Fall 2022

Virtual Machine for SpartanGold

William Wang
San Jose State University

Virtual Machine for SpartanGold

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

William Wang

December 2022

The Designated Project Committee Approves the Project Titled

Virtual Machine for SpartanGold

by

William Wang

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science

ABSTRACT

Virtual Machine for SpartanGold

by William Wang

The field of blockchain and cryptocurrencies can be both difficult to grasp and improve upon, which makes aids that can assist in these tasks very useful. SpartanGold is a simplified blockchain-based cryptocurrency created at San Jose State University as a learning aid for blockchain and cryptocurrencies. In its current state, it closely resembles Bitcoin, and it is also easily expandable to implement other features.

This project extends SpartanGold with a virtual machine resembling the Ethereum Virtual Machine. Implementing this feature results in SpartanGold having Ethereum-related features, which would allow the cryptocurrency to both be a helpful learning aid for Ethereum and be able to solve interesting blockchain problems associated with virtual machines and smart contracts.

Using my virtual machine implementation, I was able to produce a simplified token that resembles Ethereum tokens and works with SpartanGold. This token demonstrates the SpartanGold Virtual Machine's usefulness in simulating smart contracts of real world interest. Going forward, developers can experiment with the SpartanGold Virtual Machine to test out new ideas without dealing with the full complexity of the Ethereum Virtual Machine.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

1.1 Brief Background

Virtual machines are an important concept in the field of blockchain and cryptocurrency, as they are an essential part to running contract bytecode. The most well-known example of a cryptocurrency virtual machine is the Ethereum Virtual Machine (EVM) from Ethereum, which allows the cryptocurrency to execute powerful smart contracts that can perform interesting transactions, such as running decentralized applications (dApps). A lesser known example is the Bitcoin Script for Bitcoin, which also allows the cryptocurrency to execute contracts, although these tend to be less powerful and do just enough to perform simple transactions. Having such a feature in a cryptocurrency allows it to be more flexible in its transactions and do more interesting tasks with its tokens, so it is imperative to understand how they work under the hood and implement one for the language.

1.2 Motivation

The source code behind established cryptocurrencies can be difficult to grasp and to improve upon, especially when it comes to virtual machines. It is often useful to have a concrete example on hand to visualize the processes being performed under the hood when experimenting in the field of blockchain and cryptocurrency, so an alternative token that is easier to manipulate is required. At San Jose State University, SpartanGold was created to act as a simplified blockchain-based cryptocurrency to be used as a learning aid and as a springboard for implementing additional features. The cryptocurrency resembles Bitcoin but is significantly easier to work with, making it a perfect candidate for implementing a simple virtual machine that can run smart contracts.

1.3 The SpartanGold Virtual Machine

This project seeks to add virtual machine functionality to SpartanGold, allowing it to run smart contracts. The virtual machine's design is based on the Ethereum Virtual Machine and shares many opcode functionalities with it. The source code for the virtual machine and its supporting files is written entirely in JavaScript, allowing it to seamlessly connect with the JavaScript-based SpartanGold. The bytecode is formatted similarly to the ones generated by Ethereum smart contracts, allowing them to be run by the SpartanGold virtual machine. By attaching a virtual machine that can take in bytecode resembling Ethereum smart contracts, clients can use their SpartanGold tokens to facilitate a transaction similar to those done in Ethereum. The implementation uses special clients and blocks that store and retrieve smart contracts, a virtual machine to translate the bytecode, and an opcode table to implement the functionalities needed.

1.4 Competing Approaches

To the best of my knowledge, in the blockchain education space there are very few competing approaches. Many papers explain the concepts in simpler terms like in Srivastava et al. [1], but my approach provides a physical token that can be manipulated to enhance the learning experience. The underlying SpartanGold structure has also been extended to create new uses for the blockchain, such as the creation of TontineCoin from Pardeshi et al. [2] and a crowdfunding non-fungible token from Basu et al. [3], but these approaches are not necessarily focused on blockchain education.

1.5 Expected Results

The overall effect of my implementation is that clients will be able to create smart contracts that transfer SpartanGold tokens between each other, allowing SpartanGold

to resemble an easier to manipulate version of Ethereum. This token, which resembles a native coin in standard currencies, will then be able to solve interesting Ethereum problems in the SpartanGold system without needing to use the actual Ethereum token. As a proof of concept, I produced a simplified token with basic functionality equivalent to the Ethereum token using the virtual machine I implemented.

1.6 Paper Summary

In the following sections, I will be detailing various aspects of my implementation of the SpartanGold virtual machine. First, I will describe how other established cryptocurrencies designed their virtual machines and the features I chose to adapt to my implementation. Next, I will describe the structure of my implementation, discussing my design choices and challenges while implementing the virtual machine. Then, I will describe the types of contracts that can be written to run on the SpartanGold virtual machine with examples I have already written. Finally, I will discuss any future improvements and work that can be done to expand on the SpartanGold virtual machine.

CHAPTER 2

Background and Related Work

In this section, I will go into more detail about some of the background and other works related to my project, such as the Bitcoin Script and the Ethereum Virtual Machine.

2.1 Bitcoin Script

The Bitcoin Script [4] is the scripting language used by Bitcoin for its transactions. The language is based on Forth, and uses a stack-based instruction sequence that is run from left to right. To maintain stability, the language is also intentionally designed to be not Turing-complete, so it has no built-in loop structure.

Scripts written in the Bitcoin Script language typically describe conditions that must be met before a recipient is able to claim and spend the Bitcoins associated with the transaction. These requirements usually involve a public key associated with the recipient and a signature associating private key ownership with the public key, but the language allows for different scripts that require more or less conditions to be met. A valid transaction is made when nothing in the associated script fails and the top stack item evaluates to a non-zero value upon the script's conclusion, which is checked by combining the script provided by the Bitcoin sender with the inputs provided by the Bitcoin recipient.

