



# Integrating Smart Contracts into the Bazo Blockchain

## Bachelor Thesis

Spring Term 2018

Department of Computer Science  
University of Applied Sciences Rapperswil

Authors: Ennio Meier, Marco Steiner  
Advisor: Prof. Dr. Thomas Bocek  
External Co-Examiner: Sven Stucki  
Internal Co-Examiner: Prof. Dr. Andreas Steffen

## Abstract

This thesis focuses on the integration of smart contracts into the Bazo Blockchain with the goal of creating a platform for decentralized applications. Smart contracts are programs that are stored on a blockchain and can be triggered by transactions. Smart contracts offer opportunities in automation and bring along advantages provided by blockchains such as immutability, public visibility of transactions and decentralization. As blockchains are trust-less protocols, it is guaranteed that smart contracts are executed as intended. The integration of smart contracts was solved by implementing a virtual machine that executes the instructions sent by a transaction. This stack based virtual machine uses byte arrays as base type, which enables the virtual machine to deal with numbers of arbitrary length. Working with elements of arbitrary size requires taking the length of the elements into account, when calculating the gas cost, in addition to the base cost of the instruction type. Furthermore, the virtual machine was embedded within the mining application, which required to alter the blockchain protocol and the mining application. As a result, smart contracts can be deployed and transactions that call functions of smart contracts can be executed. Calling a smart contract function leads to the execution of the contract in the virtual machine and persisting the result in the blockchain. The Bazo Blockchain continues to be a research project, this means it's not ready for production use due to complex setup and handling. Follow-up theses could simplify the development of smart contracts for the Bazo Blockchain by creating a high-level programming language that can be compiled to Bazo virtual machine instructions.

# Management Summary

## Integrating Smart Contracts into the Bazo Blockchain

<b>Introduction</b>	The Bazo Blockchain is a blockchain to test diverse mechanisms and algorithms. In the current version mechanisms to run it on mobile devices and Proof of Stake are integrated. It was only possible to transfer Bazo coins before this thesis. The idea of this work was to enhance the Bazo Blockchain with smart contracts.
<b>Solution</b>	The integration of smart contracts consisted of two sub goals. First a virtual machine had to be implemented which allows to use byte values as instructions. Therefore, a smart contract can be written as a set of bytes and stored on the blockchain with minimal memory usage. The second sub goal was the integration of the virtual machine into the miner, the application responsible for processing the transactions. After that, the code has been refactored and improved.
<b>Findings</b>	The project was evaluated on functionality and performance. With the creation of a tokenization contract the functionality was tested. In order to evaluate the performance, a modular exponentiation contract was created and compared with an implementation of modular exponentiation written in Go. The tests showed that the required functionality has been implemented. The performance test showed that the resource efficiency of the virtual machine could be increased.
<b>Outlook</b>	Since writing contracts in instructions of the virtual machine is rather time consuming and complicated to understand, a compiler could be built which compiles a more easy to read high level language into the byte code of the virtual machine.

## Acknowledgements

In this section we would like to thank Prof. Dr. Thomas Bocek for his assistance and inputs throughout. We were continuously impressed about his enthusiasm for Blockchain technology and his knowledge about it. There were many occasions where we were extraordinarily grateful to be able to contribute to the Bazo Blockchain. Building your own virtual machine really brings great insight into how a computer works on a lower level or how some of the programming languages are platform independent. We are convinced having worked on such a new and possibly groundbreaking technology has great benefits for our future careers.

We would also like to thank the other students for their work on the Bazo Blockchain. It has been an educative experience to work on such a fast-growing project that takes many different aspects into account.

# Contents

<b>1. Introduction</b>	<b>8</b>
1.1. Motivation . . . . .	9
1.2. Description of Work . . . . .	9
<b>2. Background and Related Work</b>	<b>10</b>
2.1. Background . . . . .	10
2.1.1. Blockchain . . . . .	10
2.1.2. Smart Contracts . . . . .	10
2.1.3. Transactions . . . . .	10
2.1.4. Virtual Machine . . . . .	11
2.2. Related Work . . . . .	11
2.2.1. How Bazo Works - an Overview . . . . .	12
2.2.2. Previous Work . . . . .	14
2.2.3. Similar Projects . . . . .	14
<b>3. Design</b>	<b>16</b>
3.1. Virtual Machine . . . . .	16
3.1.1. Types of Virtual Machines . . . . .	17
3.1.2. Guidelines . . . . .	18
3.1.3. Notable Design Aspects . . . . .	18
3.2. Contract Deployment . . . . .	19
3.3. Execution of a Contract Method . . . . .	19
3.4. VM Integration . . . . .	20
3.4.1. Transaction Types of the Miner . . . . .	20
3.4.2. Accounts . . . . .	21
3.4.3. Execution Context . . . . .	22
3.5. Parser . . . . .	23
3.5.1. «Enhanced Bazo Byte Code» . . . . .	23
3.5.2. Compile Process . . . . .	23
3.6. Smart Contracts . . . . .	24
3.6.1. Coding Smart Contracts . . . . .	24
3.7. Fee . . . . .	26
<b>4. Implementation</b>	<b>27</b>
4.1. Software Architecture . . . . .	27
4.2. Protocol . . . . .	28
4.2.1. Encoding . . . . .	29
4.2.2. Decoding . . . . .	29
4.3. Miner . . . . .	30
4.3.1. Constructors . . . . .	30
4.3.2. VM Entry Point . . . . .	30

4.4. Virtual Machine . . . . .	31
4.4.1. Stack . . . . .	31
4.4.2. VM Instruction Cycle . . . . .	31
4.4.3. Error Handling . . . . .	33
4.4.4. Gas Calculation . . . . .	33
4.4.5. Trace Function . . . . .	34
4.5. Opcodes . . . . .	35
4.5.1. Arithmetic . . . . .	35
4.5.2. Bool Operations . . . . .	35
4.5.3. Comparison Operators . . . . .	36
4.5.4. Control Flow Operations . . . . .	36
4.5.5. Data Structures . . . . .	37
4.6. Context . . . . .	37
4.6.1. Data from a Transaction . . . . .	38
4.6.2. Data from the Receiver Account . . . . .	38
4.7. Parser . . . . .	39
4.7.1. Tokens File . . . . .	39
4.7.2. Parser File . . . . .	40
4.8. Testing . . . . .	42
4.8.1. Unit Testing . . . . .	42
4.8.2. Fuzz Testing . . . . .	43
4.8.3. Integration Testing . . . . .	43
<b>5. Evaluation</b>	<b>44</b>
5.1. Tokenization Contract . . . . .	44
5.1.1. Results . . . . .	44
5.2. Benchmarking Contract . . . . .	44
5.2.1. Results . . . . .	45
<b>6. Conclusion</b>	<b>48</b>
6.1. Future Work . . . . .	49
<b>A. Installation Guidelines</b>	<b>53</b>
<b>B. Opcodes</b>	<b>56</b>
<b>C. Tokenization Contract</b>	<b>59</b>
<b>D. Definition of Task</b>	<b>61</b>

## Glossary

instruction set	The instruction set is the program or contract itself. It is an array of opcodes.
opcode	Short for operation code. Operation codes are byte values according to which the vm performs certain operations.
P2P	Peer to peer package, used to communicate between miners and clients.
panic	Terminal exception in Go.
Proof of Stake	Consensus mechanism for blockchain.
Proof of Work	Consensus mechanism for blockchain.

# 1. Introduction

<b>The Bazo Blockchain</b>	The Bazo Blockchain was started as a research project at the University of Zürich in cooperation with a Financial Service Provider. The first goal of Bazo as a research project was, to provide consumer bonus coins based on blockchain technology for the financial service provider.
<b>Benefits of a Blockchain Based Solution</b>	The most important benefit of a blockchain based solution is the reduction of administrative work for the financial service provider and the businesses taking part in the coin redemption program. This is because a business contract between financial service provider and business has to be created for every product or service before a consumer can buy them with the bonus points.
<b>Intentions</b>	Since the consumer bonus coins become coins of a cryptocurrency the intention is to get businesses and customers to exchange goods directly with bonus coins without the financial service provider as a third party. As mentioned above this is not yet the case, the customers can redeem their coins only via the financial service provider who previously created contracts with the businesses to provide their services for the customers.
<b>Vision</b>	With the new solution the financial service provider would only take on the role as exchange point of Bazo coins against fiat currency. Businesses and customers could use the consumer bonus coins in the same way as an actual currency.
<b>Evolution of Bazo</b>	Although the research project with the financial service provider continues, the requirements of this thesis are independent from the requirements of the financial service provider. In the spirit of the name Bazo, which means base in Esperanto, the vision for Bazo is to become a foundation for decentralized applications.
<b>Scope of Work</b>	As smart contracts are an important part of the foundation on which such distributed applications are built on, this thesis provides the environment on which smart contracts can be run on by building a virtual machine (VM) and integrating it into the miner.



## 1.1. Motivation

<b>Context</b>	As a blockchain in its simplest form solves the problem of keeping track on who owns what and making transactions of such goods, there is still a huge chance for automation and reliability. In this stage the blockchain is basically just a ledger but with the integration of smart contracts, one transaction can trigger a multitude of different actions which otherwise had to be executed manually. Triggering the execution of a smart contract is much faster and more reliable than a human reacting to a received transaction towards one account. The limitation here is of course that there has to be a set of transactions whichs can be dealt with in the same manner to make automation work but this is usually the case.
<b>Chance</b>	As transaction speed and cost approach nearly zero compared to traditional payment systems like paying a bill at the post office, it is more and more feasible to make even very small transactions. This could be used to pay content creators like journalists, artists or musicians fairer by using smart contracts creating a small transaction each time a video or article is viewed. This is just an example and having provided the base, of course many other things can be built upon such a system.
<b>Solution</b>	Since the participants of the blockchain are free to exchange coins for goods or services, a smart contract facilitates these value transactions as it states in a programming language according to what terms a party is ready to do business with other parties and also makes such transactions faster as there is no involvement of humans in order to verify or execute the transaction. One can be sure that a valid call of the contract results in its execution according to the terms described in it.
<b>Summary</b>	In the context of automation, speed and transparency, smart contracts provide a huge benefit to the blockchain.

## 1.2. Description of Work

This thesis is concerned with the integration of smart contracts into the Bazo Blockchain. The goal of contract integration contained multiple subgoals. First a virtual machine had to be built which is able to execute the code later stored on the blockchain. The second subgoal was to implement the VM into the miner of the Bazo Blockchain because contract execution should be part of transaction verification.

## 2. Background and Related Work

The section *Background* introduces the tools and technologies used in the Bazo Blockchain. In the section *Related Work* the theses of other students who worked on the Bazo Blockchain and some independent projects are described.

### 2.1. Background

#### 2.1.1. Blockchain

A blockchain is a distributed database. It consists of blocks of data chained together by hash functions. The data inside these blocks are transactions. In order to make the transactions non-repudiable digital signatures are used. The transactions are validated by a miner. The miner validates the signatures and checks if the assets transmitted by the transaction, usually tokens or coins representing a real monetary value, actually exists on the account. Since the Bazo Blockchain is account based, it also updates the balance of the account according to the transmitted value. Previous theses in the context of Bazo dive deeper into cryptocurrencies and blockchains. [16] [5]

#### 2.1.2. Smart Contracts

A smart contract is basically an agreement or contract written in computer code, saved in a transaction and therefore distributed over the whole network. The smart contract can then be called by another transaction and is executed by a VM which is part of the miner.

#### 2.1.3. Transactions

In the context of data base management systems a transaction is a unit of work performed within the system. [13] This definition is also applicable for blockchains. As later explained there are multiple transaction types and only some are used to send coins. Others are used mostly to change the system state or create a new account.

### 2.1.4. Virtual Machine

A virtual machine is an abstraction layer. A VM abstracts a program or even a whole operating system from the hardware it is actually run on to make it portable.

## 2.2. Related Work

At the time of writing the thesis, the [Bazo Blockchain Github organization](#) contains the software systems as described in Figure 2.1. There are also some smaller tools which are not further described. Figure 2.1 shows the Bazo Blockchain and its most important software systems. It also describes how the systems interact with each other and with the out world. It shows multiple miner instances to represent the blockchain and to describe how the miners interact with each other.

