Ethereum: The Ultimate Smart Contract and Decentralized Application Platform

In the last few months, there has been a great amount of interest into the area of using Bitcoin-like blockchains, the mechanism that allows for the entire world to agree on the state of a public ownership database, for more than just money. Perhaps the first, and oldest, such alternative application is colored coins, which is a protocol that allows users to label specific bitcoins and treat them as assets representing some real world value - whether company shares, collectibles or even existing currencies like gold and USD. A more independent alternative, Ripple, also includes the ability to create custom currencies and assets, but adds a decentralized exchange. More recently, Mastercoin has started to go even further, allowing more complex financial contracts such as hedging, trust-free dice rolls, binary options and self-stabilizing currencies - essentially, almost any common financial instrument imaginable. Taken together, all of these projects can be thought of as initial efforts toward a sort of "cryptocurrency 2.0" - they are to Bitcoin what Web 2.0 was to the World Wide Web circa 1995.

At the same time, there has been significant interest in "decentralized autonomous corporations" - autonomous entities that operate on the blockchain in a completely transparent and publicly managed way without any central control whatsoever. Rather than the relationships of the investors, owners and employees of the corporation being mediated by a legal contract or a set of organizational bylaws, the funds and corporate resources are managed directly on the blockchain. However, decentralized autonomous corporations are difficult to implement today, simply because the scripting systems of Bitcoin, and even proto-cryptocurrency 2.0 alternatives like Ripple and Mastercoin, are far too limited to allow the kind of arbitrarily complex computation that DACs require. Although these platforms have begun to offer increasingly complex contracts such as financial derivatives, order matching and trust-free bets, the way that the protocols are set up is inherently limited and closed-ended: each of these use cases is treated as a specific transaction type, not allowing any way for users to build contracts that the developers have not specifically chosen to include.

What this project intends to do is take cryptocurrency 2.0, and generalize it - create a fully-fledged, Turing-complete (but heavily fee-regulated) cryptographic ledger that allows participants to encode arbitrarily complex contracts, autonomous agents and relationships that will be mediated entirely by the blockchain. On-chain currencies, futures contracts, prediction markets, Namecoin-style domain name systems and even provably fair gambling sites will become trivial to implement, existing as simple, hundred-line-of-code contracts on the chain.

Basic Building Blocks

At its core, Ethereum starts off as a fairly regular proof-of-work mined cryptocurrency without many extra complications; in fact, Ethereum is actually in many ways simpler than the Bitcoin-based cryptocurrencies that we use today. The concept of a transaction having multiple inputs and outputs, for example, is gone, replaced by a more intuitive balance-based model (to prevent transaction replay attacks, as part of each account balance we also store an incrementing nonce). Sequence numbers and lock times are also removed, and all transaction and block data is encoded in a single format. Instead of addresses being the RIPEMD160 hash of the SHA256 hash of the public key, addresses are simply the last 20 bytes of the SHA256 hash of the public key. Unlike other cryptocurrencies, which aim to have "features", Ethereum intends to take features away, and instead provide its users with near-infinite power through an all-encompassing mechanism called "contracts".

Ethereum Client P2P Protocol

The Ethereum client P2P protocol is a fairly standard cryptocurrency protocol, and can just as easily be used for any other cryptocurrency; the only modification is the introduction of the "Greedy Heaviest Observed Subtree" (GHOST) protocol first introduced by Yonatan Sompolinsky and Aviv Zohar here: http://www.cs.huji.ac.il/~avivz/pubs/13/btc*scalability*full.pdf; the motivation for and implementation of GHOST will be described in more detail below. The Ethereum client will be entirely reactive; it will not do anything by itself (except for the networking daemon maintaining connections) unless provoked. However, the client will also be more powerful; unlike bitcoind, which only stores a limited amount of data about the blockchain, the Ethereum client will also act as a fully functional backend for a block explorer.

