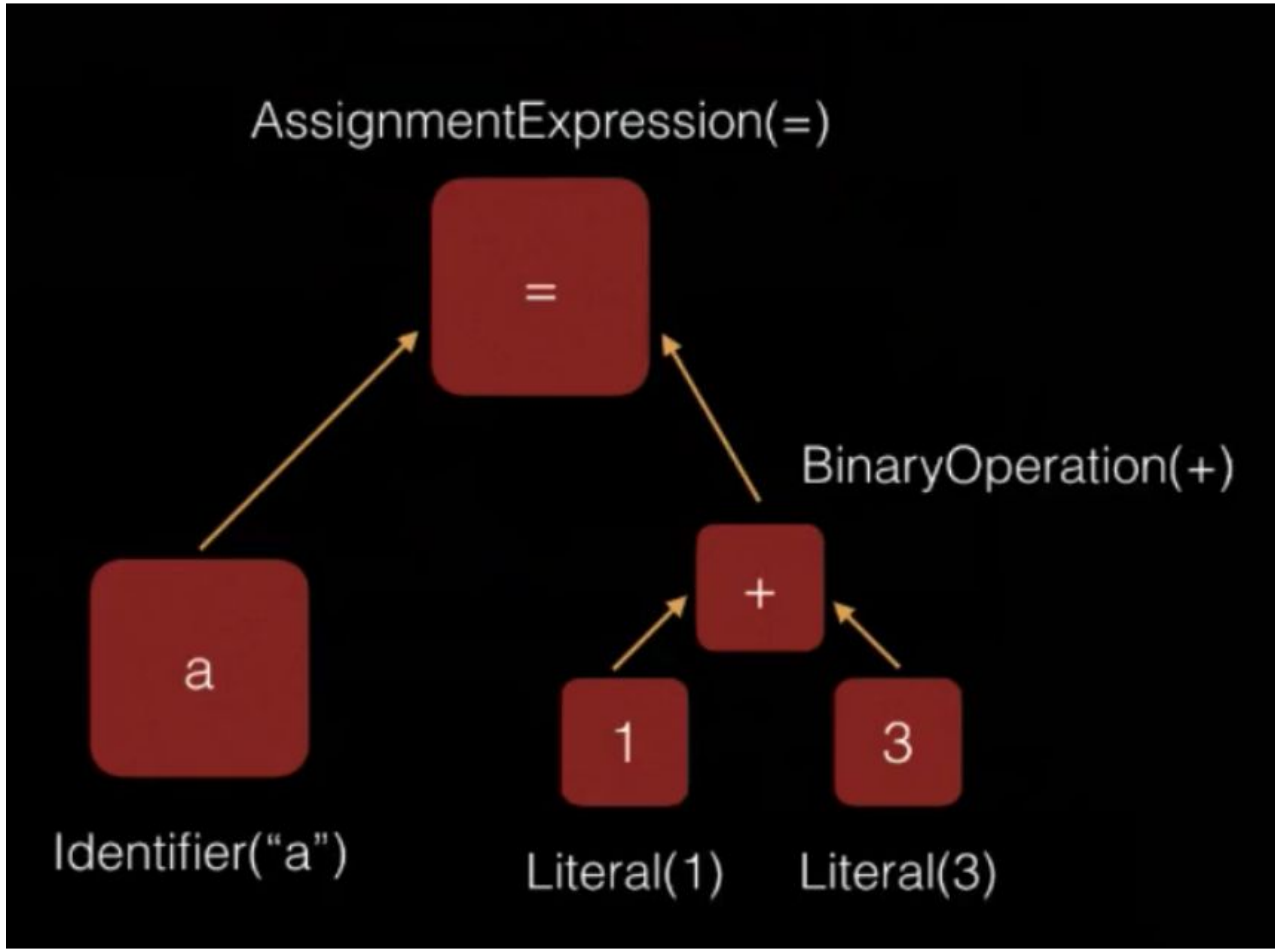
PARSING

ETHEREUM VIRTUAL MACHINE (EVM)

- Parsing transforms a stream of source code bytes into an **Abstract Syntax Tree (AST)**
- Stream of characters is taken from the source code, and then a tree is constructed for all the operations

Example:

```
a = 1 + 3;
```



DEMO:
https://github.com/ethereum/solidity/tree/develop/libsolidity

PARSING

ETHEREUM VIRTUAL MACHINE (EVM)