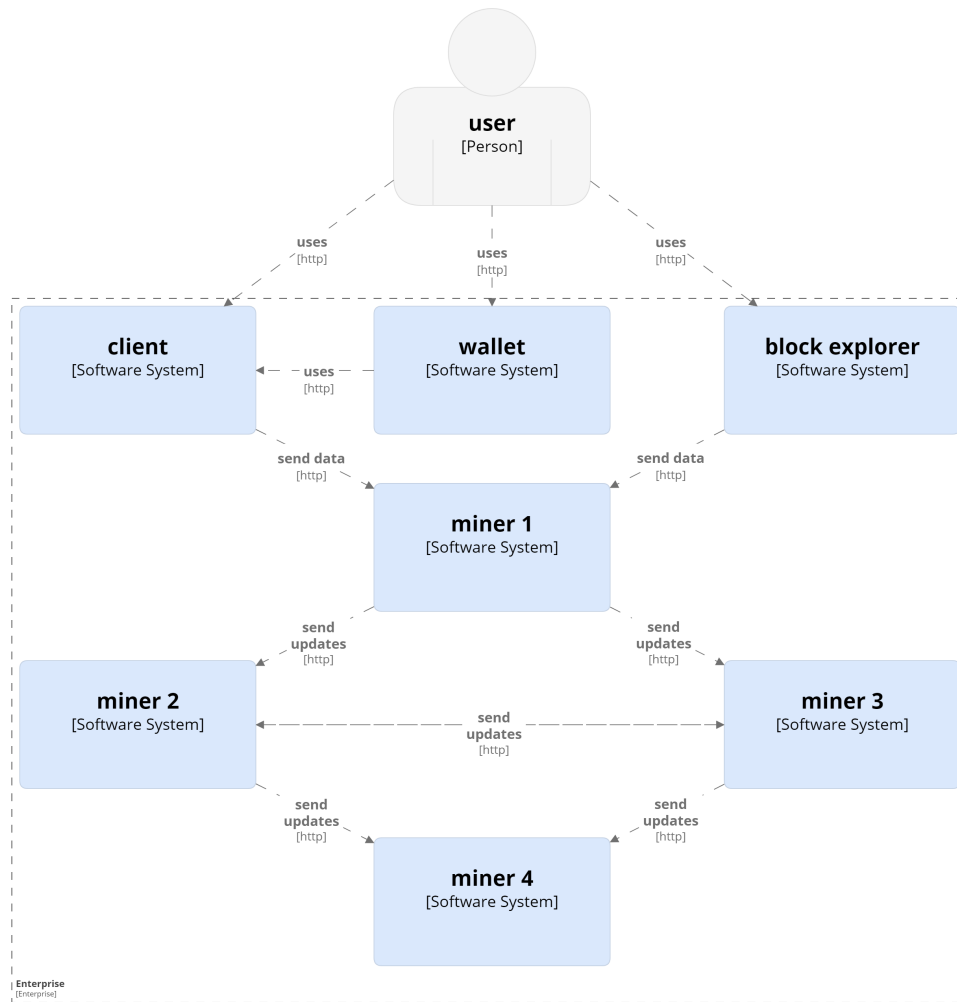


Figure 2.1.: Bazo Blockchain system context diagram

<b>Miner</b>	The miner is at the heart of the blockchain. It validates transactions, keeps the ledger up to date, executes smart contracts via the VM and broadcasts new transactions or blocks to other miners.
<b>Wallet</b>	The wallet allows users to operate on the blockchain. It also stores the public/private key pair. [2]
<b>Client</b>	The client is used to communicate with the miner by creating and sending new transactions.
<b>Block Explorer</b>	The block explorer is used to view transactions. It basically visualizes the blockchain.
<b>Miscellaneous</b>	There are also some smaller tools, like keypairgen but they are not further discussed here. The organization on Github is accessible under the following link: <a href="https://github.com/bazo-blockchain">https://github.com/bazo-blockchain</a> .

### 2.2.1. How Bazo Works - an Overview

In Figure 2.2 one of the miners is expanded, which makes it possible to see its packages and their relationships with other components.

As shown in Figure 2.2 the transactions sent by the user over the Web API are received and processed by the *P2P* package. If it receives a new transaction it will add it to the unprocessed transaction pool via the *Storage* package.

The miner will continuously fetch transactions from the unprocessed transactions pool, verify their signature and their validity, add the transaction hash to the current block and save the transaction in the processed transaction pool. If the transaction calls a contract, the miner will call the VM and it will execute the called contract function. After that, the miner will change its state according to the results of the VM.

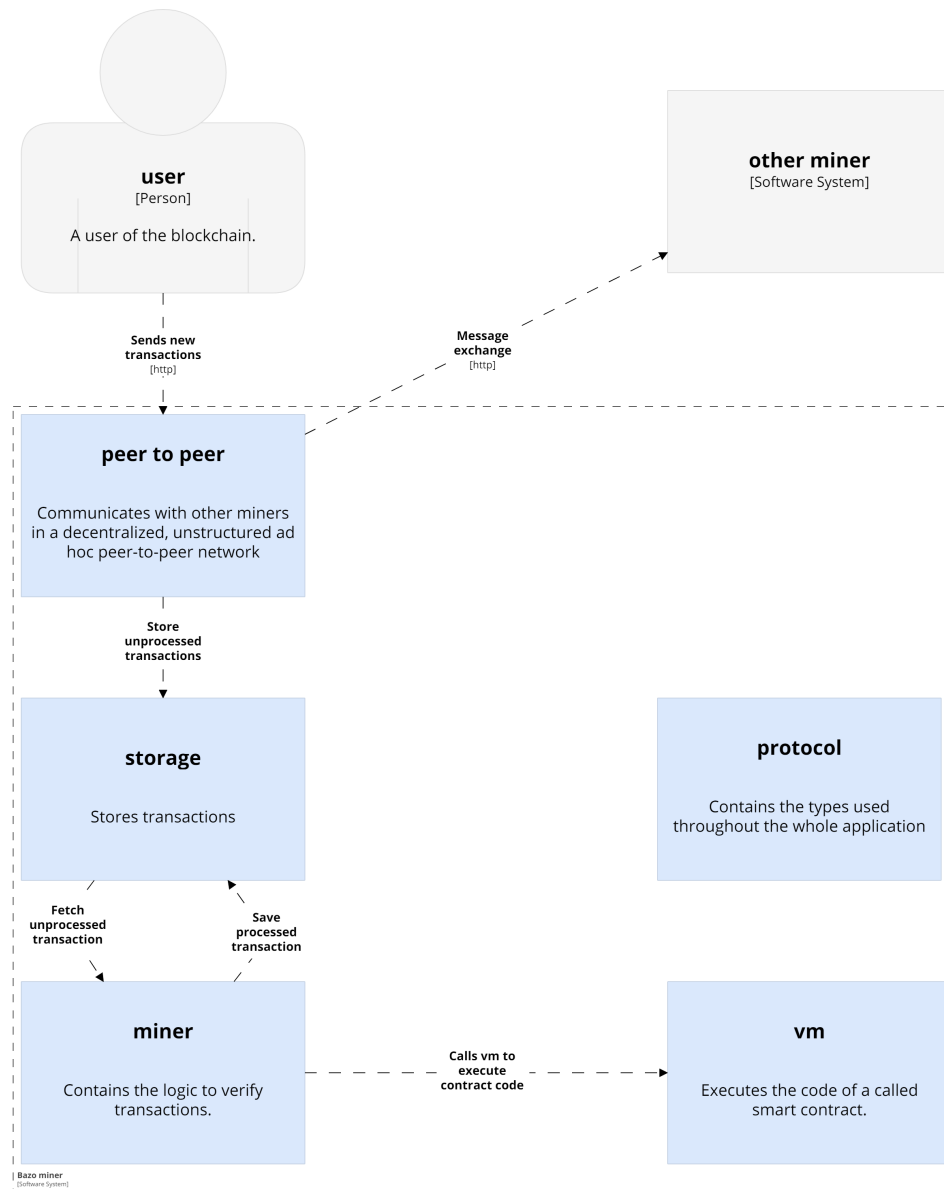


Figure 2.2.: Bazo Blockchain container diagram

### 2.2.2. Previous Work

As the Bazo Blockchain was started in 2017 there have been several theses. Within these theses different aspects of the blockchain have been implemented.

**Bazo – A Cryptocurrency from Scratch [16]** was written by Livio Sgier. He implemented the miner during this work. He partitioned the software into the *P2P*, *Storage*, *Miner* and *Protocol* package.

**A Progressive Web App (PWA)-based Mobile Wallet for Bazo [2]** was written by Jan von der Assen. His work was a proof of concept for a portable wallet which allows for mobile payments in a sandbox environment. In Figure 2.1 the result of his work is represented as the wallet software system.

**A Blockchain Explorer for Bazo [4]** was written by Luc Boillat. The Bazo Blockchain Explorer allows users of the blockchain to inspect blockchain data through a graphical user interface, requiring no installation to use, since the application is available on the internet as a web app. [4]

**Proof of Stake for Bazo [3]** was written by Simon Bachmann. Simon Bachmann studied multiple algorithms and implementations for [Proof of Stake](#) and chose the best for the later implementation of PoS which was also part of the work.

**Design and Prototypical Implementation of a Mobile Light Client for the Bazo Blockchain [5]** was written by Marc-Alain Chételat. The blocks of the Bazo Blockchain only contain the transaction headers but stores the transactions separately. During his work he implemented a light client which contains only the transaction headers and their hashes but not the actual transactions. This allows a client to also run on mobile devices. He also implemented a multi-signature mechanism which allows to verify transactions within three seconds. To do this a Bazo signature server is used to check if the accounts balance is high enough to send the transaction. If it does, the transaction is signed and therefore verified.

### 2.2.3. Similar Projects

Since the code of NEO and Ethereum is on Github and open source, their implementation has been studied and sometimes been implemented analogously.

<b>NEO</b>	NEO is a blockchain project «that utilizes blockchain technology and digital identity to digitize assets, to automate the management of digital assets using smart contracts, and to realize a smart economy with a distributed network.»[11] NEO utilizes a consensus mechanism called the Delegated Byzantine Fault Tolerance. NEO is implemented in C#. [14]
------------	---

**Ethereum**

The goal of Ethereum is to create a platform for the development of decentralized apps in order to create a «more globally accessible, more free, and more trustworthy Internet, an internet 3.0». [11] There are several implementations of the client such as go-ethereum (written in Go), cpp-ethereum (written in C++) and others. Ethereum's consensus mechanism is [Proof of Work](#) but a Proof of Stake algorithm is already being developed and likely to go live in 2018.

**What are the Differences to Bazo**

Both, Ethereum and NEO are public, permission-less smart contract platforms. At the time of writing, the Bazo Blockchain is a permissioned blockchain because of the initial requirements from the financial services provider. As for now, the Bazo Blockchain is just a research project. The goal is to become a public, permission-less platform for decentralized applications. To reach this goal and to be able to create and maintain a competitive blockchain, a dedicated team would be needed.

### 3. Design

As the term software design is generally defined in a broad way to also be suitable for larger systems first an elaboration on how the term design is used in this thesis. The word design itself is defined as: Do or plan (something) with a specific purpose in mind. [15] In previous theses on Bazo, the chapter design also provides an overview over the software. Therefore, it was decided to provide an overview over the implementation of the Bazo VM and the parser and to outline the most important purposes they are built for. This chapter also describes the goals, requirements, restrictions and an overview of the integration of the VM into the miner. Furthermore, the orientation taken while building the foundation for the creation, deployment and execution of smart contracts is explained.

#### 3.1. Virtual Machine

Figure 3.1 shows the VM execution cycle. It describes how the VM executes a smart contract. Afterwards, different types of VMs are explained and the most important guidelines and goals that were kept in mind throughout the building of the VM are described.

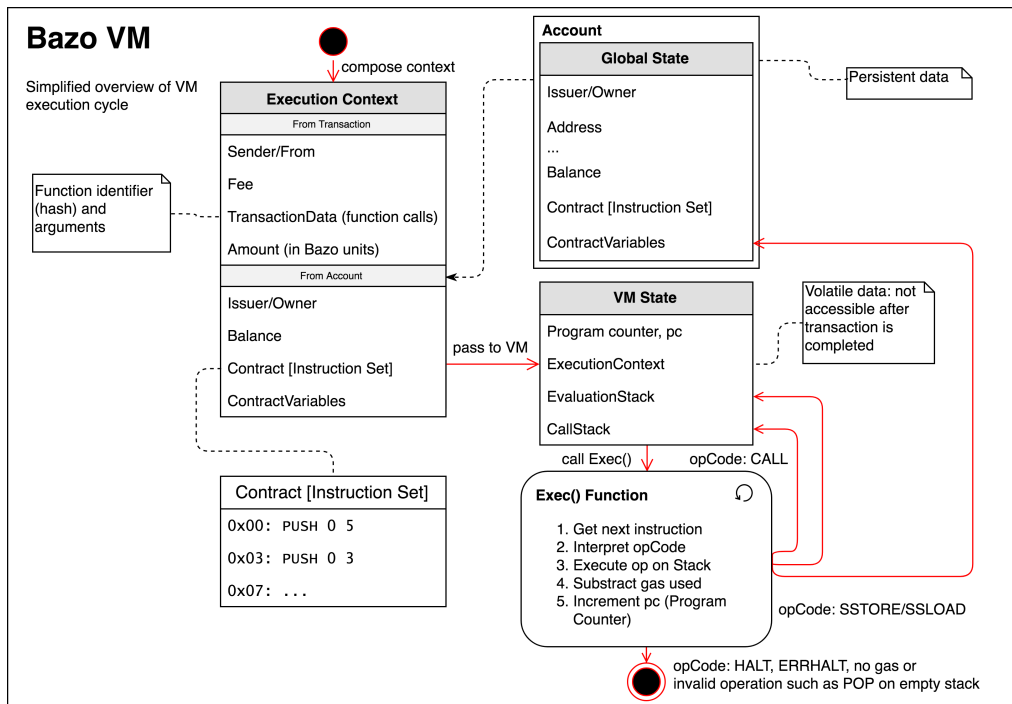


Figure 3.1.: Virtual machine execution cycle overview adapted from [18] and [6]



### 3.1.1. Types of Virtual Machines

There are two types of virtual machines. On the one hand, there are register based virtual machines. Examples of register based virtual machines are the Lua VM and the Dalvik VM. On the other, there are stack based virtual machines. The Java Virtual Machine and the .NET CLR are both stack based virtual machines. [17]

**Register based** The data structure of where the operands are stored is based on registers of the CPU, therefore the instructions need to contain the addresses (registers) of the operands. This leads to longer instructions. Figure 3.2 shows how adding two number works on a register based virtual machine. [17] The instruction is `ADD R1 R3 R2`.