2.2 Ethereum Virtual Machine

The Ethereum Virtual Machine [5] is a virtual state machine that provides an environment allowing for the execution of more sophisticated contracts, commonly referred to as "smart contracts". The system also uses a stack-based instruction sequence to run contracts, but a major difference is its quasi-Turing-complete design, which allows it to run any computable problem if given enough resources. Stability is maintained instead through a "gas" system, where instructions are only run if enough

gas is provided to run them.

Scripts written for the Ethereum Virtual Machine tend to be more complex and can be run separate from other Ethereum processes. In addition to setting requirements that must be met in order to transfer Ethereum, scripts can also be written to store data on the blockchain, run decentralized applications (dApps), exchange alternate tokens, and much more. Once a smart contract is run to completion, any Ethereum transferred, as well as any other changes to the blockchain, are applied and broadcast to the rest of the network, provided all the conditions set by the contract creator have been met.

2.3 Other Relevant Works

In addition to the Bitcoin Script and Ethereum Virtual Machine, there are other blockchain-related projects that utilize virtual machines in their implementations as well, most of which are based on the Ethereum Virtual Machine. Ellul et al. [6] designed a virtual machine based on the Ethereum Virtual Machine called AlkylVM that attempts to integrate the Internet of Things (IoT) with the blockchain. Both Khoury et al. [7] and Puneet et al. [8] used the Ethereum Virtual Machine to create a decentralized voting platform on the Ethereum blockchain. Westerkamp et. al [9] designed a supply-chain traceability system that runs on the Ethereum Virtual Machine. From these examples, it is clear that understanding and implementing a virtual machine is an important topic in the blockchain and cryptocurrency space.

CHAPTER 3

Design and Implementation

In this section, I will go over the design choices and implementation of the SpartanGold Virtual Machine.

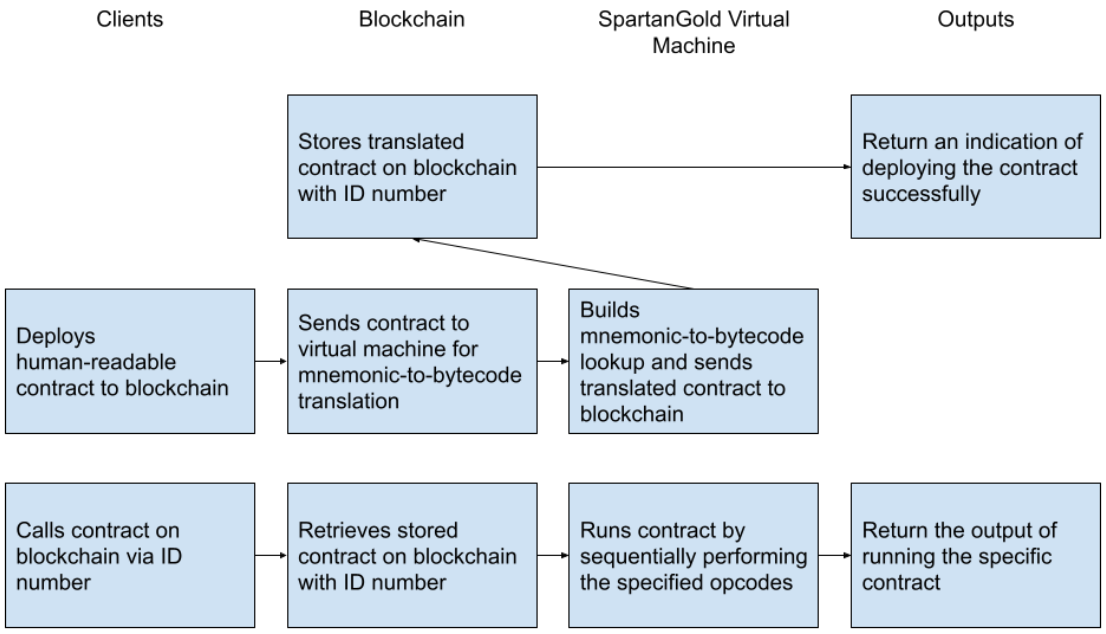

Figure 1: Execution Sequence for the SpartanGold Virtual Machine

3.1 Virtual Machine Functionality Walkthrough

The virtual machine accepts a human-readable file with a list of opcodes via a file name read from arguments passed in from a client.

```
// Alice deploys a contract.
setTimeout(() => {
  alice.postContractDeployTransaction([], {
    bytecodeFile: "test.gleam",
    gasPrice: 100,
    gasLimit: 100000
  });
}, 1000);
```

Figure 2: Passing in file

The text, which represents a smart contract, is first translated to bytecode, which builds up a mnemonic-to-bytecode map within the virtual machine.

```
/**
* Translates a text file of opcode mnemonics to a buffer of bytecode.
* @param {File} instructions - A file of the opcodes in hex format.
* @returns {Buffer} - A buffer of the bytecode instructions.
*/
translate(instructions) {
  let lookupTable = this.buildMnemonicLookup();
  let contents = fs.readFileSync(instructions, 'utf8');
  let output = "";
  let lines = contents.trim().split(/\r?\n/);
  lines.forEach((ln) => {
    // Skip comment lines and empty lines.
    if (ln.startsWith("//") || ln.match(/^\s*$/)) {
      return;
    }
    ln.split(' ').forEach((v) => {
      let byte = lookupTable[v];

      output += byte !== undefined ? byte : this.toByteString(v);
    });
  });
  return output;
}
```

Figure 3: Translating to bytecode

Next, the bytecode is stored on the blockchain with an associated ID value, allowing other clients to run the contract in the future.

```
this.contracts.set(this.numContracts, raw);
this.numContracts++;
```

Figure 4: Translating to bytecode

When the contract is run, the virtual machine retrieves the bytecode from the blockchain's storage via the same ID number.

```
let bytecodeString = this.lastBlock.contracts.get(data.contractAddress);
```

