Telegram Open Network Virtual Machine

Nikolai Durov

September 5, 2018

Abstract

The aim of this text is to provide a description of the Telegram Open Network Virtual Machine (TON VM or TVM), used to execute smart contracts in the TON Blockchain.

Introduction

The primary purpose of the Telegram Open Network Virtual Machine (TON VM or TVM) is to execute smart-contract code in the TON Blockchain. TVM must support all operations required to parse incoming messages and persistent data, and to create new messages and modify persistent data.

Additionally, TVM must meet the following requirements:

- It must provide for possible future extensions and improvements while retaining backward compatibility and interoperability, because the code of a smart contract, once committed into the blockchain, must continue working in a predictable manner regardless of any future modifications to the VM.
- It must strive to attain high "(virtual) machine code" density, so that the code of a typical smart contract occupies as little persistent blockchain storage as possible.
- It must be completely deterministic. In other words, each run of the same code with the same input data must produce the same result,

regardless of specific software and hardware used.[1]

The design of TVM is guided by these requirements. While this document describes a preliminary and experimental version of TVM,[2] the backward compatibility mechanisms built into the system allow us to be relatively unconcerned with the efficiency of the operation encoding used for TVM code in this preliminary version.

TVM is not intended to be implemented in hardware (e.g., in a specialized microprocessor chip); rather, it should be implemented in software running on conventional hardware. This consideration lets us incorporate some high-level concepts and operations in TVM that would require convoluted microcode in a hardware implementation but pose no significant problems for a software implementation. Such operations are useful for achieving high code density and minimizing the byte (or storage cell) profile of smart-contract code when deployed in the TON Blockchain.

[1] For example, there are no floating-point arithmetic operations (which could be efficiently implemented using hardware-supported *double* type on most modern CPUs) present in TVM, because the result of performing such operations is dependent on the specific underlying hardware implementation and rounding mode settings. Instead, TVM supports special integer arithmetic operations, which can be used to simulate fixed-point arithmetic if needed.

[2] The production version will likely require some tweaks and modifications prior to launch, which will become apparent only after using the experimental version in the test environment for some time.

Contents

1 Overview

This chapter provides an overview of the main features and design principles of TVM. More detail on each topic is provided in subsequent chapters.

1.0 Notation for bitstrings

The following notation is used for bit strings (or *bitstrings*)—i.e., finite strings consisting of binary digits (bits), `0` and `1`—throughout this document.

1.0.1. Hexadecimal notation for bitstrings. When the length of a bitstring is a multiple of four, we subdivide it into groups of four bits and represent each group by one of sixteen hexadecimal digits `0`–`9`, `A`–`F` in the usual manner: $\texttt{0}_{16} \leftrightarrow \texttt{0000}$, $\texttt{1}_{16} \leftrightarrow \texttt{0001}$, ..., $\texttt{F}_{16} \leftrightarrow \texttt{1111}$. The resulting hexadecimal string is our equivalent representation for the original binary string.

1.0.2. Bitstrings of lengths not divisible by four. If the length of a binary string is not divisible by four, we augment it by one `1` and several (maybe zero) `0`s at the end, so that its length becomes divisible by four, and then transform it into a string of hexadecimal digits as described above. To indicate that such a transformation has taken place, a special "completion tag" `_` is added to the end of the hexadecimal string. The reverse transformation (applied if the completion tag is present) consists in first replacing each hexadecimal digit by four corresponding bits, and then removing all trailing zeroes (if any) and the last `1` immediately preceding them (if the resulting bitstring is non-empty at this point).

Notice that there are several admissible hexadecimal representations for the same bitstring. Among them, the shortest one is "canonical". It can be deterministically obtained by the above procedure.

For example, `8A` corresponds to binary string `10001010`, while `8A_` and `8A0_` both correspond to `100010`. An empty bitstring may be represented by either '', '`8_`', '`0_`', '`_`', or '`00_`'.