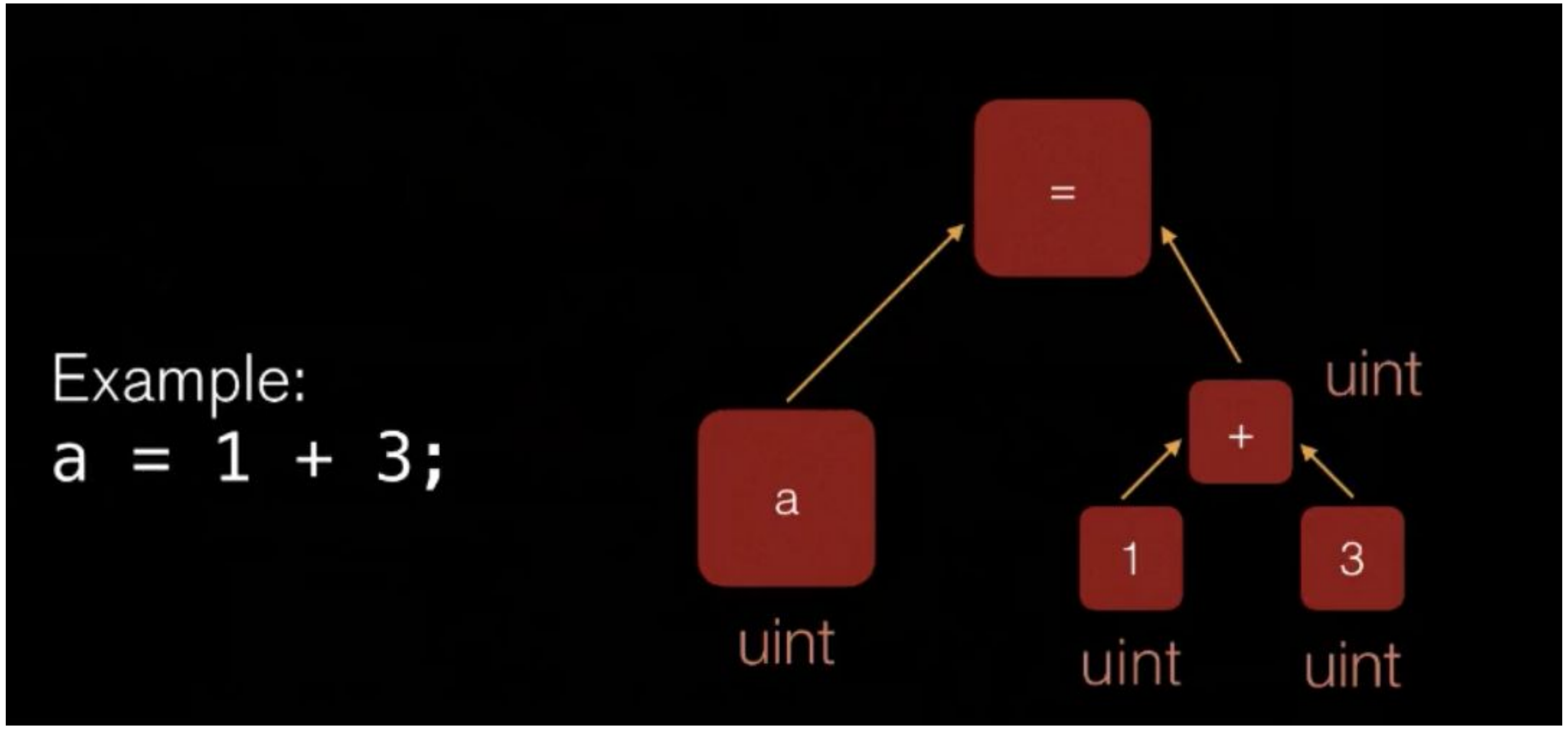
- **Lexer/scanner** is used to build the tree
  - Takes file characters and turns it into a stream of **discrete tokens**
  - Stream of tokens are now transferred to parser
  - Recursive descent parser, see what rules are applied for our tokens for **syntax checking**

```
[
contract, id:Test, lbracket, uint,
id:payed, eq, 0, eos, function,
lpar, rpar, lbracket, id:payable,
op_pluseq, msg, dot, value, eos,
rbracket, rbracket
]
```

TYPE CHECKING

ETHEREUM VIRTUAL MACHINE (EVM)

- **Type checking** - go through the AST nodes from the bottom and traverse upward
  - Check uints, strings, look for constructors, loops, etc.