Figure 5: Retrieving file

Then, the opcodes are run sequentially by looking up the operation's definition and performing the required actions.

```
opcodes[0x01] = new OpCode('ADD', 3, (args, vm) => {
  let x = BigNumber.from(vm.stack.pop());
  let y = BigNumber.from(vm.stack.pop());
  let res = x.add(y);
  vm.stack.push(res.toHexString());
});
```

Figure 6: Looking up opcode

The result is an output that represents the outcome of running the contract, which depends on how the contract was written.

```
let result = this.lastBlock.vm.evaluate(bytecodeString, 1000, this.address,
                                        data.contractAddress, 1, data.args);
outputs.push({amount: data.amount, address: result.val});
this.lastBlock.storage.concat(result.storage);
```

Figure 7: Consolidating results

3.2 Shared Ethereum Features

To emulate the Ethereum Virtual Machine as closely as possible, I made sure to include some features of Ethereum in my design, such as gas and memory storage.

3.2.1 Gas

To implement the gas feature, a value determined from the Ethereum Yellow Paper is set as a fixed gas price for an operation, which is deducted from the total gas a client provides beforehand in order to run the contract. Some operations have variable costs, but I set the price to be an arbitrary fixed value of 0 in my implementation to keep it simple.

Table 1: Sample Opcodes and Costs

Opcode	Cost	Notes
ADD	3	Opcode with fixed cost. Value determined from Ethereum Yellow Paper.
CODECOPY	0	Opcode with variable cost. Value set to 0 for simplicity. Actual cost depends on how many bytes of data are being copied, with a base cost of 3 plus 3 for each byte copied.

3.2.2 Storage

To implement the two types of memory from the Ethereum Virtual Machine in my implementation, I used a series of lists stored in the virtual machine and on the blockchain. Memory, or short-term memory, is stored in a list on the virtual machine. It has elements added and used via certain operations, and is cleared at the conclusion of the contract. Storage, or long-term memory, performs similarly to memory, except that its elements are retrieved and stored on the blockchain at the conclusion of the contract.

3.3 Challenges

Naturally, some opcodes were easier than others, but a few of them presented interesting design choices in my implementation. Below, I list a few representative examples, as well as how I chose to implement it in the SpartanGold Virtual Machine.

3.3.1 SMOD

In order to implement opcodes that operated on signed values such as SMOD, I needed to be able to represent 256-bit numbers. However, JavaScript is not able to store values that big, which presented a problem. To solve this issue, I decided to store any values that needed up to 256 bits using the BigNumber class from Ethereum, which could represent and manipulate the values properly.

3.3.2 KECCAK256

Some operations, such as the KECCAK256 opcode, involved an algorithm that would have to implemented in order to function properly. However, working implementations of those algorithms also existed via predefined library functions and would also allow the opcode to function properly. To reduce the amount of potential bugs in my implementation, I chose to go with previously implemented library functions for these kinds of opcodes.

3.3.3 JUMP/JUMPI

A pair of opcodes that was particularly challenging to implement was the set of jump opcodes, JUMP and JUMPI. Because my virtual machine's program counter counted up by byte and the jump destinations in the bytecode referred to line numbers, I needed a way to associate the current byte and the current line in order for jumps to work correctly. I was able to solve this problem via a dictionary, with the key being the line number and the value being the byte number. This system would allow the bytecode to remain as is, while jumping the code to the actual correct location.

3.3.4 PUSHARG

An opcode that does not exist in the Ethereum Yellow Paper's instruction set that I implemented was PUSHARG. Similar to the other push opcodes, this operation pushed data onto the virtual machine's stack. However, instead of pushing a specific number of bytes of data read from the space after the opcode, PUSHARG pushes an arbitrarily-sized data element read from the list of arguments passed in with the contract. By including this opcode, I was able to insert any necessary arguments needed for a particular contract at the time it is needed without having to know exactly where it is located in the bytecode.

CHAPTER 4

List of Opcodes in the SpartanGold Virtual Machine

This is a list of every implemented opcode in the SpartanGold Virtual Machine, sorted by bytecode value. The associated bytecode values for each opcode are identical to the ones from the Ethereum Yellow Paper, with the exception of user-defined opcodes, which use unassigned bytecode values.

4.1 0s: Stop and Arithmetic Operations

These opcodes allow the code to perform arithmetic operations such as addition, subtraction, multiplication, and division on unsigned, 256-bit integers, to perform mod opcodes on unsigned, 256-bit integers, and to stop the bytecode execution. Some opcodes are only used with signed, twos-complement integers, and some opcodes are a combination of multiple simpler opcodes run together.

Bytecode	Opcode	Cost	Description
0x00	STOP	0	Halts execution.
0x01	ADD	3	Addition operation.
0x02	MUL	5	Multiplication operation.
0x03	SUB	3	Subtraction operation.
0x04	DIV	5	Integer division operation.
0x05	SDIV	5	Signed integer division operation (truncated).
0x06	MOD	5	Modulo remainder operation.
0x07	SMOD	5	Signed modulo remainder operation.
0x08	ADDMOD	8	Modulo addition operation.
0x09	MULMOD	8	Modulo multiplication operation.

4.2 10s: Comparison and Bitwise Logic Operations

These opcodes allow the code to perform comparison operations such as "less than", "greater than", "equal to", and "is zero" on unsigned, 256-bit integers and to perform bit-manipulating operations such as "and", "or", "xor", "not", "left shift", and "right shift" on unsigned, 256-bit integers. Some opcodes are only used on signed, twos-complement integers.

Bytecode	Opcode	Cost	Description
0x10	LT	3	Less-than comparison.
0x11	GT	3	Greater-than comparison.
0x12	SLT	3	Signed less-than comparison.
0x13	SGT	3	Signed greater-than comparison.
0x14	EQ	3	Equality comparison.
0x15	ISZERO	3	Simple not operator.
0x16	AND	3	Bitwise AND operation.
0x17	OR	3	Bitwise OR operation.
0x18	XOR	3	Bitwise XOR operation.
0x19	NOT	3	Bitwise NOT operation.
0x1b	SHL	3	Left shift operation.
0x1c	SHR	3	Logical right shift operation.