1.0.3. Emphasizing that a string is a hexadecimal representation of a bitstring. Sometimes we need to emphasize that a string of hexadecimal digits (with or without a `_` at the end) is the hexadecimal representation of a bitstring. In such cases, we either prepend `x` to the resulting string (e.g., `x8A`), or prepend `x{` and append `}` (e.g., `x{2D9_}`, which is `00101101100`).

This should not be confused with hexadecimal numbers, usually prepended by `0x` (e.g., `0x2D9` or `0x2d9`, which is the integer 729).

1.0.4. Serializing a bitstring into a sequence of octets. When a bitstring needs to be represented as a sequence of 8-bit bytes (octets), which take values in integers $0 \dots 255$, this is achieved essentially in the same fashion as above: we split the bitstring into groups of eight bits and interpret each group as the binary representation of an integer $0 \dots 255$. If the length of the bitstring is not a multiple of eight, the bitstring is augmented by a binary `1` and up to seven binary `0`s before being split into groups. The fact that such a completion has been applied is usually reflected by a "completion tag" bit.

For instance, `00101101100` corresponds to the sequence of two octets (`0x2d`, `0x90`) (hexadecimal), or $(45, 144)$ (decimal), along with a completion tag bit equal to `1` (meaning that the completion has been applied), which must be stored separately.

In some cases, it is more convenient to assume the completion is enabled by default rather than store an additional completion tag bit separately. Under such conventions, $8n$-bit strings are represented by $n + 1$ octets, with the last octet always equal to `0x80` $= 128$.

1.1 TVM is a stack machine

First of all, *TVM is a stack machine.* This means that, instead of keeping values in some "variables" or "general-purpose registers", they are kept in a (LIFO) *stack*, at least from the "low-level" (TVM) perspective.[3]

Most operations and user-defined functions take their arguments from the top of the stack, and replace them with their result. For example, the integer addition primitive (built-in operation) `ADD` does not take any arguments describing which registers or immediate values should be added together and where the result should be stored. Instead, the two top values are taken from the stack, they are added together, and their sum is pushed into the stack in their place.

[3] A high-level smart-contract language might create a visibility of variables for the ease of programming; however, the high-level source code working with variables will be translated into TVM machine code keeping all the values of these variables in the TVM stack.

1.1.1. TVM values. The entities that can be stored in the TVM stack will be called *TVM values*, or simply *values* for brevity. They belong to one of several predefined *value types*. Each value belongs to exactly one value type. The values are always kept on the stack along with tags uniquely determining their types, and all built-in TVM operations (or *primitives*) only accept values of predefined types.

For example, the integer addition primitive `ADD` accepts only two integer values, and returns one integer value as a result. One cannot supply `ADD` with two strings instead of two integers expecting it to concatenate these strings or to implicitly transform the strings into their decimal integer values; any attempt to do so will result in a run-time type-checking exception.

1.1.2. Static typing, dynamic typing, and run-time type checking. In some respects TVM performs a kind of dynamic typing using run-time type checking. However, this does not make the TVM code a "dynamically typed language" like PHP or Javascript, because all primitives accept values and return results of predefined (value) types, each value belongs to strictly one type, and values are never implicitly converted from one type to another. If, on the other hand, one compares the TVM code to the conventional microprocessor machine code, one sees that the TVM mechanism of value tagging prevents, for example, using the address of a string as a number—or, potentially even more disastrously, using a number as the address of a string—thus eliminating the possibility of all sorts of bugs and security vulnerabilities related to invalid memory accesses, usually leading to memory corruption and segmentation faults. This property is highly desirable for a VM used to execute smart contracts in a blockchain. In this respect, TVM's insistence on tagging all values with their appropriate types, instead of reinterpreting the bit sequence in a register depending on the needs of the operation it is used in, is just an additional run-time type-safety mechanism.

An alternative would be to somehow analyze the smart-contract code for type correctness and type safety before allowing its execution in the VM, or even before allowing it to be uploaded into the blockchain as the code of a smart contract. Such a static analysis of code for a Turing-complete machine appears to be a time-consuming and non-trivial problem (likely to be equivalent to the stopping problem for Turing machines), something we would rather avoid in a blockchain smart-contract context.