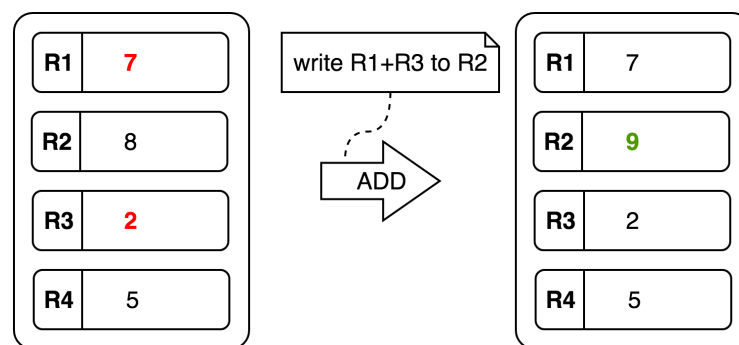


Figure 3.2.: Register based virtual machine

**Stack based** A stack based virtual machine is based on a LIFO (last in, first out) stack. Operations are carried out by popping and pushing back results on the stack. The main advantage is a stack pointer that implicitly addresses the operands, which means that no addresses are passed in instructions. The instructions set is longer since `POP` and `PUSH` instructions have to be included to retrieve and store the operands. [17] The instruction set to add two numbers as shown in figure 3.3 are:

---

1	<code>POP</code>
2	<code>POP</code>
3	<code>ADD</code>
4	<code>PUSH</code>

---

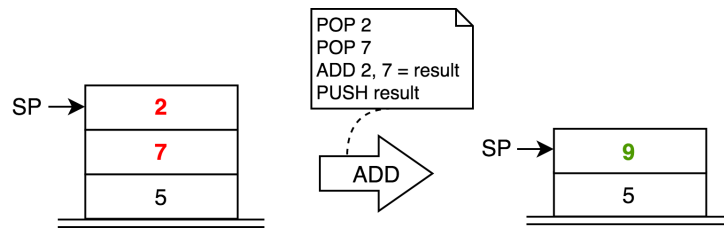


Figure 3.3.: Stack based virtual machine

**Design decision** Despite the register based virtual machine having advantages such as more possibilities for optimizations and having no overhead from pushing and popping, over a stack based virtual machine, we decided to implement a stack based virtual machine. This decision was most influenced by the implementation of related projects, namely Ethereum and NEO. Besides that, the implementation of a stack based virtual machine is simpler and much more resources are available online.

### 3.1.2. Guidelines

As the Bazo VM continues to be improved or extended by other students, it was paid attention to make the software readable, testable, changeable and extendable.

<b>Readability</b>	In order to make the code readable code reviews were performed that led to refactorings.
<b>Testability</b>	The code has been written using test driven development, which required writing testable code. In later stages of the project the tests became very useful and made necessary refactorings easier.
<b>Extendability and Changeability</b>	The code should be structured and decoupled enough to make necessary changes or extensions to the VM easy for following projects. As described before, having many tests makes the refactoring of the code easier.

### 3.1.3. Notable Design Aspects

<b>Fault Tolerance</b>	As the VM just executes the instructions it receives without any checks, it could enter an error state and <a href="#">panic</a> . It must not be possible to crash the miner with a malicious or erroneous contract.
<b>Informative Error Messages</b>	Throughout the implementation of the VM it was paid attention to write informative error messages, since that simplifies the development of a compiler, which is known to be a follow-up project. Furthermore, the debugging of smart contracts should be easy and streamlined as possible.

### 3.2. Contract Deployment

<b>State of the Art</b>	Other blockchains such as Ethereum allow to create and deploy a contract over an IDE. These contracts can be written in a high level language and they can be deployed automatically. In this work the foundation for smart contract deployment and execution has been laid out and designed accordingly. To make it as easy for the end user as mentioned above, a lot more work and resources would have to be invested.
<b>Current Contract Deployment</b>	Currently, contracts are account transactions and as Bazo is permissioned, have to be created with the root key pair. These transactions are added to the unprocessed transactions pool of the miner. When creating the account transaction, the contract and the initial values of the contract have to be provided. The miner then validates the transaction and creates a contract account.
<b>Restrictions</b>	This means that yet only developers can create new contracts.
<b>Rationale</b>	To make contract creation available for account owners, it needs to be possible to create a contract account with any valid key pair of a Bazo account. Also the <i>P2P</i> package would have to support the features implemented during this work. Both tasks by themselves would have exceeded the scope of this thesis.

### 3.3. Execution of a Contract Method

<b>State of the Art</b>	In other blockchains such as Ethereum, smart contracts can be called via the wallet or the Web API. The results can be checked via API, wallet or block explorer.
<b>Current Contract Execution</b>	In Bazo a transaction which calls a contract has to be added to the unprocessed transactions pool. The miner takes the transaction out of the pool and validates it. If the data field of the transaction is set, the miner will setup the VM context and execute the called function in the VM. After a successful execution, the changed contract variables are written back.
<b>Restrictions</b>	As the features are not yet implemented in the client nor the <i>P2P</i> package, it is not possible to send this kind of transaction via the Web API of the miner. It is also not yet possible to check the result, since smart contracts are not part of the block explorer.
<b>Rationale</b>	Implementing the features in the other applications as well would have exceeded the scope of this work.

### 3.4. VM Integration

The VM was integrated into the miner. This section describes the most important aspects of the already existing miner implementation which influenced the integration of the VM and the reasons for the adjustments made to the miner. The section also introduces the VM context which is used to provide data for the VM, gives an overview over the data the VM has access to and the rationale behind the solution with the context.

#### 3.4.1. Transaction Types of the Miner

In previous works concerning the miner, different transaction types were implemented or extended. Below it is described what the purpose of these transactions is, how they are extended in this work and why they were chosen for extension.

##### FundsTx

This type of transaction is used to transfer Bazo coins from one account to another and can be used by any externally owned account.<sup>[16]</sup> For this reason it was extended with a *Data* field to enable users to call smart contract functions.

---

```

1  type FundsTx struct {
2      Header byte
3      Amount uint64
4      Fee    uint64
5      TxCnt  uint32
6      From   [32]byte
7      To     [32]byte
8      Sig1   [64]byte
9      Sig2   [64]byte
10     Data    []byte
11 }

```

---

Figure 3.4.: Struct of the type FundsTx

**AccTx**

The account transaction is used to create a new account. As for now, this type of transaction is only allowed to root accounts, since the signature has to be signed with the private key of a root account. [16] It is important to note that since the miner distinguishes between the transaction types for the corresponding functionality by a switch case in most of its files. It would have been very time consuming to extend the miner by another transaction. Therefore and because most importantly a smart contract needs all fields of an account anyway, it was decided to add the required fields for the contract to the account transaction and leave them set to nil in a normal account transaction. The required fields are *Contract* which stores the code of the contract and *ContractVariables* which contains the state variables.

---

```

1  type AccTx struct {
2      Header      byte
3      Issuer      [32]byte
4      Fee         uint64
5      PubKey      [64]byte
6      Sig         [64]byte
7      Contract    []byte
8      ContractVariables []byteArray
9  }

```

---

Figure 3.5.: Struct of the type AccTx

**ConfigTx**

This type of transaction is used to change system parameters such as block size, block interval or minimum fee. No changes to the configuration transaction were necessary. For more information about this transaction type, see the thesis of Livio Sgier [16].

**StakeTx**

Since the Bazo Blockchain is in transition from [Proof of Work](#) to Proof of Stake there is a StakeTx transaction type. This type of transaction is being continuously improved in other works. For more information refer to the theses of Simon Bachmann [3] and Marc-Alain Chételat [5].

**3.4.2. Accounts**

An account is the result of processing an account transaction. Account specifically refers to the object created on the heap of the miner created by an account transaction. Accounts can be modified by a funds transaction. Figure 3.6 shows the struct for Account.

---

```

1  type ByteArray []byte
2
3  type Account struct {
4      Address      [64]byte // 64 Byte
5      Issuer       [32]byte // 32 Byte
6      Balance      uint64   // 8 Byte
7      TxCnt        uint32   // 4 Byte
8      IsStaking    bool      // 1 Byte
9      HashedSeed   [32]byte // 32 Byte
10     StakingBlockHeight uint32   // 4 Byte
11     Contract      []byte
12     ContractVariables []ByteArray
13 }

```

---

Figure 3.6.: Struct of the type Account

<b>Externally Owned Accounts</b>	Externally owned accounts are accounts that are owned by the person who has access to the combination of the public and private key. Having both, the person is able to execute transactions from the account. Externally owned accounts do not have an Issuer, a Contract or ContractVariables.
<b>Smart Contract Accounts</b>	Smart contract accounts are created and owned by externally owned accounts. The field Issuer is set and shows which externally owned account issued the account transaction of the contract account. A smart contract account contains its code in the contract field and if necessary contains its state in contract variables. Contract variables can be altered through contract functions.

### 3.4.3. Execution Context

<b>Consistency</b>	Since the blockchain is a distributed database and the VM needs to change the data of the blockchain eventually, consistency was a very important aspect of its integration. In order to keep the data of the miner consistent even after a potential failure of the VM, the context in which the VM is executed is created by the miner and passed to the VM. This is referred to as execution context.
<b>Content of the Context</b>	The execution context is composed with data coming from the transaction and the account. The execution context contains all the data needed to start the execution of the contract.
<b>Access to the Data</b>	Specific instructions that get the value from the context and put it on the top of the stack are provided. This for example allows the creation of a contract with functions only the contract issuer can call. In this case, the issuer field is accessed from the contract to verify ownership.

### 3.5. Parser

Since writing all contracts directly in byte code can be very complicated and time-consuming, it was decided to write a very basic parser. The goal of the parser was to make writing contracts easier by allowing the usage of labels and comments. Having labels resolves the problem of counting addresses when using control flow [opcodes](#) like `JMP` and `CALL` since they generally take an address as argument and change the program counter accordingly. The parser could be used as foundation for building a compiler which translates a contract written in a high-level language into »Bazo Byte Code«. For the rest of this thesis the code that can be interpreted by the parser is referred to as «Enhanced Bazo Byte Code» and the code the virtual machine operates on and which is stored on the blockchain as «Bazo Byte Code».

#### 3.5.1. «Enhanced Bazo Byte Code»

Figure 3.7 shows an example of a contract written in «Enhanced Bazo Byte Code». It is possible to have single line comments and inline comments as well, as seen in line 1 and line 5. Comments and empty lines are ignored by the parser. The first word in line is either an opcode or a label, which ends with a colon. Opcodes are optionally followed by arguments. It is predefined what types of arguments an opcode has. The `CALL` opcode in line 4 for instance takes a label and a byte as argument.

---

```

1  # This is a simple program which calls a function
2  PUSH 55780
3  PUSH 5
4  CALL addNums 2
5  HALT # stops execution
6
7  addNums:
8  LOAD 0
9  LOAD 1
10 ADD
11 RET

```

---

Figure 3.7.: Basic contract with function call written in «Enhanced Bazo Byte Code»

#### 3.5.2. Compile Process

First the parser splits the contract written in «Enhanced Bazo Byte Code» into tokens. A token consists of a token type and a value. The token type represents a kind of lexical unit e.g. opcode, label or a sequence of input characters. The token types are the symbols that are processed by the parser. [1] To get to the «Bazo Byte Code» the resulting set is iterated and the token is replaced by the corresponding byte value.

## 3.6. Smart Contracts

A smart contract consists of an ABI (application binary interface) and one or more callable functions. Smart contracts are deployed by a transaction (AccTx) and executed by a transaction (FundsTx). When someone wants to call a certain function in a smart contract, a special transaction to the public address of the smart contract is executed. The transaction contains an identifier in the data field, so the ABI can match the identifier with the function the caller wants to execute. Arguments passed to that function are also transmitted in that field. Since a transaction is processed simultaneously on all nodes of the network, all functions have to be deterministic.

### 3.6.1. Coding Smart Contracts

Smart contracts for the NEO blockchain can be developed in C#, Java, Kotlin, F# or Python. [14] There are different ways to create an Ethereum smart contract. There are different high-level programming languages that can be compiled to Ethereum byte code. Solidity has been developed by the Ethereum community and is the industry standard. Solidity is heavily inspired by JavaScript with the idea to attract JavaScript developers to write smart contracts. In this section a simple contract is written once in Solidity and once in «Enhanced Bazo Byte Code».

#### Sample Smart Contract in Solidity

---

```
1  contract MyFirstContract {
2      uint myData; //State variable
3
4      function set(uint x) public {
5          myData = x;
6      }
7
8      function add(uint amount) public {
9          myData += amount;
10     }
11
12     function sub(uint amount) public {
13         myData -= amount;
14     }
15
16     function get() public constant returns (uint) {
17         return myData;
18     }
19 }
```

---

This contract has the state variable myData. Calling the function set() with an uint parameter sets the variable. Calling the function add or sub allows the transaction sender to either add or



subtract a certain amount from that variable. In order to call a function a transaction must be executed.