4.3 20s: KECCAK256

This opcode allows the code to calculate a Keccak-256 hash, which is often used to calculate addresses in cryptocurrencies such as Ethereum. Usually, the cost for performing a KECCAK256 operation depends on the size of the input data, with a base cost of 30 plus 6 for each byte of input data, but to keep the code simple for a prototype, I decided to make the operation free, since the reduction is fairly insignificant compared to the other opcodes.

Bytecode	Opcode	Cost	Description
0x20	KECCAK256	0 (variable)	Compute Keccak-256 hash.

4.4 30s: Environmental Information

These opcodes allow the code to access information related to the current running environment, such as the addresses of the caller and account, the values of the input and output data, and the sizes of the input and output data. An additional opcode allows the code to copy itself into memory, which usually has a base cost of 3 plus 3 for each byte copied, but to keep the code simple for a prototype, I decided to make the operation free, since the reduction is fairly insignificant compared to the other opcodes.

Bytecode	Opcode	Cost	Description
0x30	ADDRESS	2	Get address of currently executing account.
0x33	CALLER	2	Get caller address.
0x34	CALLVALUE	2	Get deposited value by the instruction or transaction responsible for this execution.
0x35	CALLDATALOAD	3	Get input data of current environment.
0x36	CALLDATASIZE	2	Get size of input data in current environment.
0x38	CODESIZE	2	Get size of code running in current environment.
0x39	CODECOPY	0 (variable)	Copy code running in current environment to memory.

4.5 40s: Block Information

These opcodes allow the code to access information related to the block, such as the block hash, the block reward address, the block timestamp, and other block-related information.

Bytecode	Opcode	Cost	Description
0x40	BLOCKHASH	20	Get the hash of one of the 256 most recent complete blocks.
0x41	COINBASE	2	Get the current block's beneficiary address.
0x42	TIMESTAMP	2	Get the current block's timestamp.

4.6 50s: Stack, Memory, Storage and Flow Operations

These opcodes allow the code to manipulate internal data storage, prepare data items for external data storage, and make conditional moves within the code. The four areas of data manipulation where these opcodes operate are on the instruction stack, the temporary internal memory, the permanent external memory, and valid jump locations. Usually, external memory storage operations have a variable cost dependent on the type of data stored and the status of the storage area, but to keep the code simple for a prototype, I decided to make the operation free. Unlike the other variable opcodes, this reduction is not insignificant since the original costs are at least 100 times more than other opcodes.

Bytecode	Opcode	Cost	Description
0x50	POP	2	Remove item from stack.
0x51	MLOAD	3	Load word from memory.
0x52	MSTORE	3	Save word to memory.
0x54	SLOAD	0 (variable)	Load word from storage.
0x55	SSTORE	0 (variable)	Save word to storage.
0x56	JUMP	8	Alter the program counter.
0x57	JUMPI	10	Conditionally alter the program counter.
0x5b	JUMPDEST	1	Mark a valid destination for jumps.

4.7 60s and 70s: Push Operations

These opcodes allow the code to push data items ranging between sizes of 1 and 32 bytes onto the instruction stack. Unlike the other opcodes, these operations read the data item to be pushed from directly adjacent to them in the bytecode.

Bytecode	Opcode	Cost	Description
0x60	PUSH1	3	Place 1 byte item on stack.
0x61	PUSH2	3	Place 2 byte item on stack.
0x62	PUSH3	3	Place 3 byte item on stack.
0x63	PUSH4	3	Place 4 byte item on stack.
0x64	PUSH5	3	Place 5 byte item on stack.
0x65	PUSH6	3	Place 6 byte item on stack.
0x66	PUSH7	3	Place 7 byte item on stack.
0x67	PUSH8	3	Place 8 byte item on stack.
0x68	PUSH9	3	Place 9 byte item on stack.
0x69	PUSH10	3	Place 10 byte item on stack.
0x6a	PUSH11	3	Place 11 byte item on stack.
0x6b	PUSH12	3	Place 12 byte item on stack.
0x6c	PUSH13	3	Place 13 byte item on stack.
0x6d	PUSH14	3	Place 14 byte item on stack.
0x6e	PUSH15	3	Place 15 byte item on stack.
0x6f	PUSH16	3	Place 16 byte item on stack.
0x70	PUSH17	3	Place 17 byte item on stack.
0x71	PUSH18	3	Place 18 byte item on stack.
0x72	PUSH19	3	Place 19 byte item on stack.
0x73	PUSH20	3	Place 20 byte item on stack.
0x74	PUSH21	3	Place 21 byte item on stack.
0x75	PUSH22	3	Place 22 byte item on stack.
0x76	PUSH23	3	Place 23 byte item on stack.
0x77	PUSH24	3	Place 24 byte item on stack.
0x78	PUSH25	3	Place 25 byte item on stack.
0x79	PUSH26	3	Place 26 byte item on stack.
0x7a	PUSH27	3	Place 27 byte item on stack.
0x7b	PUSH28	3	Place 28 byte item on stack.
0x7c	PUSH29	3	Place 29 byte item on stack.
0x7d	PUSH30	3	Place 30 byte item on stack.
0x7e	PUSH31	3	Place 31 byte item on stack.
0x7f	PUSH32	3	Place 32 byte item on stack.

4.8 80s: Duplication Operations

These opcodes allow the code to duplicate data items on the instruction stack, ranging from the 1st to 16th items, counting down. Duplicated data items are placed at the top of the instruction stack.