One should bear in mind that one always can implement compilers from statically typed high-level smart-contract languages into the TVM code (and

we do expect that most smart contracts for TON will be written in such languages), just as one can compile statically typed languages into conventional machine code (e.g., x86 architecture). If the compiler works correctly, the resulting machine code will never generate any run-time type-checking exceptions. All type tags attached to values processed by TVM will always have expected values and may be safely ignored during the analysis of the resulting TVM code, apart from the fact that the run-time generation and verification of these type tags by TVM will slightly slow down the execution of the TVM code.

1.1.3. Preliminary list of value types. A preliminary list of value types supported by TVM is as follows:

- *Integer* — Signed 257-bit integers, representing integer numbers in the range $-2^{256} \ldots 2^{256} - 1$, as well as a special "not-a-number" value `NaN`.
- *Cell* — A *TVM cell* consists of at most 1023 bits of data, and of at most four references to other cells. All persistent data (including TVM code) in the TON Blockchain is represented as a collection of TVM cells (cf. [1, 2.5.14]).
- *Slice* — A *TVM cell slice*, or *slice* for short, is a contiguous "sub-cell" of an existing cell, containing some of its bits of data and some of its references. Essentially, a slice is a read-only view for a subcell of a cell. Slices are used for unpacking data previously stored (or serialized) in a cell or a tree of cells.
- *Builder* — A *TVM cell builder*, or *builder* for short, is an "incomplete" cell that supports fast operations of appending bitstrings and cell references at its end. Builders are used for packing (or serializing) data from the top of the stack into new cells (e.g., before transferring them to persistent storage).
- *Continuation* — Represents an "execution token" for TVM, which may be invoked (executed) later. As such, it generalizes function addresses (i.e., function pointers and references), subroutine return addresses, instruction pointer addresses, exception handler addresses, closures, partial applications, anonymous functions, and so on.

This list of value types is incomplete and may be extended in future revisions of TVM without breaking the old TVM code, due mostly to the fact that

all originally defined primitives accept only values of types known to them and will fail (generate a type-checking exception) if invoked on values of new types. Furthermore, existing value types themselves can also be extended in the future: for example, 257-bit *Integer* might become 513-bit *LongInteger*, with originally defined arithmetic primitives failing if either of the arguments or the result does not fit into the original subtype *Integer*. Backward compatibility with respect to the introduction of new value types and extension of existing value types will be discussed in more detail later (cf. **5.1.4**).

1.2 Categories of TVM instructions

TVM *instructions*, also called *primitives* and sometimes *(built-in) operations*, are the smallest operations atomically performed by TVM that can be present in the TVM code. They fall into several categories, depending on the types of values (cf. **1.1.3**) they work on. The most important of these categories are:

- *Stack (manipulation) primitives* — Rearrange data in the TVM stack, so that the other primitives and user-defined functions can later be called with correct arguments. Unlike all other primitives, they work with values of arbitrary types.
- *Constant* or *literal primitives* — Push into the stack some "constant" or "literal" values embedded into the TVM code itself, thus providing arguments to the other primitives. They are somewhat similar to stack primitives, but are less generic because they work with values of specific types.
- *Arithmetic primitives* — Perform the usual integer arithmetic operations on values of type *Integer*.
- *Cell (manipulation) primitives* — Create new cells and store data in them (*cell creation primitives*) or read data from previously created cells (*cell parsing primitives*). Because all memory and persistent storage of TVM consists of cells, these cell manipulation primitives actually correspond to "memory access instructions" of other architectures. Cell creation primitives usually work with values of type *Builder*, while cell parsing primitives work with *Slice*s.

- *Continuation* and *control flow primitives* — Create and modify *Continuation*s, as well as execute existing *Continuation*s in different ways, including conditional and repeated execution.
- *Custom* or *application-specific primitives* — Efficiently perform specific high-level actions required by the application (in our case, the TON Blockchain), such as computing hash functions, performing elliptic curve cryptography, sending new blockchain messages, creating new smart contracts, and so on. These primitives correspond to standard library functions rather than microprocessor instructions.