### Sample smart contract in «Enhanced Bazo Byte Code»

Compiled Smart Contract with ABI would look like this:

---

```

1  CALLDATA      # Puts the arguments passed to the smart contract
2                # and the function hash on top of stack
3  # ABI:
4  DUP
5  PUSH set
6  EQ
7  JMPIF set
8
9  DUP
10 PUSH add
11 EQ
12 JMPIF add
13
14 DUP
15 PUSH sub
16 EQ
17 JMPIF sub
18
19 HALT
20
21 :set           # set function
22 SSTORE myData # stores the variable in ContractVariables
23 HALT
24
25 :add           # add function
26 POP
27 SLOAD myData  # loads the variable and puts a local copy on the stack
28 ADD
29 SSTORE myData # overwrites the variable in ContractVariables
30 HALT
31
32 :sub           # sub function
33 ...

```

---

### 3.7. Fee

<b>Incentive</b>	Running a node in the network carries costs and the node operators want to be compensated. It is also an incentive to get more people to mine blocks because they can earn money by doing so.
<b>Security Aspects of the Fee</b>	Moreover, the fee can be seen as a way to secure the network. The execution of a contract must be deterministic. Since the virtual machine is Turing complete, it is possible to create contracts, which stay in an endless loop causing the network to get stuck and not accepting new transactions. Subtracting gas with the processing of every instruction, the processing of the transaction comes to an end eventually because no more gas is available.
<b>Fees in other Blockchains</b>	In Ethereum and NEO this fee is called gas. Ethereum calculates the cost depending on which instruction is used and uses the smallest unit of its currency. NEO even separates the fee from the actual currency in order to keep the costs of a transaction stable even if the value of the coin itself rises. Bitcoin calculates the cost depending on the size of the transaction.
<b>Fees in Bazo</b>	The fee is expressed in the smallest unit of Bazo coins. The cost of execution vary depending on the complexity of the opcode and depending on the size of the processed elements, since it is possible to work with elements of arbitrary size. Separating fee from the coins of the blockchain was considered. See <a href="#">Chapter 6</a> for more information.

## 4. Implementation

This chapter focuses on explaining why things were implemented in a certain manner. In this chapter the trade-offs and decisions which were made while building the VM are documented. Since this should be useful for follow-up work, it was done in a way that makes it easier to decide if something needs to be changed or to help reach the same conclusion.

### 4.1. Software Architecture

Figure 4.1 shows the structure of the project on package-level and the dependencies among packages. The miner application was built with software engineering principles in mind. Particular attention was paid to modularity. This made it easy to integrate the virtual machine into the existing miner application. Furthermore, the following figure shows which packages were added and which were modified to integrate smart contracts into the Bazo Blockchain. As the miner and all other projects are written in Go, the new components are written in Go as well.

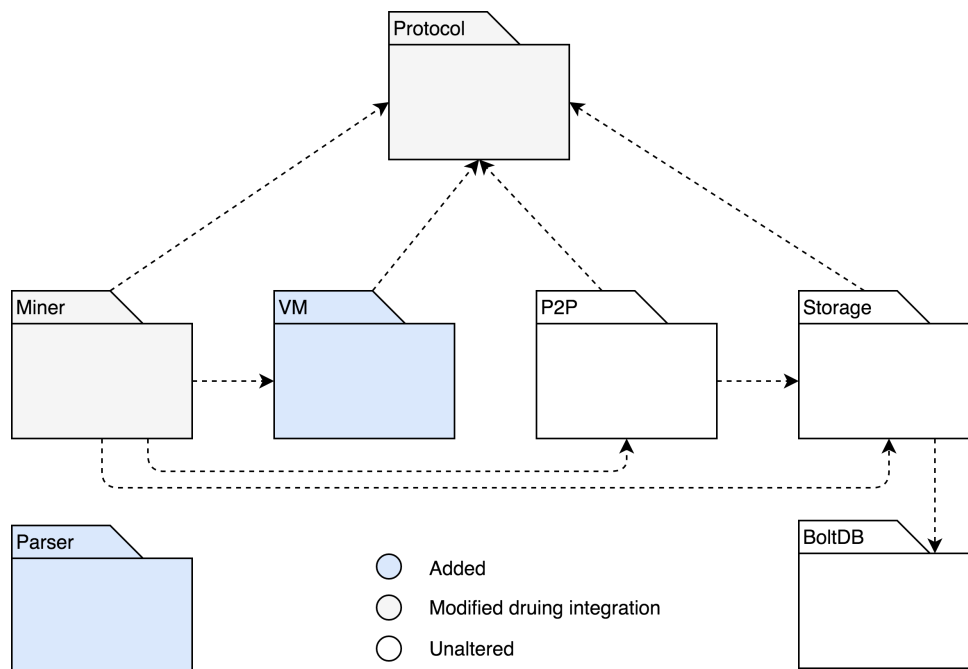


Figure 4.1.: UML Package Diagram

**Protocol** This package contains the building blocks for the Bazo Blockchain. In particular it contains the structure, encoding, decoding and hashing functions of all transaction types,

blocks and accounts. In addition a transaction interface is defined, allowing an abstract treatment of transactions. [16]

**Miner** The *Miner* package contains all mining related components. This includes the validation and consolidation of transactions into blocks, the calculation of the consensus mechanism and the building of the merkle tree. In addition to that, blockchain related tasks such as rollback operations and state changes are contained in this package. Access to *P2P* and *Storage* packages are needed in order to handle transactions over the network and signal storage-related operations. [16] To execute the virtual machine and access its results, the miner also needs access to the *VM* package.

**VM** This package contains the virtual machine itself and all its components, namely the evaluation stack, the call stack, the implementations of data structures and all available opcodes.

**P2P** All networking related operations are implemented in the *P2P* package [16]. Since for the integration of the virtual machine nothing had to be changed, the content of this package is not discussed any further.

**Storage** The storage package is concerned with memory-related tasks. The lower-level functionality was implemented with the external *BoltDB* package. Since entries are encoded before writing to storage and decoded when loaded [16], this package could be left unaltered. For that reason, this package is also not described further.

**BoltDB** This package contains the *BoltDB* package which is an external dependency. BoltDB is a simple, lightweight key/value base. [16]

**Parser** The *Parser* package contains the parser, which consists of a Tokenize and Parse function and tokens. These components are needed to compile «Enhanced Bazo Byte Code» to a byte code instruction set that can be interpreted by the virtual machine. This package is completely stand-alone and does not have any dependencies. The package is also kept in a separate repository.

## 4.2. Protocol

As mentioned in Chapter 3.4 the structs of Account, FundsTx and AccTx had to be adapted. The structs originally had a fixed length. The structs are transferred encoded between the individual components. The original encoding and decoding functions of those structs is based on the fixed length of the struct. Since the added fields are optional and have an arbitrary length, the encoding and decoding functions had to be redesigned. It was decided to use the gob package, which is a package from the Go standard library. The gob package manages streams of binary values between an encoder and decoder. A stream of gobs is self-describing. [8] The main disadvantage is that these functions are no longer platform independent. It was decided to use gob since all other software system which directly interact with the miner are implemented in Go and because it is the fastest standard encoding library in Go.

### 4.2.1. Encoding

The new encoding function for the `AccTx` struct is shown in figure 4.2. Switching to `gob` has lead to a better readability and to fewer lines of code. The type information is defined in line 2. The function `.Encode()` in line 12 makes sure that all type information data is sent before it is needed. [8] The encoding functions of the other structs, that were redesigned are very similar and therefore not further described.

---

```

1  func (tx *AccTx) Encode() (encodedTx []byte) {
2      encodeData := AccTx{
3          tx.Header,
4          tx.Issuer,
5          tx.Fee,
6          tx.PubKey,
7          tx.Sig,
8          tx.Contract,
9          tx.ContractVariables,
10     }
11     buffer := new(bytes.Buffer)
12     gob.NewEncoder(buffer).Encode(encodeData)
13     return buffer.Bytes()
14 }
```

---

Figure 4.2.: New gob based encoding function

### 4.2.2. Decoding

Figure 4.3 shows the decoding function of `AccTx`. The `decoder.Decode()` function in line 5 reads the next value from the input stream and stores it in the data represented by the `AccTx` interface. [8]

---

```

1  func (*AccTx) Decode(encodedTx []byte) *AccTx {
2      var decoded AccTx
3      buffer := bytes.NewBuffer(encodedTx)
4      decoder := gob.NewDecoder(buffer)
5      decoder.Decode(&decoded)
6      return &decoded
7  }
```

---

Figure 4.3.: New gob based decoding function

## 4.3. Miner

This section describes what changes had to be made to the miner to integrate the VM.

### 4.3.1. Constructors

*AccTx*s now have an optional field for contracts and contract variables. If the *AccTx* is meant to create an external account nil has to be passed for both parameters in the constructor. If the *FundsTx* is intended to transfer coins, the data field has to be nil.

### 4.3.2. VM Entry Point

The balance of a *FundsTx* is updated in `addFundsTx()` in the `block.go` file. Therefore, it was decided to use it as entry point for the VM. Figure 4.4 shows the function without parts that are irrelevant for the integration. Before the function checks whether the transaction calls a smart contract, it performs various checks, i.e. if the account exists or if the sender has a balance high enough to transfer the defined amount of Bazo coins. If all these checks are successful the function validates if the transaction is a valid smart contract call. If so, a new context object with the receiver account and the transaction is created as seen in line 8. As next step the virtual machine is initialized with the context object. In line 12 the `Exec()` function is called which starts the virtual machine. If the `Exec()` function returns with an error the execution of the `addFundsTx` is aborted and the context changes are not persisted. After the VM execution, the finalizing steps are carried out, such as updating the balances of both accounts and writing the block header to storage.

---

```

1  func addFundsTx(b *protocol.Block, tx *protocol.FundsTx) error {
2      ... // Various checks
3      // Check if transaction has data and receiver is a smart contract account
4      if tx.Data != nil && b.StateCopy[tx.To].Contract != nil {
5          context := protocol.NewContext(*b.StateCopy[tx.To], *tx)
6          virtualMachine := vm.NewVM(context)
7
8          // Check if vm execution run without error
9          if !virtualMachine.Exec(false) {
10             return errors.New(virtualMachine.GetErrorMsg())
11         }
12         //Update changes vm has made to the contract variables
13         context.PersistChanges()
14     }
15     ... // Finalization
16 }

```

---

Figure 4.4.: `addFundsTx` function

## 4.4. Virtual Machine

This chapter describes the most important parts of the VM implementation and the rationale behind it.

### 4.4.1. Stack

#### Maximum Stack Size

Facing the concern of excess memory usage of the contract on the miner, we decided to limit the stack size to 1MB which seems to be well above what the contracts will need. We neglected using the gas amount for maximum storage determination because it would be just a soft limit.

#### Data Structure

As underlying data structure of the stack, keeping in mind to keep the code of the VM as short and simple as possible and without unnecessary conversions it was decided to use a two dimensional byte array. We neglected using a simple array of bytes as data structure where elements with greater length than a byte are pushed using multiple indexes, as it is common when having no abstraction layer. We have also neglected the use of `big.Int` as the underlying data type which is used by Ethereum. `big.Int` can also be of arbitrary length. The reason for this decision was that converting signed operations to unsigned operations and optimizing the elimination of leading zeros posed a problem. Still, `big.Int` was used for many opcodes of the Bazo VM. A multidimensional byte array was chosen to achieve simplicity, readability and extendability of the code. The downside of smaller conversions from `big.Int` to byte array are accepted.

#### Data Type

It was important that the data type used for the underlying data structure of stack was of arbitrary length or at least very big because of cryptographic applications. Splitting up an element into multiple bytes so that it can be saved when using an array of fixed length as underlying data structure would have been a lot more complex and more error-prone to implement.

#### Pass by Reference

We neglected working with references because even though there are more elements created on the heap of the physical machine, it shouldn't make a difference considering the vast availability of resources on modern computers and our rather small contracts. In hindsight and considering the results of Chapter 5 working with references and using pass by reference might have improved the performance of the VM in the benchmark test.

### 4.4.2. VM Instruction Cycle

This section describes the `Exec()` function shown in figure 3.1. The instruction cycle can be described as an end-less loop with a switch statement that interprets the instructions one after

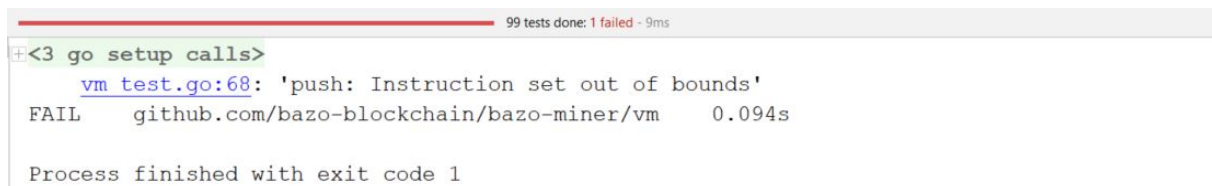
another and acts accordingly.