Bytecode	Opcode	Cost	Description
0x80	DUP1	3	Duplicate 1st stack item.
0x81	DUP2	3	Duplicate 2nd stack item.
0x82	DUP3	3	Duplicate 3rd stack item.
0x83	DUP4	3	Duplicate 4th stack item.
0x84	DUP5	3	Duplicate 5th stack item.
0x85	DUP6	3	Duplicate 6th stack item.
0x86	DUP7	3	Duplicate 7th stack item.
0x87	DUP8	3	Duplicate 8th stack item.
0x88	DUP9	3	Duplicate 9th stack item.
0x89	DUP10	3	Duplicate 10th stack item.
0x8a	DUP11	3	Duplicate 11th stack item.
0x8b	DUP12	3	Duplicate 12th stack item.
0x8c	DUP13	3	Duplicate 13th stack item.
0x8d	DUP14	3	Duplicate 14th stack item.
0x8e	DUP15	3	Duplicate 15th stack item.
0x8f	DUP16	3	Duplicate 16th stack item.

4.9 90s: Exchange Operations

These opcodes allow the code to exchange the positions of data items on the instruction stack, ranging from the 2nd to 17th items, counting down, which are exchanged with the 1st item.

Bytecode	Opcode	Cost	Description
0x90	SWAP1	3	Exchange 1st and 2nd stack item.
0x91	SWAP2	3	Exchange 1st and 3rd stack item.
0x92	SWAP3	3	Exchange 1st and 4th stack item.
0x93	SWAP4	3	Exchange 1st and 5th stack item.
0x94	SWAP5	3	Exchange 1st and 6th stack item.
0x95	SWAP6	3	Exchange 1st and 7th stack item.
0x96	SWAP7	3	Exchange 1st and 8th stack item.
0x97	SWAP8	3	Exchange 1st and 9th stack item.
0x98	SWAP9	3	Exchange 1st and 10th stack item.
0x99	SWAP10	3	Exchange 1st and 11th stack item.
0x9a	SWAP11	3	Exchange 1st and 12th stack item.
0x9b	SWAP12	3	Exchange 1st and 13th stack item.
0x9c	SWAP13	3	Exchange 1st and 14th stack item.
0x9d	SWAP14	3	Exchange 1st and 15th stack item.
0x9e	SWAP15	3	Exchange 1st and 16th stack item.
0x9f	SWAP16	3	Exchange 1st and 17th stack item.

4.10 a0s: Logging Operations

These opcodes allow the code to maintain a log record to keep track of the the current status of the instruction stack. A log record can be empty to indicate an empty stack, or up to the top 4 data items can be added as one record.

Bytecode	Opcode	Cost	Description
0xa0	LOG0	0 (variable)	Append log record with no topics.
0xa1	LOG1	0 (variable)	Append log record with one topics.
0xa2	LOG2	0 (variable)	Append log record with two topics.
0xa3	LOG3	0 (variable)	Append log record with three topics.
0xa4	LOG4	0 (variable)	Append log record with four topics.

4.11 b0s: Miscellaneous Operations

These opcodes are only defined in the SpartanGold Virtual Machine. They are defined here because other scripts do not use the bytecode values starting from this range, allowing for the custom definition of operations. The only opcode defined here pushes an arbitrarily-sized data item onto the stack, similar to the push opcodes. Unlike the push opcodes, the operation does not read from directly adjacent data, but from data passed in with the bytecode via data arguments.

Bytecode	Opcode	Cost	Description
0xb0	PUSHARG	0 (user-defined)	Place 1st argument passed in on stack.

4.12 f0s: System operations

These opcodes allow the code to perform system-related operations, such as modifying the program state, producing an output value, or indicating invalid opcodes. While every operation here is effectively free, the designated invalid instruction is technically not an operation and does not have a defined cost.

Bytecode	Opcode	Cost	Description
0xf3	RETURN	0	Halt execution returning output data.
0xfd	REVERT	0	Halt execution reverting state changes but returning data and remaining gas.
0xfe	INVALID	0 (null operation)	Designated invalid instruction.

CHAPTER 5

SpartanGold Integration

Integrating the SpartanGold Virtual Machine with SpartanGold was a fairly straightforward process, but it required the creation of specialized programs for the block and client in order for everything to work properly. The structure of the SpartanGold code and details about these specialized programs are described in the following sections.

5.1 SpartanGold Code Structure

The code structure of SpartanGold has five main classes that are combined together via a driver class. The `Block` class defines the collection of SpartanGold transactions and links to the previous block. The `Blockchain` class maintains the settings and configurations of the SpartanGold blockchain, which allows for custom blocks. The `Transaction` class defines the structure of a SpartanGold transaction, including any information that must be passed along in the blockchain. The `FakeNet` class simulates a network that allows SpartanGold transactions to be made and recorded. The `Client` class defines a SpartanGold user with functions that allow for SpartanGold transactions to be made. The classes of relevance for integrating the SpartanGold Virtual Machine and SpartanGold are the `Block` and `Client` classes. The network used on the SpartanGold Virtual Machine is unchanged so the `FakeNet` class should also be unchanged. While the blockchain and transactions on the SpartanGold Virtual Machine differ slightly from SpartanGold, the structure of the `Blockchain` and `Transaction` classes must remain as is in order for customization to work. Therefore, the `Block` and `Client` classes, which require additional or updated functionalities on the SpartanGold Virtual Machine, must be extended.

5.2 VmBlock

The virtual machine's `VmBlock` class is an extension of the `Block` class from SpartanGold, with the following modifications:

- `updateContracts()`: This is a new function that takes the previous block and a potential new bytecode contract as its arguments. It updates the number of contracts, the collection of contracts, and the data storage of contracts on the block. If a new contract is passed in, the contract is translated via the virtual machine, added to the collection of contracts, and the contract counter is incremented.
- `constructor()`: This is an overridden function from the `Block` class. It runs the parent constructor and then runs `updateContracts()`.
- `rerun()`: This is an overridden function from the `Block` class. It runs `updateContracts()` and then runs the parent function.