1.3 Control registers

While TVM is a stack machine, some rarely changed values needed in almost all functions are better passed in certain special registers, and not near the top of the stack. Otherwise, a prohibitive number of stack reordering operations would be required to manage all these values.

To this end, the TVM model includes, apart from the stack, up to 16 special *control registers*, denoted by `c0` to `c15`, or `c`(0) to `c`(15). The original version of TVM makes use of only some of these registers; the rest may be supported later.

1.3.1. Values kept in control registers. The values kept in control registers are of the same types as those kept on the stack. However, some control registers accept only values of specific types, and any attempt to load a value of a different type will lead to an exception.

1.3.2. List of control registers. The original version of TVM defines and uses the following control registers:

- `c0` — Contains the *next continuation* or *return continuation* (similar to the subroutine return address in conventional designs). This value must be a *Continuation*.
- `c1` — Contains the *alternative (return) continuation*; this value must be a *Continuation*. It is used in some (experimental) control flow primitives, allowing TVM to define and call "subroutines with two exit points".
- `c2` — Contains the *exception handler*. This value is a *Continuation*, invoked whenever an exception is triggered.

- `c3` — Contains the *current dictionary*, essentially a hashmap containing the code of all functions used in the program. For reasons explained later in **4.6**, this value is also a *Continuation*, not a *Cell* as one might expect.
- `c4` — Contains the *root of persistent data*, or simply the *data*. This value is a *Cell*. When the code of a smart contract is invoked, `c4` points to the root cell of its persistent data kept in the blockchain state. If the smart contract needs to modify this data, it changes `c4` before returning.
- `c5` — Contains the *root of temporary data*. It is a *Cell*, initialized by a reference to an empty cell before invoking the smart contract and discarded after its termination.
- `c6` — Contains the *output actions*. It is also a *Cell* initialized by a reference to an empty cell, but its final value is considered one of the smart contract outputs. For instance, the `SENDMSG` primitive, specific for the TON Blockchain, simply inserts the message into a list stored in the output actions.

More control registers may be defined in the future for specific TON Blockchain or high-level programming language purposes, if necessary.

1.4 Total state of TVM (SCCCG)

The total state of TVM consists of the following components:

- *Stack* (cf. **1.1**) — Contains zero or more *values* (cf. **1.1.1**), each belonging to one of *value types* listed in **1.1.3**.
- *Control registers* `c0`–`c15` — Contain some specific values as described in **1.3.2**. (Only seven control registers are used in the current version.)
- *Current continuation* `cc` — Contains the current continuation (i.e., the code that would be normally executed after the current primitive is completed). This component is similar to the instruction pointer register (`ip`) in other architectures.

- *Current codepage* `cp` — A special signed 16-bit integer value that selects the way the next TVM opcode will be decoded. For example, future versions of TVM might use different codepages to add new opcodes while preserving backward compatibility.

- *Gas limits* `gas` — Contains four signed 64-bit integers: the current gas limit g_l, the maximal gas limit g_m, the remaining gas g_r, and the gas credit g_c. Always $0 \leq g_l \leq g_m$, $g_c \geq 0$, and $g_r \leq g_l + g_c$; g_c is usually initialized by zero, g_r is initialized by $g_l + g_c$ and gradually decreases as the TVM runs. When g_r becomes negative or if the final value of g_r is less than g_c, an *out of gas* exception is triggered.

Notice that there is no "return stack" containing the return addresses of all previously called but unfinished functions. Instead, only control register `c0` is used. The reason for this will be explained later in **4.1.9**.

Also notice that there are no general-purpose registers, because TVM is a stack machine (cf. **1.1**). So the above list, which can be summarized as "stack, control, continuation, codepage, and gas" (SCCCG), similarly to the classical SECD machine state ("stack, environment, control, dump"), is indeed the *total* state of TVM.[4]

1.5 Integer arithmetic

All arithmetic primitives of TVM operate on several arguments of type *Integer*, taken from the top of the stack, and return their results, of the same type, into the stack. Recall that *Integer* represents all integer values in the range $-2^{256} \leq x < 2^{256}$, and additionally contains a special value `NaN` ("not-a-number").