<b>Precondition</b>	The virtual machine is embedded into the miner. Precondition that the VM instruction cycle is started is that the transaction must be sent to a smart contract account and the transaction data field is not empty. If these preconditions can be fulfilled, the VM instruction cycle is started by the miner.
<b>Starting point</b>	At the starting point of the execution cycle the virtual machine contains a set of instructions, has access to the execution context and the program counter set to zero.
<b>Steps</b>	<p>The instruction cycle can be divided into three steps which are repeated over and over again until an invalid instruction occurs. Furthermore, the execution can be halted by an instruction or if the program counter is out of bounds. All this steps are handled in the <code>Exec()</code> function. These are the three steps:</p> <ol style="list-style-type: none"> <li><b>1. Fetch</b> The instruction the program counter points to is fetched. After it is fetched, the program counter is increased, which then points to the next element.</li> <li><b>2. Decode</b> In this step the instruction is matched with pre-defined opcodes, which can be interpreted by switch statement of the <code>Exec()</code> function. Within this section of the function, the costs for the instruction are subtracted.</li> <li><b>3. Execute</b> The instruction is executed according to the opcode. There are opcodes for arithmetic operations, e.g. <code>ADD</code>, <code>MOD</code>, <code>SUB</code>, for flow operations e.g. <code>JMP</code>, <code>CALL</code>, <code>RET</code> which are allowed to change the program counter and therefore move back and forward in the instruction set, cryptographic operations, such as <code>SHA3</code>, <code>CHECKSIG</code> and context operations, e.g. <code>ADDRESS</code>, <code>ISSUER</code>, <code>CALLER</code>, <code>CALLDATA</code> that can be used to push context data composed from the transaction and the receiver account to the stack and opcodes for storing and loading state variables <code>SSTORE</code>, <code>SLOAD</code>. All implemented opcodes are listed in table <a href="#">B.1</a>.</li> </ol>
<b>Return Value</b>	The <code>Exec()</code> function returns a boolean. If the return value is true the execution was successful and no error occurred. If the return value is false an error occurred.
<b>Persisting state variables</b>	Changes to the state variables are loaded and persisted explicitly. That means, when loading a state variable a copy is pushed to the stack, all changes are made to the copy. The changes are only persisted if the <code>Exec()</code> function returns with true. This way, there is no need to roll back if the contract could not be executed successfully.



#### 4.4.3. Error Handling

<b>Problem</b>	Since the virtual machine just processes one instruction after another and as the contracts are currently written in the opcodes directly, it is easily possible to make a mistake and write a contract which the VM can not execute.
<b>Example</b>	An example for this is an opcode which tells the VM to push six bytes on the stack, when there is only one byte left in the instruction set. This causes an error in the VM which could terminate the miner. This should neither be possible by accident nor by choice.
<b>Solution</b>	As a result, many guards in front of operations which could panic have been placed. This allows the graceful failure of the VM. As the error handling of Go works over separate return values they have been added to the functions.
<b>Implementation</b>	The message of this error object is later pushed on the stack of the VM after that the VM halts. Also to make the debugging less complicated, the name of the opcode in which the error occurred is prepended to the error message. Therefore, it is possible to determine what caused the error up to instruction type. An example error message is shown in Figure 4.5.
<b>Things Left Out</b>	It would also have been possible to provide the number of the instruction that failed but as the code of a contract gets larger and as the instructions are not enumerated, it becomes very time consuming to count all codes to determine which one failed. This seemed not to add value or make contract creation easier and therefore this has been neglected.



```
+<3 go setup calls>
  vm test.go:68: 'push: Instruction set out of bounds'
FAIL    github.com/bazo-blockchain/bazo-miner/vm    0.094s

Process finished with exit code 1
```

Figure 4.5.: Example of an error message.

#### 4.4.4. Gas Calculation

The goal is to calculate the price based on the instruction and based on the size of the elements used by the instruction. This was solved by adding a gas price and a gas factor to every opcode. The gas price shows the base price of the instruction. The gas price is subtracted before the operation is executed. The gas factor is a multiplier. The length of any element popped from the stack is divided by 64, rounded up and multiplied by the gas factor. The result of this equation

is the gas cost that is subtracted when elements are popped from the stack. This ensures that a sufficient amount of gas is available, before the actual manipulation is done. Figure 4.6 shows how the gasCost is calculated.

$$gasCost = \left( \frac{length_{element} + 63}{64} \right) \times gasFactor$$

Figure 4.6.: Gas cost equation

#### 4.4.5. Trace Function

The trace function is activated by passing true as a parameter to the `Exec()` function. The trace function prints the instruction that is processed, its parameters, the contents of the evaluation stack and the memory usage to the console. This is useful for debugging. Every opcode has a list of parameter types. This list shows, which parameters are taken from the instruction set. Available parameter types are **BYTES**, **BYTE**, **ADDR** and **LABEL**. The trace function loops through the parameter types list of the current opcode and treats every opcode as specified using a switch statement. Figure 4.7 shows an example trace output.

**BYTES** are variable length, the first byte that follows after the opcode in the instruction set defines how many bytes are pushed. Counting starts at zero, therefore the maximum is 256 bytes.

**BYTE** reads a single byte.

**ADDR** reads 32 bytes from the instruction set. This type is used to read addresses.

**LABEL** reads the next two bytes and treats them as an integer. This type is mainly used for control flow opcodes.

---

```

1 0000: push  [10] (bytes)
2      Stack: [[10]]
3      1 of max. 1000000 Bytes in use
4 .....
5 0003: push  [8] (bytes)
6      Stack: [[8] [10]]
7      2 of max. 1000000 Bytes in use
8 .....
9 0006: call  11 (label) [2] (byte)
10     Stack: []
11     0 of max. 1000000 Bytes in use
12 ...

```

---

Figure 4.7.: Trace function output

## 4.5. Opcodes

In this section the implementation of the opcodes, the rationale behind their implementation and the most important aspects when working with them are described.

### 4.5.1. Arithmetic

<b>Implementation</b>	Arithmetic opcodes use the arithmetic operations provided by <code>big.Int</code> . The two elements on top of the stack are popped, the operation is executed and the result is pushed back to the stack.
<b>Element Size</b>	It is not necessary to take care of the length of the elements, since <code>big.Int</code> allows for elements of arbitrary size.
<b>Signing Byte</b>	Important to note is that, these operations are always signed. Therefore when pushing a number the signing byte has to be provided in the first byte by setting it to zero or one. If this is not done, an error can occur or result in strange effects, because the first byte is used to describe the sign of the number.
<b>Rationale for <code>big.Int</code></b>	Concerning the implementation of arithmetic opcodes for the VM facing the need to provide operations for large elements it was decided to use <code>big.Int</code> . It was and neglected to use <code>Int64</code> in order to achieve the possibility to work with numbers of at least 256 byte. The downside of adjusting the gas calculation to take the element size into account was accepted.

### 4.5.2. Bool Operations

<b>Implementation</b>	To the category of bool operations belong <code>EQ</code> , <code>NEQ</code> , <code>LT</code> , <code>GT</code> , <code>LTE</code> and <code>GTE</code> . All of these opcodes use the the operations provided by <code>big.Int</code> and all of them work with signed numbers. Only <code>EQ</code> and <code>NEQ</code> work also with bytes.
<b>EQ and NEQ</b>	While writing the integration tests it turned out that the opcodes <code>EQ</code> and <code>NEQ</code> are oftentimes used for the comparison between function hashes in the ABI. Since function hashes are not signed numbers but just bytes it was decided to make these operations unsigned because the parsing of the signing byte would destroy the byte representation of the hash. As a result, signed numbers and bytes can be compared.
<b>Rationale</b>	As mentioned before the most prominent use case for <code>EQ</code> and <code>NEQ</code> is the comparison of function hashes. Therefore, they are unsigned. For the other opcodes an order has to exist and therefore the first byte of the elements is parsed to set the sign of the <code>big.Ints</code> on which the operation is performed on.

### 4.5.3. Comparison Operators

**Implementation** Bit operations are the two opcodes **SHIFTR** and **SHIFTL** for shifting bits to the right and the left. Same as for the arithmetic operations, this is based on the implementation of **big.int**.

**Unsigned** Important to note is that these operations are primarily thought to be used for bytes. They do not parse the leading byte in order to set the sign in **big.Int**. This means that when these operations are used on signed numbers the result could be that the signing byte becomes greater than zero or one and the number therefore is set into an invalid state. To avoid this the opcode has to be changed or reimplemented depending on the use case.

**Rationale** Concerning the implementation of opcodes for bit operations facing the need to provide operations for large elements it was decided to use the operations provided by **big.Int** instead implementing the operations by ourselves to achieve the possibility to work with numbers bigger than 256 byte. It was also decided to make the operations unsigned because they are primarily thought of as bit operations. The downsides of having to adjust gas calculation to take element size into account is accepted. If in hindsight these operations are primarily used for numbers the implementation can be changed easily.

### 4.5.4. Control Flow Operations

**Implementation** Control flow opcodes like **JMP** are used to change the the program counter and therefore changing the sequence of execution. These opcodes read labels from the instruction set.

**Call Stack** The **CALL** and **CALLIF** opcodes are special, since a new call stack is allocated when they are executed. The call stack has a return address and copies of the values that are passed to the call stack. How many values have to be passed to the call stack is provided by an argument of the **CALL** and **CALLIF** opcode. These copies are used for the operations within the scope of the called function. **RET** is used at the end of the function to jump back to the return address. The remaining values on the call stack are pushed to the evaluation stack and the call stack is deleted.

**Rationale** Control flow opcodes are essential to make the virtual machine Turing complete. Having this type of opcodes allows the contract creator to build loops and conditions. With the introduction of a call stack it is possible to make function calls.

#### 4.5.5. Data Structures

<b>Overview</b>	The two basic data structures map and array are directly implemented as opcodes. Structs could be created on top of an array. For each data structure opcodes are available to add, remove, set and retrieve values.
<b>Implementation</b>	Both data structures were implemented on top of a byte array. The required methods to add, remove, set and retrieve values for both data types have been implemented.
<b>Rationale</b>	The ability to marshal the data structures into a byte array in order to push it on the stack was important. The overhead for doing this should be as low as possible. Further important concerns are keeping the implementation simple and the contents of the structure viewable in the trace function for debugging. It was decided to implement the data structures on top of byte array. The possibility of using the Go native implementation of map and array were neglected. The reasons for this decision are that the array or map would have to be marshalled and unmarshalled by each opcode in order to pop the data structure from the stack, perform the operation and push it back. Also, an array of bytes could not be used as an index for a map since it is not comparable. The heightened possibility of errors in the implementation in contrast to using the Go standard types was accepted, since the blockchain is a research blockchain although measures have been taken to provide as much correctness as possible.

#### 4.6. Context

For the VM execution to have any effect it is necessary to change the miner's state eventually. This section describes how the state changes of a contract execution are persisted and explains the data the context is composed with. Figure 4.8 shows the interface for the VM context.

<b>Consistency</b>	One problem with an immediate change of the miner's state is that the contract execution could fail in the middle of the contract resulting in inconsistencies. Such inconsistencies could be resolved with a rollback but this was neglected since this is more complicated to implement than letting the VM work on copies. Therefore, it was decided to provide the VM with copies of the state variables.
<b>Decoupling</b>	In order to reduce the coupling of the miner it was decided to provide the access to the state variables via a context object which implements the necessary getters and setters. The context object contains the logic needed to create copies and to avoid encapsulation breaches. The context object is setup by the miner, the reference to it is then passed to the VM and the changes are written back after a successful execution of the contract by the miner again.

**Readability** To clearly describe which operations the VM uses to access state variables in the code of the VM an interface was created which the VM uses and the context implements. If necessary this would also allow for easier testing since this makes the mocking and overwriting of specific methods of the context possible.

---

```

1  type Context interface {
2      GetContract() []byte
3      GetContractVariable(index int) ([]byte, error)
4      SetContractVariable(index int, value []byte) error
5      GetAddress() [64]byte
6      GetIssuer() [32]byte
7      GetBalance() uint64
8      GetSender() [32]byte
9      GetAmount() uint64
10     GetTransactionData() []byte
11     GetFee() uint64
12     GetSig1() [64]byte
13 }

```

---

Figure 4.8.: Context interface

#### 4.6.1. Data from a Transaction

**Sender/Address** The sender field shows the public address of the transaction sender

**Fee** The maximum price the transaction can cost.

**TransactionData** This field contains the identifier to the function the sender wants to call on a certain smart contract and its arguments.

**Amount** This field shows the amount of Bazo units send in this transaction.

#### 4.6.2. Data from the Receiver Account

**Issuer/Owner** This field contains the public address of the account owner.

**Balance** This field contains the number of coins in this account.

**Contract** This field is the smart contract itself and contains the byte code. The data type is []byte, so it can be packed into a transaction field.

**ContractVariables** This field contains the state variables that are changed by executing transactions.

## 4.7. Parser

The language that the parser processes cannot be described as a high-level programming language and is very strongly aligned to the actual byte code. The parser package contains two classes and a test file. One file is `tokens.go` which contains the available opcodes with its arguments and the token types. The second file is `parser.go` which can be split into two main functions. The following sections contain an elaboration of mentioned files.

### 4.7.1. Tokens File

#### Token Struct and Types

Figure 4.9 shows the Token struct. The `tokenType` field can be `OPCODE` (Value: 0), `BYTES` (Value: 1), `BYTE` (Value: 2), `ADDR` (Value: 3), `LABEL` (Value: 4) which are all int constants. The `value` field contains the argument passed to the token.