5.3 VmClient

The virtual machine's `VmClient` class is an extension of the `Client` class from SpartanGold, with the following modifications:

- `postContractDeployTransaction()`: This is a new function that resembles `postTransaction()` from the `Client` class. It takes a list of outputs, a fixed fee and a list of data elements. It runs `updateContracts()` from the `VmBlock` class, passing in the previous block hash and the bytecode file retrieved from the data elements as its arguments. Returns a generic transaction updated with information about the added bytecode file.
- `postContractCallTransaction()`: This is a new function that resembles `postTransaction()` from the `Client` class. It takes a list of outputs, a fixed fee and a list of data elements. It retrieves from the block the bytecode associated

with the ID value specified in the data elements and evaluates the contract using the virtual machine, providing gas and any arguments needed to successfully run the file. The relevant output and storage lists are updated depending on the return values of the specific contract.

CHAPTER 6

Validation

In this section, I will demonstrate some of the functionalities of the SpartanGold Virtual Machine via some example bytecode files.

6.1 Timestamp-based SpartanGold Transfer

In this example, we have three clients: Alice, Bob, and Charlie. Alice wishes to transfer 100 SpartanGold to either Bob or Charlie, depending on the current timestamp value on the block.

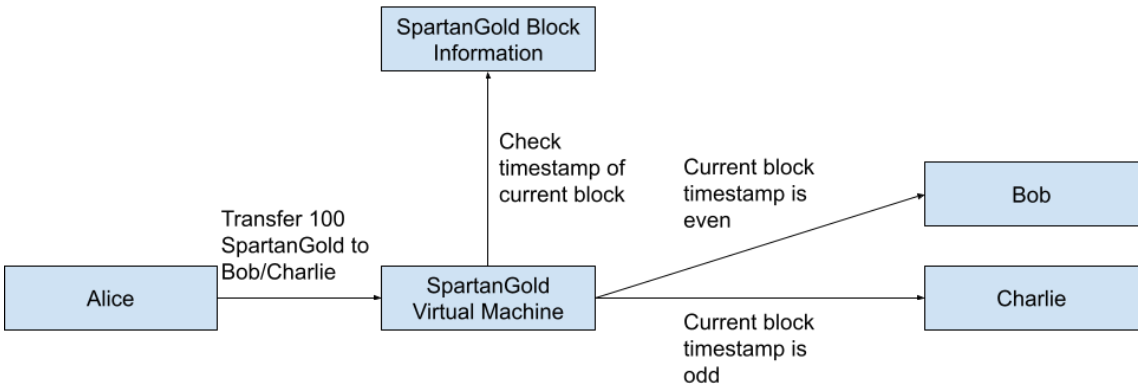

Figure 8: Execution Sequence for Timestamp-based SpartanGold Transfer

This is an important use case because it demonstrates that we are able to write a simple smart contract that runs on the virtual machine, transfers SpartanGold,

and depends on the state of the current block. The bytecode and console snippets below show what the code looks like, as well as the results from running it in each circumstance.

```
<Buffer b0 60 00 52 b0 60 01 52 b0 60 02 52 42 60 02 06 60 14 57 60 00 51 60 17 56 5b 60 01 51 5b f3>
```

Figure 9: Bytecode

```
Initial balances:
Alice: Showing balances:
    /T9FW6T/FRkbvph5ipaQd/cyv0LMryVLGmpGI8DgO+A=: 10000
    IcKUeJpwToOkFqakdEKybGAVCssYeClJn8meJmJqIw8=: 500
    yvacB5zQEf8M9JuXQouFeIJTHV84zvUvgBu7gf5MVEw=: 500
    MtZSEXqjZiOztDAxROBjZIStZ9+dbNvgy6yWFYkJbc4=: 400
    QqBPVg2YOErY95klI2IMkU+gWBniFF5UZ6uHb8uMq4A=: 300
```

Figure 10: Initial Balance: Even Timestamp

```
----------
Deployed contract with ID 1
----------
Last Block: {"chainLength":1,"timestamp":1669429449538,"transactions":[],"pre
vBlockHash":"124438933d01bf7209473377e17698f023c620009a1c0eb1ec25474c33c6d668
","proof":15357,"rewardAddr":"QqBPVg2YOErY95klI2IMkU+gWBniFF5UZ6uHb8uMq4A="}
----------
Bob address: IcKUeJpwToOkFqakdEKybGAVCssYeClJn8meJmJqIw8=, Charlie address: y
vacB5zQEf8M9JuXQouFeIJTHV84zvUvgBu7gf5MVEw=Block Timestamp 1669429449538
Recipient: IcKUeJpwToOkFqakdEKybGAVCssYeClJn8meJmJqIw8=
----------
```