If one of the results does not fit into the supported range of integers—or if one of the arguments is a `NaN`—then this result or all of the results are replaced by a `NaN`, and (by default) an integer overflow exception is generated. However, special "quiet" versions of arithmetic operations will simply produce `NaN`s and keep going. If these `NaN`s end up being used in a "non-quiet" arithmetic operation, or in a non-arithmetic operation, an integer overflow exception will occur.

[4]Strictly speaking, there is also the current *library context*, which consists of a dictionary with 256-bit keys and cell values, used to load library reference cells of **3.1.7**.

1.5.1. Absence of automatic conversion of integers. Notice that TVM *Integer*s are "mathematical" integers, and not 257-bit strings interpreted differently depending on the primitive used, as is common for other machine code designs. For example, TVM has only one multiplication primitive `MUL`, rather than two (`MUL` for unsigned multiplication and `IMUL` for signed multiplication) as occurs, for example, in the popular x86 architecture.

1.5.2. Automatic overflow checks. Notice that all TVM arithmetic primitives perform overflow checks of the results. If a result does not fit into the *Integer* type, it is replaced by a `NaN`, and (usually) an exception occurs. In particular, the result is *not* automatically reduced modulo 2^{256} or 2^{257}, as is common for most hardware machine code architectures.

1.5.3. Custom overflow checks. In addition to automatic overflow checks, TVM includes custom overflow checks, performed by primitives `FITS` n and `UFITS` n, where $1 \leq n \leq 256$. These primitives check whether the value on (the top of) the stack is an integer x in the range $-2^{n-1} \leq x < 2^{n-1}$ or $0 \leq x < 2^n$, respectively, and replace the value with a `NaN` and (optionally) generate an integer overflow exception if this is not the case. This greatly simplifies the implementation of arbitrary n-bit integer types, signed or unsigned: the programmer or the compiler must insert appropriate `FITS` or `UFITS` primitives either after each arithmetic operation (which is more reasonable, but requires more checks) or before storing computed values and returning them from functions. This is important for smart contracts, where unexpected integer overflows happen to be among the most common source of bugs.

1.5.4. Reduction modulo 2^n. TVM also has a primitive `MODPOW2` n, which reduces the integer at the top of the stack modulo 2^n, with the result ranging from 0 to $2^n - 1$.

1.5.5. *Integer* is 257-bit, not 256-bit. One can understand now why TVM's *Integer* is (signed) 257-bit, not 256-bit. The reason is that it is the smallest integer type containing both signed 256-bit integers and unsigned 256-bit integers, which does not require automatic reinterpreting of the same 256-bit string depending on the operation used (cf. **1.5.1**).

1.5.6. Division and rounding. The most important division primitives are `DIV`, `MOD`, and `DIVMOD`. All of them take two numbers from the stack, x and y (y is taken from the top of the stack, and x is originally under it),

compute the quotient q and remainder r of the division of x by y (i.e., two integers such that $x = yq + r$ and $|r| < |y|$), and return either q, r, or both of them. If y is zero, then all of the expected results are replaced by `NaN`s, and (usually) an integer overflow exception is generated.

The implementation of division in TVM somewhat differs from most other implementations with regards to rounding. By default, these primitives round to $-\infty$, meaning that $q = \lfloor x/y \rfloor$, and r has the same sign as y. (Most conventional implementations of division use "rounding to zero" instead, meaning that r has the same sign as x.) Apart from this "floor rounding", two other rounding modes are available, called "ceiling rounding" (with $q = \lceil x/y \rceil$, and r and y having opposite signs) and "nearest rounding" (with $q = \lfloor x/y + 1/2 \rfloor$ and $|r| \leq |y|/2$). These rounding modes are selected by using other division primitives, with letters `C` and `R` appended to their mnemonics. For example, `DIVMODR` computes both the quotient and the remainder using rounding to the nearest integer.