---

```

1  type Token struct {
2      tokenType int
3      value     string
4  }

```

---

Figure 4.9.: Token Struct

#### Opcodes

The opcodes are needed to replace the opcode token with the matching value and to make sure only valid opcodes are passed. The opcodes need to be the same as in the virtual machine and are int constants.

#### Array of Opcodes

The array of opcodes shows which opcodes have arguments and how many of them, in order to check if only the allowed amount of words is found in a single line of the contract. If more arguments than defined are passed to an opcode token, an illegal word in line exception is thrown.

### 4.7.2. Parser File

#### Tokenize() Function

The function that is run first is the **Tokenize()** function. The process of the **Tokenize()** function is shown in figure 4.10. The source code of the contract written in «Enhanced Bazo Byte Code» is passed as a string. The string is converted to an array of lines. The **Tokenize()** function takes every first word in every line and matches it with available token types. Every first word in line must either be a comment, a label, empty or an opcode. The rest of the words in the same line are the parameters of the opcode or an inline comment, marked with a **#**. Comments and empty lines are ignored. Labels end with an colon (e.g. addNums:). If the first word is a label it is added to the labelMap to later replace it with the address.

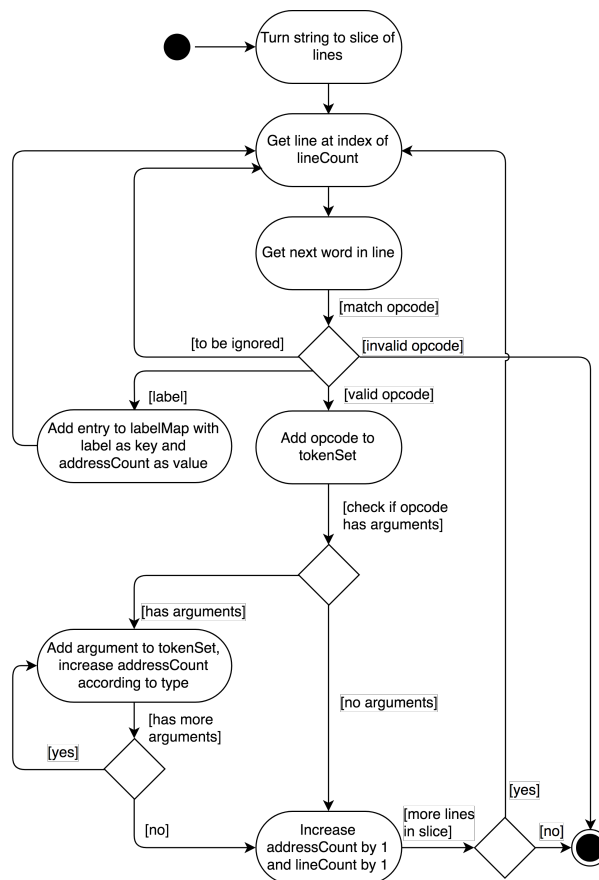


Figure 4.10.: UML Activity Diagram of the Tokenize Function

#### Token Set

Figure 4.11 shows the generated token set to the basic contract shown in Figure 3.7.



---

```

1  {
2  [{0 PUSH} {1 55780}],
3  [{0 PUSH} {1 5}],
4  [{0 CALL} {4 addNums} {2 2}],
5  [{0 HALT}],
6  [{0 LOAD} {2 0}],
7  [{0 LOAD} {2 1}],
8  [{0 ADD}],
9  [{0 RET}],
10 }
```

---

Figure 4.11.: Token set of contract shown in 3.7

### Parse() Function

The `Parse()` function compiles the token set to «Bazo Byte Code». Figure 4.12 shows the process of the function. The function iterates over all tokens in the token set and matches the different types. If the type is `OPCODE` the matching byte value is added to the instruction set. If the type is `LABEL` the value is loaded from the `labelMap`, which is the jump address. A special type is `BYTES`. If an opcode takes `BYTES` as an argument the first byte of the value shows the length of the byte representation. That means, that the byte representation of the value must be prepended with the amount of bytes. `BYTE` appends a single byte value to the instruction set. `ADDR` appends 32 bytes to the instruction set.

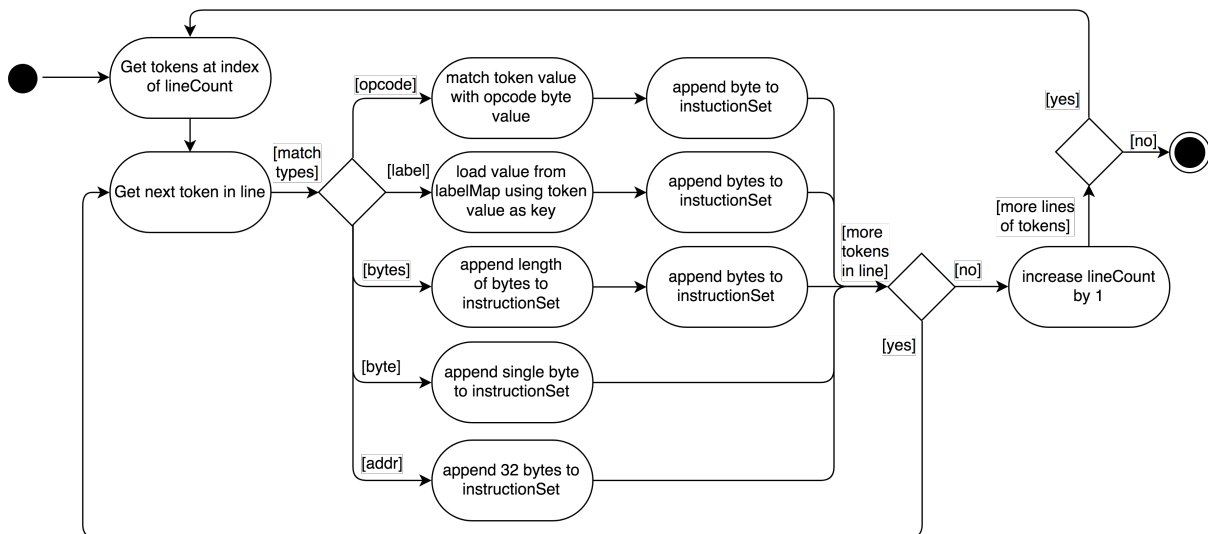


Figure 4.12.: UML Activity Diagram of the Parse Function

**Instruction Set** Figure 4.13 shows the resulting instruction set.

---

```

1  {
2    0, 1, 217, 228, 0, 0, 5, 21, 0, 13, 2, 50, 28, 0, 28, 1, 4, 24,
3  }

```

---

Figure 4.13.: Contract compiled to «Bazo Byte Code»

## 4.8. Testing

The virtual machine, the parser and the code written for the integration were extensively tested from the start. Depending on the type of component and its relations, different testing methods have been applied. Overall a relatively high test coverage could be achieved. In the sections below, the different testing methods for each package are explained.

### 4.8.1. Unit Testing

All packages have been unit tested. The test coverage of each modified package is compared with its coverage before the integration to ensure that the test coverage is not negatively affected by the integration.

<b>Protocol</b>	As described in the sections before, not many changes have been made to the <i>Protocol</i> package. The test coverage before the VM integration was 70%. The updated test coverage is 68%. The reason for a lower coverage is that the <i>vm_context.go</i> file has been added, which has many getters and setters that do not need to be tested.
<b>Miner</b>	The test coverage before the VM integration was 65%. The updated test coverage is 65%, which shows that the integration has not negatively affected the coverage.
<b>VM</b>	The overall statement coverage is 81%. Every component of the <i>VM</i> package has a test coverage over 79%. The main component is the <i>vm.go</i> file. For every opcode at least one unit test has been made. Arithmetic opcodes are based on the <i>big.Int</i> implementation, which has been considered stable. This test coverage is influenced by the fuzz test described in Section 4.8.2.
<b>Parser</b>	The parser should be seen as a utility that is not part of the core project. For this reason, the necessity to test of this package was of low priority. Still a test coverage of 82% could be achieved, because the package is small and the core class consists only of two main functions and a few helper methods.

#### 4.8.2. Fuzz Testing

An instruction set of a smart contract must never be able to crash the miner. Calling a smart contract function with malicious instructions would cause the whole blockchain to collapse. To check if the VM fails gracefully, a fuzz test was implemented which creates contracts with random bytes and then executes them. Contracts causing the miner to crash were reproduced as unit test in order to find the bug. Once the bug was found it was mitigated. This process was repeated over and over again. The fuzz test is executed with five million random contracts with every commit to the remote repository using Travis CI which helped us to find many bugs that could have crashed the miner.

#### 4.8.3. Integration Testing

To test whether the virtual machine could be successfully integrated, an integration test was made. The goal of this test is to show that deploying and calling smart contracts over transactions are possible. The integration test consists of multiple small unit tests. It was tested whether it is possible to create smart contract accounts, call functions of these smart contracts and whether state variables are persisted over several transactions. The integration test could successfully be implemented and run.

## 5. Evaluation

This chapter covers the evaluation of the work by describing the implementation of a tokenization contract in order to show an example of the possible functionality and by providing the implementation of a modular exponentiation contract. The modular exponentiation contract is set in comparison with a Go implementation of the algorithm in order to show the overhead of the execution on the Bazo VM.

### 5.1. Tokenization Contract

Tokenization is the process of recording the rights to an asset as a digital token on a blockchain in the form of sub-currencies. [7] Tokenization is the first use-case for smart contracts that has found wide application. New possibilities for funding start-ups and companies have emerged in the form of initial coin offerings (ICOs) also known as token sales. The tokens of an ICO can be bought by sending money in the form of the blockchain currency to a smart contract. The investor then receives the corresponding amount in tokens. Ideally, the token is an integral component within the ecosystem of the company hosting the ICO. [10] Since tokenization contracts have become so widely used and because they are generally simple, it was decided to write a tokenization contract. The implementation of a very basic tokenization contract consists of a map of account addresses and balances and methods to transfer those tokens from one account to another.

#### 5.1.1. Results

As a result, a contract was created which allows to store addresses together with balances. Only the account address which is recorded as minter is allowed to change balances. The balance can be reduced by sending negative values or increased by sending positive values. If the map does not yet contain an address when sending tokens, it is added automatically. The contract is shown in Appendix C.

### 5.2. Benchmarking Contract

The performance of the virtual machine is crucial for the speed of execution and the blockchain in general. For this reason a smart contract has been developed which is suitable for comparing the speed of execution on different implementations and platforms. Taking into consideration that blockchains and its use cases depend heavily on public-key cryptography, it was decided to implement a smart contract which performs modular exponentiation. Modular exponentiation

is a one-way function and frequently used in cryptography. Figure 5.1 shows the straightforward method to calculate  $c$ .

$$c \equiv b^e \text{ mod } m$$

Figure 5.1.: Modular exponentiation straightforward method

Considering  $b$  is at least 256 bits for strong cryptography, this method is not very efficient. Therefore, a more memory efficient method has been implemented, which is shown in figure 5.2. [12] The benchmarking function has been implemented in Go and as a contract in «Bazo Byte Code». The main goal is to compare the overhead.

---

```

1  function modular_pow(base, exponent, modulus)
2      if modulus = 1 then return 0
3      c := 1
4      for e_prime = 0 to exponent-1
5          c := (c * base) mod modulus
6      return c

```

---

Figure 5.2.: Memory efficient method to compute modular exponentiation

### 5.2.1. Results

The Go testing package contains a subset of functions to measure the performance of Go code. A benchmark function runs the code  $b.N$  times.  $b.N$  is adjusted during execution, until the benchmark function can be timed reliably. [9] Three benchmark functions for both, the Go implementation and the contract, have been implemented, where the only difference is the length of  $b$ . The length of  $b$  is 32 bytes in the first benchmark function, which is the minimum length for strong cryptography. In the second benchmark function the length of  $b$  is 128 bytes and in the last,  $b$  has a length of 255 bytes, which is the maximum length the virtual machine can process. The values of  $b$ ,  $e$  and  $m$  are random generated every run. The benchmarks have been measured on a Fujitsu Celsius W530, with 15.6 GiB RAM, an Intel Xeon CPU E3-1245 v3 3.40GHz x 8 processor, running Ubuntu 17.10 as operating system. The following benchmarking measurements have been analyzed.

#### Nanoseconds per operation

Figure 5.3 shows how many nano seconds the operation took. Operation describes the code that was run, in this case the Go implementation and the benchmarking contract. The benchmarking contract took 9.41 times longer than the Go implementation in the first benchmark function with  $b = 32 \text{ bytes}$ . In the second function with  $b = 128 \text{ bytes}$ , the factor was 7.24. In the last function with  $b = 255 \text{ bytes}$  this factor went down to 5.89. The conclusion is, that the longer  $b$  is, the better the performance of the benchmarking contract gets, compared to the program written in Go.