Figure 11: Running Contract: Even Timestamp

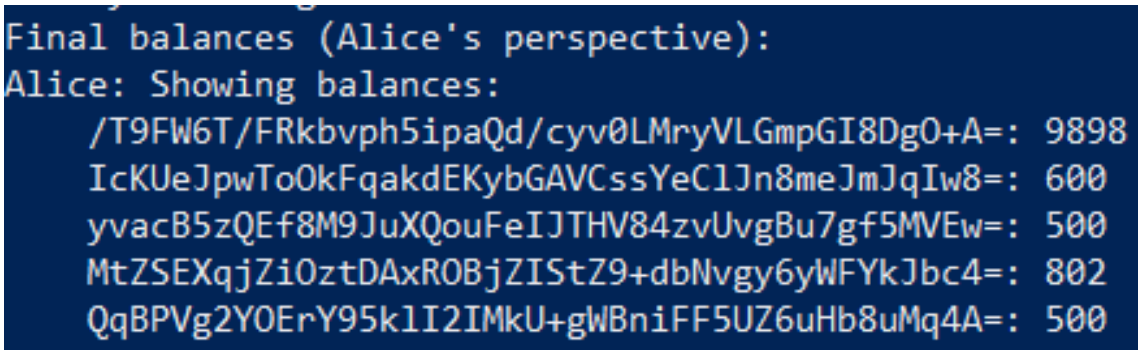

Figure 12: Final Balance: Even Timestamp

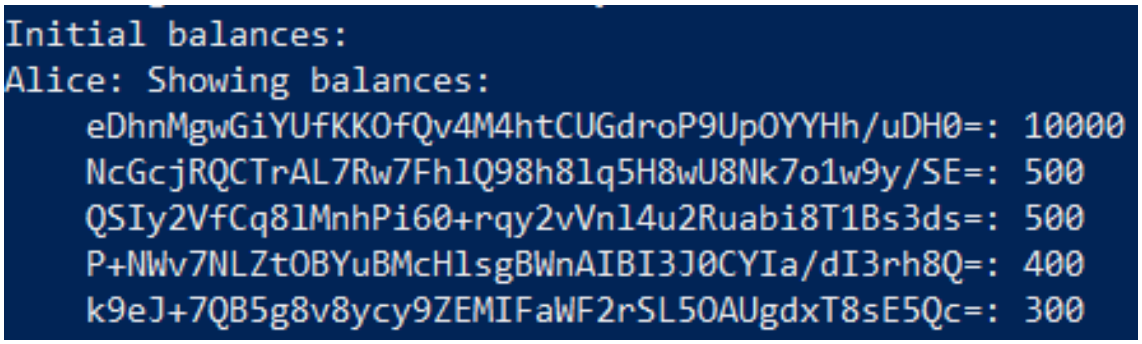

Figure 13: Initial Balance: Odd Timestamp

Figure 14: Running Contract: Odd Timestamp

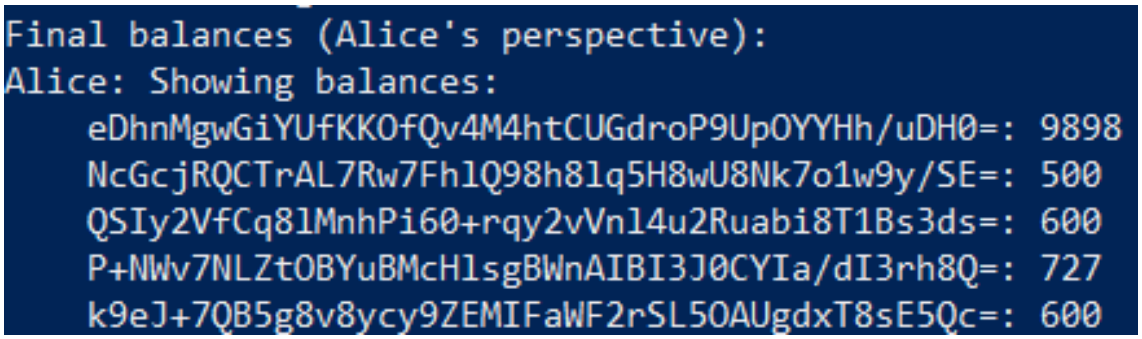

Figure 15: Final Balance: Odd Timestamp

6.2 Minimum Ethereum Token

In this example, we have JavaScript code representing a simplified token. This token includes all minimum functionality needed to qualify as a exchangeable token, with a structure to keep track of clients' balances, a constructor to initialize the token supply, and a transfer function to move tokens between clients.

```
contract MyToken {
  mapping (address => uint256) public balanceOf;

  constructor(uint256 initialSupply) {
    balanceOf[msg.sender] = initialSupply;
  }

  function transfer(address to,
        uint256 value) public returns (bool) {

    require(balanceOf[msg.sender] >= value);
    require(balanceOf[to] + value >= balanceOf[to]);

    balanceOf[msg.sender] -= value;
    balanceOf[to] += value;

    return true;
  }
}
```

Figure 16: Original JavaScript Code

This is an important use case because it demonstrates that we can define alternate tokens to be used as an alternate cryptocurrency or as a proxy for obtaining established cryptocurrencies such as SpartanGold, or even Ethereum. The bytecode and console snippets below show what the code looks like as bytecode, as well as the final status of the token representation when the bytecode is run.