STATIC ANALYZER + CODE GEN

ETHEREUM VIRTUAL MACHINE (EVM)

- **Static analyzer (for warnings and issues)**
  - Check if it's a library, payable functions, or deprecated functions
- **EVM Bytecode and Code Generation**
  - Compile with Solc, you get

```
// c2.sol
pragma solidity ^0.4.11;

contract C {
    uint256 a;
    uint256 b;

    function C() {
      a = 1;
      b = 2;
    }
}
```

```
$ solc --bin --asm c2.sol

// ... more stuff omitted

tag_2:
    /* "c2.sol":99:100  1 */
  0x1
    /* "c2.sol":95:96  a */
  0x0
    /* "c2.sol":95:100  a = 1 */
  dup2
  swap1
  sstore
  pop
    /* "c2.sol":112:113  2 */
  0x2
    /* "c2.sol":108:109  b */
  0x1
    /* "c2.sol":108:113  b = 2 */
  dup2
  swap1
  sstore
  pop
```

```
// a = 1
sstore(0x0, 0x1)
// b = 2
sstore(0x1, 0x2)
```

PROBLEMS WITH SOLC

ETHEREUM VIRTUAL MACHINE (EVM)

- Difficult to audit contracts
    - Does a function return what it is supposed to?
    - Is the compiler creating the bytecode according to the source code
- Compiler helpers and optimizations in Solidity are very complex
- Porting Solidity to other VMs
- Creating domain specific languages (DSLs) could be improved (can't trust the basis of Solidity, so gotta go back to dealing with the EVM, whereas DSLs these days completely trust C++)

PIPELINE OF A MODERN COMPILER

ETHEREUM VIRTUAL MACHINE (EVM)

- Front-end
  - Parses the source code
  - Analyzes
  - Generates intermediate representation of the program
- Middle end
  - Applies optimizations on the intermediate representation of the program
- Back-end
  - Generates the bytecode for the target machine and can do more optimizations

PIPELINE OF A MODERN COMPILER

ETHEREUM VIRTUAL MACHINE (EVM)

- Compilers work this way because they target multiple machines.
- If you look at GCC and LLVM, they support C, where you can compile the same code to multiple computers
- This applies to Solidity, but the main reason this makes sense is for verification of the program for the compiler