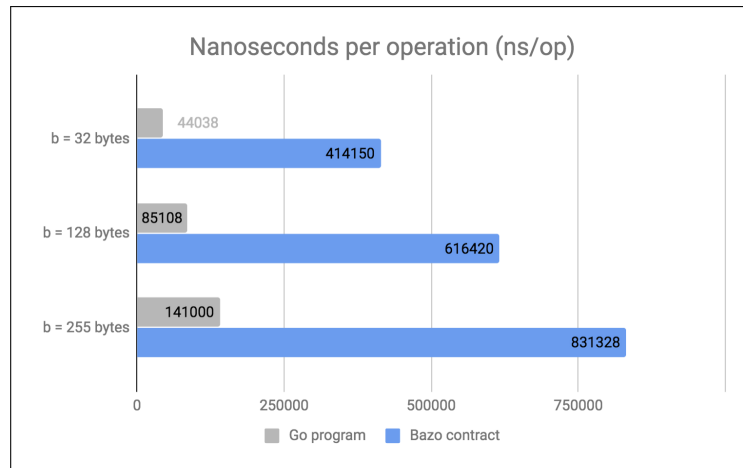


Figure 5.3.: Nanoseconds per operation diagram

### Bytes per operation

Figure 5.4 shows how many bytes per operation have been allocated. The factor between the Go implementation and the benchmarking contract was stable during benchmark functions. The benchmarking contract needs about 10 times more memory. This result was expected since the benchmarking contract often needs to duplicate and roll values to permute a loop.

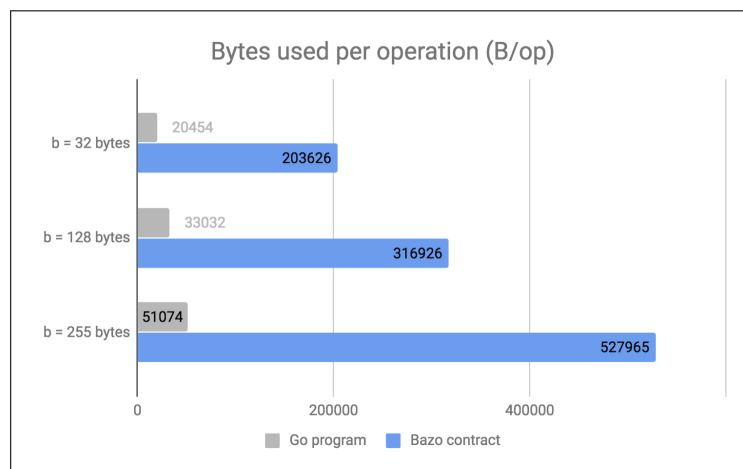


Figure 5.4.: Bytes per operation diagram

### Allocations per operation

Figure 5.5 shows how many bytes per operation have been allocated. The factor between the Go implementation and the benchmarking contract was stable in all benchmarks. The benchmarking contract needs about 12 times more bytes. This result was expected, for the same reason mentioned above.

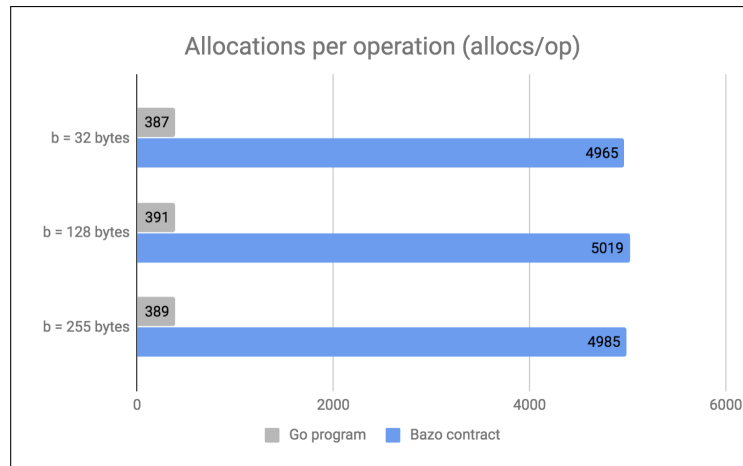


Figure 5.5.: Allocation per operation diagram

## 6. Conclusion

<b>Summary</b>	<p>This thesis consisted of the sub goals of building and integrating a virtual machine into the Bazo Blockchain, to make the execution of smart contracts possible. The exact details needed to realize these requirements were not known at the beginning and were defined by the analysis of other smart contract platforms such as Ethereum or NEO and blockchains in general. An important conclusion from this analysis was, that it is very important to ensure the VM is fault tolerant and does not crash when a malicious contract is executed. The possibility of extensions by follow-up works was kept in mind. For this reason, the VM has been implemented in a manner that new opcodes can be defined easily. In addition, care was taken to describe the error messages as helpful as possible.</p>
<b>Unique Features</b>	<p>When calculating the gas costs a different approach was chosen, which is not used by any another blockchain, as both the instruction and the size of the elements are taken into account. Furthermore, our VM differs from others because it can work with elements of arbitrary size, which is helpful for many cryptographic functions. The only limitation here is that the current maximal pushable element cannot be larger than 256 bytes.</p>
<b>Parser</b>	<p>The parser was originally not part of the scope. The decision to implement this simple parser was made when larger contracts had to be implemented, which contained many control flow opcodes and therefore addresses often had to be counted. This parser could serve as a basis for future works.</p>
<b>Concluding statements</b>	<p>The goals of this thesis could be achieved. The achievement of these goals was successfully proven by extensive testing. This work laid the foundation for Bazo as a platform for decentralized applications. However, in order to keep up with existing smart contract platforms, performance improvements and an increase in usability for creating and calling contracts would be necessary.</p>



## 6.1. Future Work

<b>Compiler</b>	Whilst it is already possible to write contracts using the opcodes of the VM or »Enhanced Bazo Byte Code«. This is very hard to read for humans. To make the writing of contracts easier, a compiler which processes a higher-level language and translates it into the »Bazo Byte Code« would be useful.
<b>IDE</b>	There should also be an environment in which contracts can be written, tested and directly deployed.
<b>Separation of Gas and Currency</b>	The execution of a contract costs a certain amount of coins. This is called gas. When blockchains gain popularity the price of the coin in fiat currency usually rises. Since the gas price stays the same but the coin becomes more valuable the actual execution cost of a contract rises too. A solution to this problem is to provide a separate currency for the gas.
<b>Contract Pays the Fee</b>	Another feature would be the ability call a contract and execute a function where the fee is paid by the contract instead of the caller. So there is no need for the user to own Bazo coins in order to interact with the blockchain.
<b>Platform Independent Encoding</b>	The mechanism which is used to encode and decode transactions should be platform independent. Currently the encoding of transactions is implemented in a Go dependent encoding mechanism. This is currently no problem since all software systems are implemented in Go and actually faster, but it also restricts future projects to the use of Go as well.

## List of Figures

2.1. Bazo Blockchain system context diagram . . . . .	11
2.2. Bazo Blockchain container diagram . . . . .	13
3.1. Virtual machine execution cycle overview adapted from [18] and [6] . . . . .	16
3.2. Register based virtual machine . . . . .	17
3.3. Stack based virtual machine . . . . .	18
3.4. Struct of the type FundsTx . . . . .	20
3.5. Struct of the type AccTx . . . . .	21
3.6. Struct of the type Account . . . . .	22
3.7. Basic contract with function call written in «Enhanced Bazo Byte Code» . . . . .	23
4.1. UML Package Diagram . . . . .	27
4.2. New gob based encoding function . . . . .	29
4.3. New gob based decoding function . . . . .	29
4.4. addFundsTx function . . . . .	30
4.5. Example of an error message. . . . .	33
4.6. Gas cost equation . . . . .	34
4.7. Trace function output . . . . .	34
4.8. Context interface . . . . .	38
4.9. Token Struct . . . . .	39
4.10. UML Activity Diagram of the Tokenize Function . . . . .	40
4.11. Token set of contract shown in 3.7 . . . . .	41
4.12. UML Activity Diagram of the Parse Function . . . . .	41
4.13. Contract compiled to «Bazo Byte Code» . . . . .	42
5.1. Modular exponentiation straightforward method . . . . .	45
5.2. Memory efficient method to compute modular exponentiation . . . . .	45
5.3. Nanoseconds per operation diagram . . . . .	46
5.4. Bytes per operation diagram . . . . .	46
5.5. Allocation per operation diagram . . . . .	47

## Bibliography

- [1] Alfred V. Aho, ed. *Compilers: principles, techniques, & tools*. 2nd ed. OCLC: ocm70775643. Boston: Pearson/Addison Wesley, 2007. 1009 pp. ISBN: 978-0-321-48681-3.
- [2] Jan von der Assen. *A Progressive Web App (PWA)-based Mobile Wallet for Bazo*. Accessed 22 Mai 2018. 2018-01. URL: <https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Jan-von-der-Assen.pdf>.
- [3] Simon Bachmann. *Proof of Stake for Bazo*. Accessed 22 Mai 2018. 2018-02. URL: <https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Simon-Bachmann.pdf>.
- [4] Luc Boillat. *A Blockchain Explorer for Bazo*. Accessed 22 Mai 2018. 2018-01. URL: <https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Luc-Boillat.pdf>.
- [5] Marc-Alain Chételat. *Design and Prototypical Implementation of a Mobile Light Client for the Bazo Blockchain*. Received on request to Prof. Dr. Thomas Bocek. 2018-03.
- [6] *Ethereum Block Architecture Image*. Accessed 15 June 2018. URL: <https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture/6413#6413>.
- [7] Ethereum Foundation. *Ethereum White Paper*. Accessed 25 Mai 2018. URL: <https://github.com/ethereum/wiki/wiki/White-Paper>.
- [8] *Golang Package gob*. Accessed 5 June 2018. URL: <https://golang.org/pkg/encoding/gob/>.
- [9] *Golang Package testing*. Accessed 11 June 2018. URL: <https://golang.org/pkg/testing/>.
- [10] *Initial coin offerings - Legal Frameworks and regulations for ICOs*. Accessed 5 June 2018. URL: <https://www.pwc.ch/en/industry-sectors/financial-services/fs-regulations/ico.html>.
- [11] Noam Levenson. *NEO versus Ethereum: Why NEO might be 2018's strongest cryptocurrency – Hackernoon*. Accessed 18 April 2018. 2017-12. URL: <https://hackernoon.com/neo-versus-ethereum-why-neo-might-be-2018s-strongest-cryptocurrency-79956138bea3>.
- [12] *Modular exponentiation*. In: Page Version ID: 841062658. 2018-05-13. URL: [https://en.wikipedia.org/w/index.php?title=Modular\\_exponentiation&oldid=841062658](https://en.wikipedia.org/w/index.php?title=Modular_exponentiation&oldid=841062658) (visited on 2018-06-14).
- [13] multiple. *Database transaction*. Accessed 21 Mai 2018. 2018-04. URL: [https://en.wikipedia.org/wiki/Database\\_transaction](https://en.wikipedia.org/wiki/Database_transaction).
- [14] *NEO White Paper*. Accessed 25 Mai 2018. 2016-04. URL: <http://docs.neo.org/en-us/index.html>.
- [15] *Oxford Dictionaries Design*. Accessed 8 June 2018. URL: <https://en.oxforddictionaries.com/definition/design>.
- [16] Livio Sgier. *Bazo – A Cryptocurrency from Scratch*. Accessed 22 Mai 2018. 2017-08. URL: <https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Livio-Sgier.pdf>.
- [17] Mark Sinnathamby. *Stack based vs Register based Virtual Machine Architecture, and the Dalvik VM*. Accessed 25 Mai 2018. 2012-07. URL: <https://markfaction.wordpress.com/2012/07/15/stack-based-vs-register-based-virtual-machine-architecture-and-the-dalvik-vm/>.

- [18] Dr Gavin Wood. *Ethereum Yellow Paper*. Accessed 15 June 2018. URL: <https://ethereum.github.io/yellowpaper/paper.pdf>.

## A. Installation Guidelines

### A.1. Miner Application

#### A.1.1. Prerequisites

The programming language Go (developed and tested with version  $\geq 1.9$ ) must be installed, the properties `$GOROOT` and `$GOPATH` must be set. For more information, please check out the official documentation.

Before the bazo-miner can be started, two public-private key-pairs are required. The key-pairs can be generated with the bazo-keypairgen application. Run the following instructions in your terminal.

1. Download the bazo-keypairgen application.

---

```
go get github.com/bazo-blockchain/bazo-keypairgen
```

---

2. Build the application.

---

```
$GOPATH/src/github.com/bazo-blockchain/bazo-keypairgen  
go build
```

---

3. Run the application to generate the *validator* public-private keypair. The validator is the keyfile's name containing the validator's public key.

---

```
./bazo-keypairgen validator.txt
```

---

4. Run the application to generate the *multisig* public-private keypair. The multisig is the keyfile's name containing the multi-signature server's public key.

---

```
./bazo-keypairgen multisig.txt
```

---

### A.1.2. Getting Started

1. Download the bazo-miner application.

---

```
go get github.com/bazo-blockchain/bazo-miner
```

---

2. Copy both previously generated files *validator.txt* and *multisig.txt* into the root folder of the bazo-miner folder.

---

```
$GOPATH/src/github.com/bazo-blockchain/bazo-keypairgen  
cp validator.txt $GOPATH/src/github.com/bazo-blockchain/bazo-miner/validator.txt  
cp multisig.txt $GOPATH/src/github.com/bazo-blockchain/bazo-miner/multisig.txt
```

---

3. Open the storage configuration file *storage/configs.go* in an editor of your choice.

---

```
$GOPATH/src/github.com/bazo-blockchain/bazo-keypairgen  
cp validator.txt $GOPATH/src/github.com/bazo-blockchain/bazo-miner/validator.txt  
cp multisig.txt $GOPATH/src/github.com/bazo-blockchain/bazo-miner/multisig.txt
```

---

Replace the value of *INITROOTPUBKEY1* with the first line of *validator.txt*. Replace the value of *INITROOTPUBKEY2* with the second line of *validator.txt*.