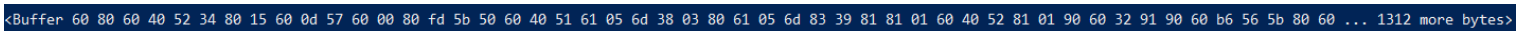

Figure 17: Bytecode

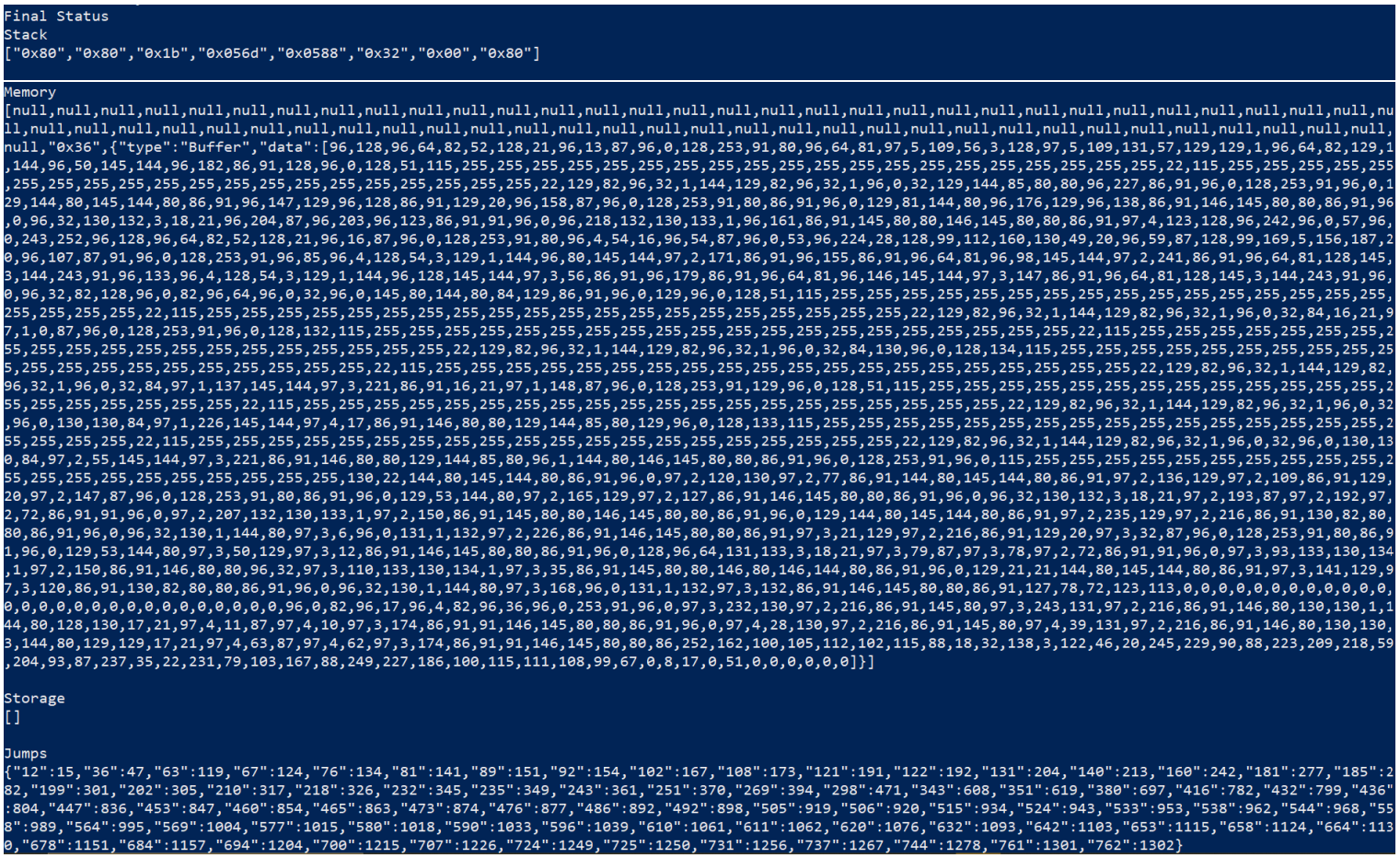

Figure 18: Minimum Token Final Status

CHAPTER 7

Conclusion

To conclude, virtual machines are important in understanding the blockchain and cryptocurrencies, but they are often opaque in how they function under the hood. By implementing the simplified SpartanGold Virtual Machine within the blockchain learning tool SpartanGold, anyone who wishes to learn more about virtual machines' role in the blockchain can do so with a token designed for learning and that has all the core features of widely used cryptocurrencies. As proof of this functionality, I demonstrated how a simple contract can be written and run, as well as how to represent a minimum Ethereum token, both with the SpartanGold Virtual Machine.

7.1 Future Work

In the future, there are a lot of different ways the SpartanGold Virtual Machine can be expanded. Below are just a few ideas for future extensions of the project.

7.1.1 Opcodes

While many opcodes, especially those listed in the Ethereum Yellow Paper, have been implemented, there are many more that have not been integrated into the SpartanGold Virtual Machine. A future direction could be implementing more opcodes from the Ethereum Yellow Paper or from other scripts such as Bitcoin Script, which would allow for different contracts to be run that use those newly implemented opcodes in their instruction sequences.

7.1.2 Contracts

Even without implementing other opcodes, the existing set of opcodes can be used to create different contracts that do interesting things. A future direction could be figuring out other contracts that can be implemented with the current instruction set in the SpartanGold Virtual Machine and that also does interesting things in regards to transferring tokens between users.

7.1.3 Tokens

Similar to contracts, the existing set of opcodes can be used to create different tokens that do interesting things. A future direction could be creating other tokens, such as the ERC721 token [10] in Ethereum, using the current instruction set available in the SpartanGold Virtual Machine.

LIST OF REFERENCES

[1] G. Srivastava, S. Dhar, A. D. Dwivedi, and J. Crichigno, ''Blockchain education,'' in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, 2019, pp. 1--5.

[2] C. Pollett, T. H. Austin, K. Potika, J. Rietz, and P. Pardeshi, ''Tontinecoin: Survivor-based proof-of-stake,'' *Peer-to-Peer Networking and Applications*, vol. 15, no. 2, pp. 988--1007, 2022.

[3] S. Basu, K. Basu, and T. H. Austin, ''Crowdfunding non-fungible tokens on the blockchain,'' *Silicon Valley Cybersecurity Conference*, pp. 109--125, 2022.

[4] ''Script.'' [Online]. Available: https://en.bitcoin.it/wiki/Script

[5] G. Wood, ''Ethereum: A secure decentralised generalised transaction ledger,'' *Ethereum project yellow paper*, vol. 151, pp. 1--32, 2014.

[6] J. Ellul and G. J. Pace, ''Alkylvm: A virtual machine for smart contract blockchain connected internet of things,'' in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1--4.

[7] D. Khoury, E. F. Kfoury, A. Kassem, and H. Harb, ''Decentralized voting platform based on ethereum blockchain,'' in *2018 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*, 2018, pp. 1--6.

[8] Puneet, A. Chaudhary, N. Chauhan, and A. Kumar, ''Decentralized voting platform based on ethereum blockchain,'' in *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2021, pp. 1--4.

[9] M. Westerkamp, F. Victor, and A. Küpper, ''Blockchain-based supply chain traceability: Token recipes model manufacturing processes,'' in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1595--1602.

[10] W. Entriken, D. Shirley, J. Evans, and N. Sachs, ''Eip-721: Non-fungible token standard,'' Jan 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721