When the client reads a message, it will perform the following protocol:

1. Hash the data, and check if the data with that hash has already been received. If so, exit. Otherwise, pass it along to the data parser.
2. If the data parser says that a data item is a valid block, go to step 3. If the data parser says that a data item is a transaction, see if there are enough funds in the sending address for the transaction to go through, and if there are add it to the local transaction list and publish it to the network. If the data parser says that a data item is a message, send the message to the message responder and return the response.
3. Check if the "parent" parameter in the block is already stored in the database. If it is not, exit
4. Check if every block header in the "uncles" parameter in the block has the block's parent as its own parent. If any is not, exit. Note that uncle blocks do not need to be in the database; they just need to have valid proof of work.
5. Call the state updater with arguments (1) the parent of the block, (2) the transaction list of the block, (3) the timestamp of the block and (4) the coinbase of the block and see if the block header outputted by the state updater is exactly the same. If not, exit. If yes, add the block to the database and publish it to the network.
6. Determine `TD(block)` ("total difficulty") for the new block. TD is defined recursively by `TD(genesis_block) = 0` and `TD(B) = TD(B.parent) + sum(u.difficulty for u in B.uncles) + B.difficulty`. If the new block has higher TD than the current block, set the current block to the new block.

If the node is mining, the node performs the following additional steps upon receiving a block:

1. Determine if the block's parent is in the database. If not, discard it.
2. Determine `TD(block)` ("total difficulty"). If `TD(block)` is higher than of any existing block in the database, start mining on that block and clear all uncles.
3. If the block's parent is the parent of the highest block, add the block header to the set of uncles and restart the proof of work.

The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate; because blocks take a certain time to propagate through the network, call it `t`, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing stale blocks 70% of the time whereas B will have a risk of producing stale blocks 90% of the time. Thus, if the stale rate is high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having enough percentage of the network to have de facto control over the mining process. GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest". Ethereum only does GHOST down one level for simplicity (ie. stale blocks must be included as uncles in the next block in order to count), but this should have 90%+ of the benefit of GHOST. Also, in Ethereum we give stales 3/4 of their block reward (and the nephew that includes them 1/8 as a reward); this modification is intended to solve the second issue of centralization bias.

Currency and Issuance

The Ethereum network includes its own built-in currency, ether. The main reason for including a currency in the network is to serve as a mechanism for paying transaction fees for anti-spam purposes; of the two main alternatives to fees, proof of work and feeless laissez-faire, the former is wasteful of resources and unfairly punitive against weak computers and the latter would lead to the network being almost immediately overwhelmed by an infinitely looping "logic bomb" contract.

Ether will have a theoretical hard cap of 2^128 units (compare 2^50.9 in BTC), although not more than 2^105 units will be released in the foreseeable future. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate), the denominations will be pre-labelled:

- 1: wei
- 2^32: finney
- 2^64: ether
- 2^96: koblitz
- 2^128: turing

Issuance model: to be determined. The main contenders are moderate Quark-style (64x for 3 weeks, 32x for 3 weeks, etc, then 1x forever), limited Mastercoin-style fundraiser, limited Ripple fundraiser or any linear combination of the above.

Data Format

All data in Ethereum will be stored in recursive length prefix encoding, which serializes arrays of strings of arbitrary length and dimension into strings. For example, `['dog', 'cat']` is serialized (in byte array format) as `[ 1, 10, 0, 3, 100, 111, 103, 0, 3, 99, 97, 116 ]`; the general idea is to store the data type (1 for array, 0 for string) followed by the length followed by the actual data (eg. converted into a byte array, 'dog' becomes `[ 100, 111, 103 ]`, so its serialization is `[ 0, 3, 100, 111, 103 ]`. Note that RLP encoding is, as suggested by the name, recursive; when RLP encoding an array, one is really encoding a string which is the concatenation of the RLP encodings of each of the elements. In the event that storing a number is required, the number will be stored as a string in big-endian base 256 format (eg. 32767 in byte array format as `[ 127, 255 ]`).

A full block is stored as:

```
[ block_header, transaction_list , uncle_list ]
```

The block header is:

```
[ parent hash, sha256(rlp_encode(uncle_list)), coinbase address, state_root, sha256(rlp_encode(transaction_list)), difficulty, timestamp,
```

Where the data for the proof of work is the RLP encoding of the block WITHOUT the nonce. `uncle_list` and `transaction_list` are the lists of the uncle block headers and tranctions in the block, respectively.