4. Build the application.

---

```
$GOPATH/src/github.com/bazo-blockchain/bazo-miner  
go build
```

---

5. Run the application.

---

```
./bazo-miner "database_file.db" ":8000" "validator.txt" "seedfile.txt" "multisig.txt"
```

---

The ipport number must be prefixed with ":". If the miner is intended to run locally, the localhost ip address has to be passed with the ipport. Otherwise the miner tries to connect to the network. Note that "database\_file.db" and "seedfile.txt" are created if they do not exist.

## A.2. Parser Application

### A.2.1. Prerequisites

The programming language Go (developed and tested with version  $\geq 1.9$ ) must be installed, the properties `$GOROOT` and `$GOPATH` must be set. For more information, please check out the official Go documentation.

### A.2.2. Getting Started

1. Download the bazo-parser application

---

```
go get github.com/bazo-blockchain/bazo-parser
```

---

2. Build the application.

---

```
$GOPATH/src/github.com/bazo-blockchain/bazo-parser  
go build
```

---

3. Run the application

---

```
./bazo-parser
```

---

4. Define the path to the smart contract. After hitting enter, the parser prints the compiled byte code instructions, ready to copy to the virtual machine or the miner.

---

```
Define the path to your contract  
./contracts/addNums.sc
```

---

## B. Opcodes

This table shows which opcodes are available. The description right of the arrow shows how the instruction is processed. The result is then pushed onto the stack. The arguments column shows which arguments are read from the instruction set.

Table B.1.: List of available opcodes

Mnemonic	Op-Code	Arguments	Description	Gas price	Gas factor
PUSH	0x00	bytes (bytes)	$\text{stack} \leftarrow \text{bytes}$	1	1
DUP	0x01	-	$\text{stack} \leftarrow 2x \text{ pop1}$	1	1
ROLL	0x02	index (byte)	removes element at index and push to ToS	1	2
POP	0x03	-	pops ToS	1	1
ADD	0x04	-	$\text{stack} \leftarrow \text{pop1} + \text{pop2}$	1	2
SUB	0x05	-	$\text{stack} \leftarrow \text{pop1} - \text{pop2}$	1	2
MULT	0x06	-	$\text{stack} \leftarrow \text{pop1} * \text{pop2}$	1	2
DIV	0x07	-	$\text{stack} \leftarrow \text{pop1} / \text{pop2}$	1	2
MOD	0x08	-	$\text{stack} \leftarrow \text{pop1} \% \text{pop2}$	1	2
NEG	0x09	-	$\text{stack} \leftarrow \text{pop1}$	1	2
EQ	0x0a	-	$\text{stack} \leftarrow 1$ if $\text{pop1} == \text{pop2}$ , 0 otherwise	1	2
NEQ	0x0b	-	$\text{stack} \leftarrow 1$ if $\text{pop1} != \text{pop2}$ , 0 otherwise	1	2
LT	0x0c	-	$\text{stack} \leftarrow 1$ if $\text{pop1} < \text{pop2}$ , 0 otherwise	1	2
GT	0x0d	-	$\text{stack} \leftarrow 1$ if $\text{pop1} > \text{pop2}$ , 0 otherwise	1	2
LTE	0x0e	-	$\text{stack} \leftarrow 1$ if $\text{pop1} \leq \text{pop2}$ , 0 otherwise	1	2
GTE	0x0f	-	$\text{stack} \leftarrow 1$ if $\text{pop1} \geq \text{pop2}$ , 0 otherwise	1	2
SHIFTL	0x10	nrOfShifts (byte)	$\text{stack} \leftarrow \text{pop1} \ll \text{nrOfShifts}$	1	2



Table B.1.: List of available opcodes

Mnemonic	op-Code	Arguments	Description	Gas price	Gas factor
SHIFTR	0x11	nrOfShifts (byte)	stack $\leftarrow$ pop 1 $\gg$ nrOfShifts	1	2
NOP	0x12	-	does nothing	1	1
JMP	0x13	address (label)	jump to address	1	1
JMPIF	0x14	address (label)	jumps to address if pop1 == 1	1	1
CALL	0x15	address (label), nrOfArgs (byte)	call a function at address with a given amount of arguments	1	1
CALLIF	0x16	address (label), nrOfArgs (byte)	calls a function if pop1 == 1 at address with a given amount of arguments	1	1
CALLEXT	0x17	address (address), functionHash (4x byte), nrOfArgs (byte)	calls a function with functionHash from an external smart contract account with a given amount of arguments	1000	2
RET	0x18	-	returns from function	1	1
SIZE	0x19	-	stack $\leftarrow$ size(pop1)	1	1
STORE	0x1a	-	stores pop1 in callStack	1	2
SSTORE	0x1b	index (byte)	stores pop1 in contractVariables at index	1000	2
LOAD	0x1c	index (byte)	loads variable at index from callStack to evaluationStack	1	1
SLOAD	0x1d	index (byte)	loads variable at index from contractVariables to evaluationStack	10	1
ADDRESS	0x1e	-	stack $\leftarrow$ receiver account address	1	1
ISSUER	0x1f	-	stack $\leftarrow$ receiver account issuer	1	1
BALANCE	0x20	-	stack $\leftarrow$ receiver account balance	1	1
CALLER	0x21	-	stack $\leftarrow$ contract caller	1	1
CALLVAL	0x22	-	stack $\leftarrow$ transaction amount in bazo coins	1	1
CALLDATA	0x23	-	stack $\leftarrow$ transaction data	1	1
NEWMAP	0x24	-	stack $\leftarrow$ new map	1	2
MAPHASKEY	0x25	-	stack $\leftarrow$ search map for key and push bool value for result	1	2
MAPPUSH	0x26	-	stack $\leftarrow$ pop1.insert(pop2, pop3)	1	2

Table B.1.: List of available opcodes

Mnemonic	op-Code	Arguments	Description	Gas price	Gas factor
MAPGET-VAL	0x27	-	$\text{stack} \leftarrow \text{pop1}[\text{pop2}]$	1	2
MAPSETVAL	0x28	-	$\text{pop1}[\text{pop2}] = \text{pop3}$	1	2
MAPRE-MOVE	0x29	-	$\text{stack} \leftarrow \text{pop1.remove}(\text{pop2})$	1	2
NEWARR	0x2a	-	$\text{stack} \leftarrow \text{new array}$	1	2
ARRAPPEND	0x2b	-	$\text{stack} \leftarrow \text{pop1.append}(\text{pop2})$	1	2
ARRINSERT	0x2c	-	$\text{stack} \leftarrow \text{pop1}[\text{pop2}] = \text{pop3}$	1	2
ARRRE-MOVE	0x2d	-	$\text{pop1.remove}(\text{pop2})$	1	2
ARRAT	0x2e	-	$\text{stack} \leftarrow \text{pop1}[\text{pop2}]$	1	2
SHA3	0x2f	-	$\text{stack} \leftarrow \text{SHA3\_HASH}(\text{pop } 1)$	1	2
CHECKSIG	0x30	-	$\text{stack} \leftarrow \text{ecdsa.Verify}(\text{pop } 1, \text{pop2})$	1	2
ERRHALT	0x31	-	return from Exec() function with false	0	1
HALT	0x32	-	return from Exec() function with true	0	1

## C. Tokenization Contract

---

```

1  CALLDATA
2  # ----- ABI -----
3  DUP
4  PUSH 1
5  EQ
6  CALLIF mint 3
7  HALT
8
9  # ----- Contract -----
10 mint:
11  LOAD 1 # load value
12  LOAD 0 # load key
13  SLOAD 1 # load address of minter
14  CALLER
15  EQ
16  CALLIF adjustBalance 2
17  RET
18
19 adjustBalance:
20  LOAD 1 # load value
21  LOAD 0 # load key
22  DUP
23  SLOAD 2 # load map
24  MAPHASKEY
25  CALLIF addKeyIfNotExists 2
26  LOAD 1 # load value
27  LOAD 0 # load key
28  SLOAD 2 # load map
29  MAPPUSH
30  SSTORE 2 # store the map
31  HALT
32
33 addKeyIfNotExists:
34  LOAD 1 # load key
35  SLOAD 2 # load map
36  MAPGETVAL
37  LOAD 0 # load value
38  ADD
39  LOAD 1 # load key
40  SLOAD 2 # load map
41  MAPSETVAL

```

```
42 SSTORE 2 # store the map
43 HALT
```

---

## D. Definition of Task

---

### Aufgabenstellung Bachelorarbeit "Integrating Smart Contracts into the Bazo Blockchain"

---

#### 1. Betreuer und Experte

Diese Arbeit wird von Prof. Dr. Thomas Bocek, HSR, IFS, [tbocek@hsr.ch](mailto:tbocek@hsr.ch) betreut.

Industriepartner:

- Keine

#### 2. Studierende

Diese Arbeit wird als Bachelorarbeit an der Abteilung Informatik durchgeführt von

- Ennio Meier
- Marco Steiner

#### 3. Einführung

Die Bazo Blockchain ist eine Blockchain um verschiedene Mechanismen und Algorithmen zu testen. In der momentanen Version sind Proof of Stake integriert, sowie Mechanismen um die Bazo Blockchain auf einem mobilen Gerät laufen zu lassen. Es lassen sich nur Bazo Coins transferieren. Die Idee dieser Arbeit ist es die Bazo Blockchain um Smart Contracts zu erweitern.

#### 4. Ziele der Arbeit

Das Ziel dieser Arbeit ist das Design, Implementation und die Evaluation einer virtuellen Maschine für Smart Contracts für die Bazo Blockchain. Diese Arbeit dokumentiert das Vorgehen, Entwurfsentscheidungen und das Design und diskutiert alle wichtigen Details und Erweiterungen. Das Ziel dabei ist es, eine minimal funktionale virtuelle Maschine testen und evaluieren zu können.

Es sollen dabei aktuelle Ansätze angeschaut (wie zum Beispiel die Ethereum Virtual Machine) und Verbesserungen vorgenommen werden.

Die virtuelle Maschine sollte turing-complet sein, das heisst, es soll eine generische virtuelle Maschine für Smart Contracts erstellt werden. Ein Smart Contract soll dabei States von Accounts lesen aber auch persistieren können.

Die Bazo Blockchain baut auf einer vorgängigen Bachelor- und Masterarbeit auf, die sich mit den Grundlagen und ausgewählten Mechanismen auseinandergesetzt haben. Es kann davon ausgegangen werden, dass die Resultate aus der Vorarbeit in brauchbarer Form verfügbar sind. Änderungen an der vorherigen Arbeit notwendig um die virtuelle Maschine ins bestehende System zu integrieren. Zusätzliche Funktionen können je nach Zeitbudget und Priorität umgesetzt werden.

---

Es finden wöchentliche Besprechungen mit dem Betreuer statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen.

Alle Besprechungen, ausser der Kick-off Besprechung, sind von den Studierenden mit einer Traktandenliste vorzubereiten und zu leiten. Bei jedem Meeting soll der aktuelle Stand des Projektes präsentiert werden (Was wurde gemacht? Was wurde erreicht?). Die Beschlüsse der Besprechungen sind durch die Studierenden zu protokollieren und an den Betreuer anschliessend zuzustellen. Die Zeit-Protokollierung wird bei Bedarf oder nach Wunsch des Betreuers eingesehen.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen (oder auch Zwischenversionen) gemäss Projektplan sind einzelne Arbeitsergebnisse in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsergebnisse erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Resultate.

## 5. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente bzw. Berichtsteile sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

## 6. Termine

Siehe auch Terminplan auf dem Skripteserver (Fachbereich/Bachelor-Arbeit\_Informatik/BAI/)

<b>Montag, den 19.02.2018</b>	<b>Beginn der Studienarbeit</b>
<b>08.06.2018</b>	<p>Die Studierenden geben den Abstract für die Diplomarbeitsschöpfung zur Kontrolle an ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool.</p> <p>Die Studierenden senden per Email das A0-Poster zur Prüfung an ihren Examinator/Betreuer.</p> <p>Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf dem Skripteserver zur Verfügung.</p>
<b>13.06.2018</b>	<p>Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract der Broschüre zur Weiterverarbeitung an das Studiengangsekretariat frei.</p> <p>Fertigstellung und Weitergabe des A0 Posters per Email bis 10.00 Uhr an das Studiengangsekretariat.</p>
<b>15.06.2018</b>	<p><b>Abgabe des Berichts an den Betreuer bis 12.00 Uhr</b></p> <p>Präsentation und Ausstellung der Bachelorarbeiten, 16 bis 20 Uhr</p>

<b>Bis zum 24.08.18</b>	Mündliche BA-Prüfung
-------------------------	----------------------

## 7. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von 30 Stunden.

Für die Beurteilung ist der verantwortliche Dozent zuständig.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/6
2. Berichte (Abstract, Mgmt Summary, technischer u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/6
3. Inhalt*)	3/6
4. Mündliche Prüfung	1/6

\*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit präzisiert (u.A. durch die Bewertungsmatrix)

Im Übrigen gelten die Bestimmungen der Abteilung Informatik für Studienarbeiten.

Rapperswil, den 12. April 2018.

Prof. Dr. Thomas Bocek