PIPELINE OF SOLC

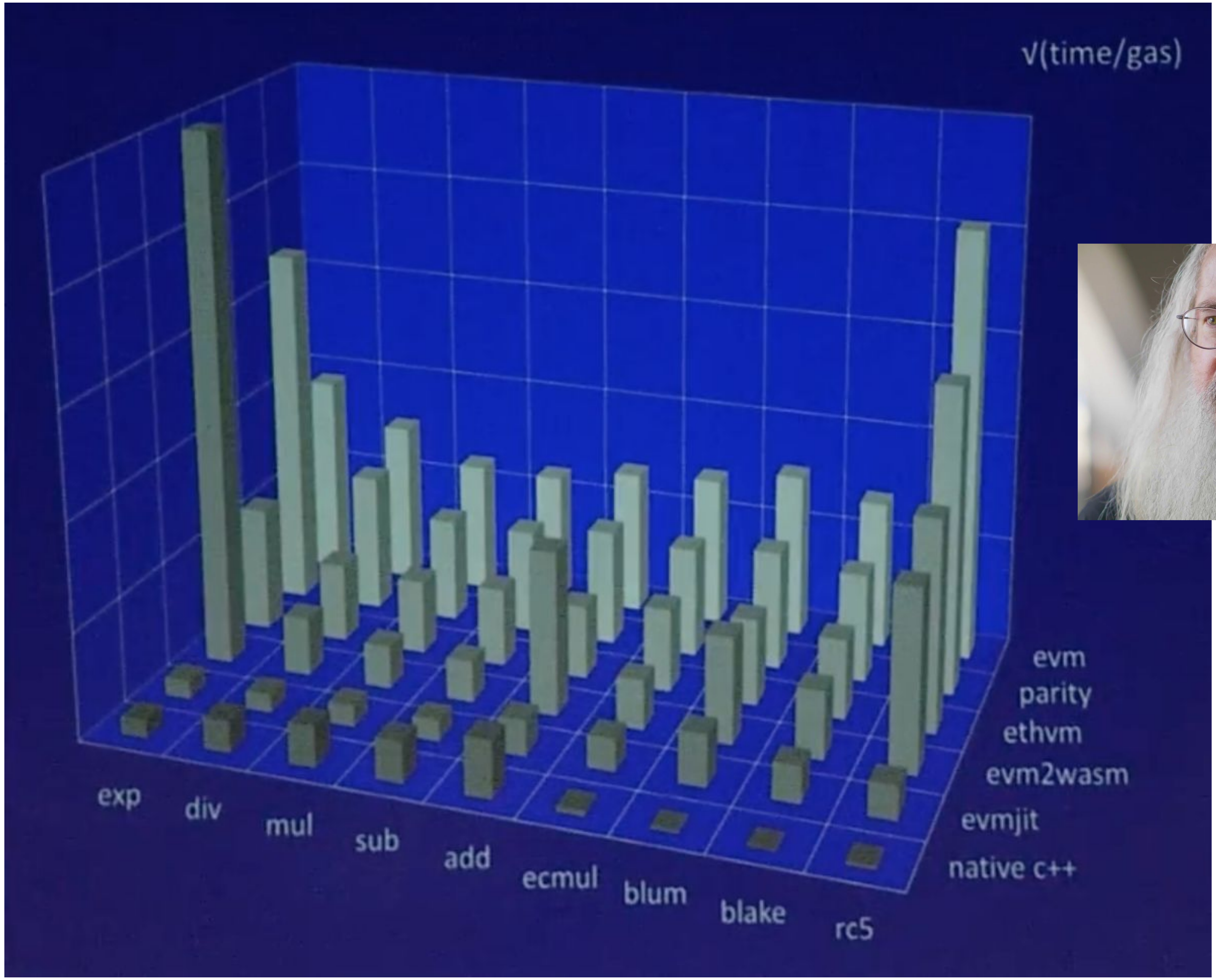
ETHEREUM VIRTUAL MACHINE (EVM)

- Front-end
  - Parses solidity
  - Analyzes
  - Generates EVM bytecode
- Back-end
  - Optimizes EVM bytecode

- But wait, where is the middle end, with optimizations for solidity?

INTERESTING GRAPH

ETHEREUM VIRTUAL MACHINE (EVM)



Thank you based Ethereum Wizard, Dr. Greg Colvin

WHY ARE EVM INTERPRETERS SLOWER

ETHEREUM VIRTUAL MACHINE (EVM)

- Interpreters have overhead (Python vs. C++ speed, see Benchmarks Game)
- 256 bit registers slow us down
  - Real hardware has 32/64bit registers
  - SIMD Intrinsics difficult to leverage on distributed, decentralized computers
- EVM currently does not have **hierarchically structured control** flow
  - This makes static analysis very difficult
  - Control flow paths currently go up **quadratically**
  - Impossible to track where next call will go path wise at scale

WHY IS EVM JIT SO QUICK?

ETHEREUM VIRTUAL MACHINE (EVM)

- A **JIT compiler** runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set
- It has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently
- However, securing is difficult, potential for a DoS attack, for Ethereum it’s needed to do compiling upfront at deployment time, else expected determinism becomes funky

4

EVM 2.0

WHY IS EVM 2.0 BETTER

ETHEREUM VIRTUAL MACHINE (EVM)

- Suggestion to extend the current EVM by adding new opcodes
- Forbid unconstrained jumps (encourage straight line code, and techniques like loop unrolling)
- Provide opcodes for **subroutines** (basically units that are sequences of program instructions, aka compiler functions)
  - Provide the ways you can do things with those jumps
- Provide opcodes for **native scalar** and **SIMD vector types**
  - SIMD crypto is helpful for speed especially
- Validates control flow, stack discipline and type safety of programs

WHAT'S SPECIAL ABOUT WEBASSEMBLY

ETHEREUM VIRTUAL MACHINE (EVM)

- See the design docs:
  - https://github.com/WebAssembly/design/blob/master/Rationale.md
- TL;DR
  - Much quicker parsing times
  - Not restricted to JavaScript for the web
  - Sandboxed for security
  - and more… , which is WiP

SEE YOU NEXT TIME

Scaling

Sharding

Casper

State Channels

Lightning/Plasma

IPFS (Extra)