The `state_root` is the root of a Merkle Patricia tree (formally specified here) containing (key, value) pairs for all addresses where each address is represented as a 20-byte binary string. At each address, the value stored in the Merkle Patricia tree is an object of one of the following two forms:

```
[ 0, balance, nonce ]
```

```
[ 1, balance, contract_root ]
```

The first column states whether the address belongs to a private key (like standard Bitcoin addresses) or a contract (see later section of this document); this distinction is vaguely similar to Bitcoin's distinction between "pubkey addresses" and "P2SH addresses", though the difference is much more profound. `nonce` denotes the number of transactions sent from the address (or an empty string if the address is fresh), and must be included in each transaction (see below). The purpose of this is to make each transaction valid only once, preventing replay attacks. `balance` refers to the contract or address's balance, denominated in wei. `contract_root` is the root of yet another Patricia tree, conaining the contract's memory. Note that in the main Patricia tree all addresses have a length of 20, even if they start with one or more zero bytes, and in the contract subtrees all indices have a length of 32, prepending with zero bytes if necessary.

Mining algorithm

Over the past five years of experience with Bitcoin and alternative cryptocurrencies, one important property for proof of work functions that has been discovered is that of "memory-hardness" - computing a valid proof of work should require not only a large number of computations, but also a large amount of memory. Currently, two major categories of memory-hard functions, scrypt and Primecoin mining, exist, but both are imperfect; neither require nearly as much memory as an ideal memory-hard function could require, and both suffer from time-memory tradeoff attacks, where the function can be computed with significantly less memory than intended at the cost of sacrificing some computational efficiency. Ethereum instead uses an algorithm called Dagger, a memory-hard proof of work based on moderately connected directed acyclic graphs (DAGs, hence the name), which, while far from optimal, has much stronger memory-hardness properties than anything else in use today: an estimated 512 MB of RAM will be required per thread.

Dagger is more fully specified here: http://vitalik.ca/ethereum/dagger.html

Transactions

A transaction is stored as:

```
[ nonce, receiving_address, value, fee, [ data item 0, data item 1 ... data item n ], v, r, s ]
```

`nonce` is the number of transactions already sent by that address, or an empty string if this is the first transaction. `(v,r,s)` is the raw Electrum-style signature of the transaction without the signature made with the private key corresponding to the sending address. From an Electrum-style signature (65 bytes) it is possible to extract the public key, and thereby the address, directly. Note that all transactions in Ethereum are valid; invalid transactions or transactions with insufficient balance simply have no effect.

Difficulty adjustment

Difficulty is adjusted by the formula:

```
D(genesis_block) = 2^36
D(block) =
    if block.timestamp >= block.parent.timestamp + 42: D(block.parent) - floor(D(block.parent) / 1024)
    else:                                              D(block.parent) + floor(D(block.parent) / 1024)
```

This stabilizes around a block time of 60 seconds automatically (note that at a block time of 60 seconds, 50% of blocks come within 42 seconds of the previous; `1 - 1/e = 63.22%` come within 60 seconds), and adjusts at a maximum rate of about 0.1% per block (~100% per 12 hours).

Transactions sent to the zero address (ie. whose hexadecimal representation is all zeroes) are a special type of transaction, creating a "contract".

Contracts

Here is where we get to the actually interesting part of the Ethereum protocol. In Ethereum, there are actually two types of entities that can generate and receive transactions: actual people (or bots, as cryptographic protocols cannot distinguish between the two) and contracts. A contract is essentially an automated agent that lives on the Ethereum network, has an Ethereum address and balance, and can send and receive transactions. A contract is "activated" every time someone sends a transaction to it, at which point it runs its code, perhaps modifying its internal state or even sending some transactions, and then shuts down. The "code" for a contract is written in a special-purpose assembly language, executed in a virtual machine consisting of 256 registers, which are not persistent, and 2^256 memory entries, which constitute the contract's permanent state. The design principles behind contracts are as follows:

1. **Simplicity** - the Ethereum protocol should be as simple as possible, even at the cost of some efficiency. Any decent programmer should be able to re-implement it.
2. **Computational universality** - contracts can execute any function that anyone may want a contract to execute, and conditionally send out money to people based on the result of the calculations
3. **Size-universality** - contracts can exist for an arbitrarily long period of time and have arbitrarily many participants
4. **First class citizen property** - contracts can send and receive ether, make transactions (potentially to other contracts), read the state of other contracts and even create other contracts themselves
5. **Pigovian fee regulation** - the only mechanism for fighting spam or bloat is fees. You can run an infinite recursion bomb on top of Ethereum for as long as you are willing to keep feeding the contracts to pay for it
6. **Everything is a contract** - the contract is the basic data type of everything in the Ethereum network, except for ether itself. Want to make your own currency? Set it up as a contract. Want to make an order selling ether in exchange for units of another currency? Set up a contract to do that. Want to make a trust-free bet? Also a contract. Want to set up a full-scale Daemon or Skynet? Well, maybe you might want to have a few thousand interlocking contracts, and be sure to feed them generously, to do that, but nothing is stopping you. Ultimately, you may wish to even outsource some heavy computation to centralized parties by offering a bounty in the contract, using SCIP to verify the validity of the result; the sky(net) is the limit.

Examples of what contracts can do

Here are some examples of how a contract might work, written in high-level pseudocode:

1) **Simulate an entire currency as a single contract**. This is surprisingly easy to implement; the idea is that sending currency units requires sending a transaction to the contract with data item 0 as the recipient and data item 1 as the value. For a transaction to be valid, it must send 200000 finney to the contract in order to "feed" the contract (as each computational step after the first 16 for any contract costs a small fee)

```
if tx.value < 200000 finney: exit
if memory[1000]:
    from = tx.sender
    to = tx.data[0]
    value = tx.data[1]
    if to <= 1000: exit
    if memory[from] < value: exit
    memory[from] = memory[from] - value
    memory[to] = memory[to] + value
else:
    memory[mycreator] = 10000000000000000
    memory[1000] = 1
```

2) **Make a trust-free exchange offer** offering ether in exchange a contract-based currency as described in example 1 (call it "primegold") at a ratio of, say, 10 shamir for 1 primegold. Say the primegold contract's address is A and the address of the person making the exchange offer is B:

```
if tx.value < 2 ether: exit
valuesent = A.memory[myaddress]
send(myaddress,A,200000 finney,524288 finney,[B,valuesent])
send(myaddress,tx.sender,valuesent * 10485760,524288 finney,[])
```

In order to take the order securely, the counterparty would probably make the transaction sending primegold to the contract and then the transaction claiming the exchange offer inside a single Ethereum transaction one after the other.

3) A **centrally managed data feed**, where only the owner of OWNERADDRESS is allowed to change the data:

```
if tx.value < 100 finney: exit
if tx.sender != OWNERADDRESS: exit
memory[tx.data[0]] = tx.data[1]
```

4) A **hedging contract** offer, where another contract, ETORO, publishes the ether/USD price at index 10.

```
if tx.value < 262144 finney: exit
if memory[1000] == 0:
    memory[1000] = 1
    memory[1001] = tx.sender
    memory[1002] = tx.value
else if memory[1000] == 1:
    if tx.value < memory[1002]: exit // Enforce a minimum 1:1 deposit ratio
    memory[1000] = 2
    memory[1003] = ETORO[10] * memory[1002] // Store the USD amount hedged
    memory[1004] = curtime
    memory[1002] += tx.value // Store the total ether amount in the contract
else if memory[1000] == 2:
    if memory[1002] * ETORO[10] < memory[1003]:
        send(myaddress, memory[500], memory[1002], 1 ether, [])
        exit
    if time > memory[1002] + 86400 * 14:
        value = memory[1003] / ETORO[10] // Calculate the USD amount hedged in ether
        send(myaddress, memory[500], value, 524288 finney, [])
        send(myaddress, memory[1003], memory[1002] - value, 524288 finney, [])
```

The contract has three stages. In stage 0, the contract is doing nothing waiting for someone to claim it. In stage 1, a party wanting to hedge their funds has claimed the contract. In stage 2, a counterparty is found. At that point, the contract stores the USD value of the amount that party 1 has claimed, and locks in both sides' funds. If the value of ether drops so low that there is not enough ether to pay even party 1 the entire USD value hedged, party 1 immediately has the right to take out everything that they can get. Otherwise, after two weeks, party 1 gets back the same USD value in ether that they put in, and party 2 gets the rest. This type of contract can essentially be used to store a fixed amount of USD (or EUR, or gold) on the blockchain *without any central issuers*.

5) A **Namecoin**-style decentralized domain name system - the implementation would essentially be a hybrid of examples 1 and 3.

6) A **reputation system** and potentially even the groundwork for a social network, as a natural extension of a decentralized DNS.

7) A "one wei invested, one vote" **shareholder-run corporation** where decisions on where to move funds can be made by a quorum of investors (and the contract can accept new investors automatically)

8) A democratically run, **self-managed community** where decisions on where to move funds can be made by 51% of members, and adding a member requires the permission of 67% of existing members

9) **Crop insurace**. How? Simple - a contract for difference using a data feed of the weather instead of any price index.

10) **Generic insurance**, relying on one of a specified set of third-party judges to adjudicate claims. The contract can even include a complex appeals process if so desired.

11) A **decentrally managed data feed**, using proof-of-stake voting to give an average (or more likely, median) of everyone's opinion on the price of a commodity, the weather or any other relevant data

12) An offer to participate in a cryptographically secure, trust-free **peer-to-peer bet**

13) **SatoshiDice**. As in, the entire gambling site.

14) A full-scale **on-chain stock market**

15) An **on-chain decentralized marketplace**, with escrow, persistent identities and a rating system all built in

16) A **self-modifying code** based version of any of the above

How do contracts work?

A contract making transaction is encoded as follows:

```
[
    nonce,
    '',
    endowment,
    fee,
    [
        data item 0,
        data item 1,
        ...
    ],
    v,
    r,
    s
]
```

The data items will, in most cases, be script codes (more on this below). Contract creation transaction validation happens as follows:

1. Deserialize the transaction, and extract its sending address from its signature.
2. Check that the balance of the creator is at least the endowment plus the fee. If not, exit.
3. Check that the fee is at least `NEWCONTRACTFEE + MEMORYFEE * number of data items`. If not, exit.
4. Take the last 20 bytes of the hash of the transaction making the contract. If a contract with that address already exists, exit. Otherwise, create the contract at that address
5. Copy data item `i` to memory slot `i` for all `i` in `[0 ... n-1]` in the contract.

With `Rx` being shorthand for "the value at register `x`", and `M[x]` for "the value at memory location `x`", the assembly language has the following commands:

- `(00) STOP` - halts execution
- `(10) ADD Rx Ry Rz` - sets `Rz <- Rx + Ry mod 2^256`
- `(11) SUB Rx Ry Rz` - sets `Rz <- Rx - Ry mod 2^256`
- `(12) MUL Rx Ry Rz` - sets `Rz <- Rx * Ry mod 2^256`
- `(13) DIV Rx Ry Rz` - sets `Rz <- floor(Rx / Ry)`
- `(14) SDIV Rx Ry Rz` - like `DIV`, except it treats values above 2^255 as negative (ie. `2^256 - x -> -x`)
- `(15) MOD Rx Ry Rz` - sets `Rz <- Rx mod Ry`
- `(16) SMOD Rx Ry Rz` - like `MOD`, but for signed values just like `SDIV` (using Python's convention with negative numbers)
- `(17) EXP Rx Ry Rz` - sets `Rz <- Rx ^ Ry mod 2^256`
- `(18) NEG Rx Ry` - sets `Ry <- 2^256 - Rx`
- `(20) LT Rx Ry Rz` - sets `Rz <- 1` if `Rx < Ry` else `0`
- `(21) LE Rx Ry Rz` - sets `Rz <- 1` if `Rx <= Ry` else `0`
- `(22) GT Rx Ry Rz` - sets `Rz <- 1` if `Rx > Ry` else `0`
- `(23) GE Rx Ry Rz` - sets `Rz <- 1` if `Rx >= Ry` else `0`
- `(24) EQ Rx Ry Rz` - sets `Rz <- 1` if `Rx = Ry` else `0`
- `(25) NOT Rx Ry` - sets `Ry <- 1` if `Rx = 0` else `0`
- `(30) SHA256 Rx Ry` - sets `Ry <- SHA256(Rx)`
- `(31) RIPEMD160 Rx Ry` - sets `Ry <- RIPEMD160(Rx)`
- `(32) ECMUL Rx Ry Rz Ra Rb` - sets `(Ra, Rb) = Rz * (Rx, Ry)` in secp256k1, using `(0,0)` for the point at infinity
- `(33) ECADD Rx Ry Rz Ra Rb Rc` - sets `(Rb, Rc) = (Rx, Ry) + (Ra, Rb)`
- `(34) ECSIGN Rx Ry Rz Ra Rb - sets`(Rz, Ra, Rb)`as the`(r,s,prefix)`values of an Electrum-style RFC6979 deterministic signature of`Rx`with private key`Ry`
- `(35) ECRECOVER Rx Ry Rz Ra Rb Rc - sets`(Rb, Rc)`as the public key from the signature`(Ry, Rz, Ra)`of the message hash`Rx`
- `(40) COPY Rx Ry` - copies `Ry <- Rx`
- `(41) STORE Rx Ry` - sets `M[Ry] <- Rx`
- `(42) LOAD Rx Ry` - sets `Ry <- M[Rx]`
- `(43) SET Rx V1 V2 V3 V4` - sets `Rx <- V1 + 2^8*V2 + 2^16*V3 + 2^24*V4` (where `0 <= V[i] <= 255`)
- `(50) JMP Rx` - sets the index pointer to the value at Rx
- `(51) JMPI Rx Ry` - if `Rx != 0`, sets the index pointer to `Ry`
- `(52) IND Rx` - sets `Rx` to the index pointer.
- `(60) EXTRO Rx Ry Rz` - looks at the contract at address `Rx` and its memory state `Ry`, and outputs the result to `Rz`
- `(61) BALANCE Rx` - returns the ether balance of address `Rx`
- `(70) MKTX Rx Ry Rz Rw Rv` - sends `Ry` ether to `Rx` plus `Rz` fee with `Rw` data items starting from memory index `Rv` (and then reading to `(Rv + 1)`, `(Rv + 2)` etc). Note that if `Rx = 0` then this creates a new contract.
- `(80) DATA Rx Ry` - sets `Ry` to data item index Rx if possible, otherwise zero
- `(81) DATAN Rx` - sets `Rx` to the number of data items
- `(90) MYADDRESS Rx` - sets `Rx` to the contract's own address
- `(91) BLKHASH Rx` - sets `Rx` to the hash of the parent block
- `(92) COINBASE Rx` - sets `Rx` to the coinbase address of the current block
- `(ff) SUICIDE Rx` - destroys the contract and clears all memory, sending the entire balance plus the negative fee from clearing memory minus `TXFEE` to the address at `Rx`

The serialization of any script code is: `opcode + field1 * 256 + field2 * 256^2 + field3 * 256^3 + field4 * 256^4 + field5 * 256^5`. For example, `GE R10 R20 R30` would serialize into the integer `30201009` in hex, or 807407625 in decimal.

Whenever ether is sent to a script, the following happens:

1. The ether's endowment increases by the amount sent
2. All registers are reset to zero.
3. The sender is placed into R0.
4. The value sent is placed into R1.
5. The fee is placed into R2.
6. The index pointer is set to zero, and `STEPCOUNT = 0`
7. Repeat forever:
    - set `TOTALFEE = 0`
    - set `STEPCOUNT <- STEPCOUNT + 1`
    - if `STEPCOUNT > 16`, set `TOTALFEE <- TOTALFEE + STEPFEE`
    - see if the command at the index pointer is a valid command and not `STOP`. If it is invalid or `STOP`, HALT and break out of the loop
    - see if the command is LOAD or STORE. If so, set `TOTALFEE <- TOTALFEE + DATAFEE`
    - see if the command will fill up a previously zero memory field. If so, set `TOTALFEE <- TOTALFEE + MEMORYFEE`
    - see if the commend will zero a previously used memory field. If so, set `TOTALFEE <- TOTALFEE - MEMORYFEE`
    - see if the command is `EXTRO` or `BALANCE`. If so, set `TOTALFEE <- TOTALFEE + EXTROFEE`
    - see if the command is a crypto operation. If so, set `TOTALFEE <- TOTALFEE + CRYPTOFEE`
    - if `TOTALFEE > contract's endowment`, HALT and break out of the loop
    - else, subtract `TOTALFEE` from contract's endowment. Note that `TOTALFEE` may be negative in some cases, in which case the endowment would actually increase
    - run the command

Note that all fees are paid to miners, except memory fees; memory fees are taken out of circulation until recovered.

Assembly language example (currency as a contract)

First, we check for a transaction fee of at least 100 finney sent to the contract:

```
0.  SET R10 6
    LOAD R10 R10
    LT R10 R1 R20
    SET R255 7
    JMPI R20 R255
5.  STOP
    0640000000000000000
```

In memory slot 6, we put the value of 100 finney. We first load 6 into register 10, and then load the value at memory slot 6 (ie. 100 finney) into register 10. If the amount send to the contract (R1) is greater than or equal to 100 finney, we jump to spot 7. Otherwise, we walk into a STOP code.

Now, we check if the currency has been created yet, which we mark using memory slot 200. If it has not yet been created, we give `2^48` units to `e21cf4e5a0b919718d907e328a73f6991f721163` as the initial distribution.

```
    SET R30 200
    LOAD R30 R31
    SET R255 22
10. JMPI R31 R255
    SET R255 15
    JMP R255
    e21cf4e5a0b919718d907e328a73f6991f721163
    01000000000000
15. SET R40 10
    LOAD R40 R40
    SET R41 11
    LOAD R41 R41
    STORE R41 R40
20. SET R50 1
    STORE R50 R30
```

We load 200 into register 30, then the value at memory slot 200 into register 31, and if that value is nonzero we skip the entire section and jump to memory slot 19 (the `ST R50 R30` at the end is memory slot 18). Otherwise, we load the address into register 40, and value 2^48 into register 41, and then store 2^48 in memory slot `e21cf4e5a0b919718d907e328a73f6991f721163`. We then store memory slot 1 in 200.

Now, we'll load some data into registers for convenience:

```
    LOAD R0 R101
    DATA 0 R102
    LOAD R102 R103
25. DATA 1 R104
```

After this, we have the sender's balance in R101, the receiver's address in R102, the receiver's balance in R103 and the amount to send in R104. Then, we check if the sender has enough to send, and if not we jump to the stop code:

```
    LT R101 R104 R20
    JMPI R20 4
```

We then make another security check where we make sure the receiving address is not less than or equal to 200 (which we already have stored in R30), so that it does not overwrite any code:

```
    LT R30 R102 R20
    NOT R20 R20
    JMPI R20 4
```

And finally:

```
30. SUB R101 R104 R101
    ADD R103 R104 R103
    STORE R101 R0
    STORE R103 R102
    STOP
```

If you've been following everything up to now, you'll be able to figure this part out yourself.

Conclusion

The Ethereum protocol's design philosophy is in many ways the opposite from that taken by many other cryptocurrencies today. Other cryptocurrencies aim to add complexity and increase the number of "features"; Ethereum, on the other hand, takes features away. The protocol does not "support" multisignature transactions, multiple inputs and outputs, hash codes, lock times or many other features that even Bitcoin provides. Instead, all complexity comes from an all-powerful, Turing-complete assembly language, which can be used to build up literally any feature that is mathematically describable. The language itself follows an Orwellian Newspeak principle; any instruction which can be replaced by a sequence of less than four other instructions has been removed. As a result, we have a cryptocurrency protocol whose codebase is very small, and yet which can do anything that any cryptocurrency will ever be able to do.