1.5.7. Combined multiply-divide, multiply-shift, and shift-divide operations. To simplify implementation of fixed-point arithmetic, TVM supports combined multiply-divide, multiply-shift, and shift-divide operations with double-length (i.e., 514-bit) intermediate product. For example, `MULDIVMODR` takes three integer arguments from the stack, a, b, and c, first computes ab using a 514-bit intermediate result, and then divides ab by c using rounding to the nearest integer. If c is zero or if the quotient does not fit into *Integer*, either two `NaN`s are returned, or an integer overflow exception is generated, depending on whether a quiet version of the operation has been used. Otherwise, both the quotient and the remainder are pushed into the stack.

2 The stack

This chapter contains a general discussion and comparison of register and stack machines, expanded further in Appendix **C**, and describes the two main classes of stack manipulation primitives employed by TVM: the *basic* and the *compound stack manipulation primitives*. An informal explanation of their sufficiency for all stack reordering required for correctly invoking other primitives and user-defined functions is also provided. Finally, the problem of efficiently implementing TVM stack manipulation primitives is discussed in **2.3**.

2.1 Stack calling conventions

A stack machine, such as TVM, uses the stack—and especially the values near the top of the stack—to pass arguments to called functions and primitives (such as built-in arithmetic operations) and receive their results. This section discusses the TVM stack calling conventions, introduces some notation, and compares TVM stack calling conventions with those of certain register machines.

2.1.1. Notation for "stack registers". Recall that a stack machine, as opposed to a more conventional register machine, lacks general-purpose registers. However, one can treat the values near the top of the stack as a kind of "stack registers".

We denote by `s0` or `s`(0) the value at the top of the stack, by `s1` or `s`(1) the value immediately under it, and so on. The total number of values in the stack is called its *depth*. If the depth of the stack is n, then `s`(0), `s`(1), ..., `s`$(n-1)$ are well-defined, while `s`(n) and all subsequent `s`(i) with $i > n$ are not. Any attempt to use `s`(i) with $i \geq n$ should produce a stack underflow exception.

A compiler, or a human programmer in "TVM code", would use these "stack registers" to hold all declared variables and intermediate values, similarly to the way general-purpose registers are used on a register machine.

2.1.2. Pushing and popping values. When a value x is *pushed* into a stack of depth n, it becomes the new `s0`; at the same time, the old `s0` becomes the new `s1`, the old `s1`—the new `s2`, and so on. The depth of the resulting stack is $n + 1$.

Similarly, when a value x is *popped* from a stack of depth $n \geq 1$, it is the old value of `s0` (i.e., the old value at the top of the stack). After this, it is removed from the stack, and the old `s1` becomes the new `s0` (the new value at the top of the stack), the old `s2` becomes the new `s1`, and so on. The depth of the resulting stack is $n - 1$.

If originally $n = 0$, then the stack is *empty*, and a value cannot be popped from it. If a primitive attempts to pop a value from an empty stack, a *stack underflow* exception occurs.

2.1.3. Notation for hypothetical general-purpose registers. In order to compare stack machines with sufficiently general register machines, we will denote the general-purpose registers of a register machine by `r0`, `r1`, and so on, or by `r`(0), `r`(1), ..., `r`$(n-1)$, where n is the total number of registers. When we need a specific value of n, we will use $n = 16$, corresponding to the very popular x86-64 architecture.

2.1.4. The top-of-stack register `s0` vs. the accumulator register `r0`. Some register machine architectures require one of the arguments for most arithmetic and logical operations to reside in a special register called the *accumulator*. In our comparison, we will assume that the accumulator is the general-purpose register `r0`; otherwise we could simply renumber the registers. In this respect, the accumulator is somewhat similar to the top-of-stack "register" `s0` of a stack machine, because virtually all operations of a stack machine both use `s0` as one of their arguments and return their result as `s0`.

2.1.5. Register calling conventions. When compiled for a register machine, high-level language functions usually receive their arguments in certain registers in a predefined order. If there are too many arguments, these functions take the remainder from the stack (yes, a register machine usually has a stack, too!). Some register calling conventions pass no arguments in registers at all, however, and only use the stack (for example, the original calling conventions used in implementations of Pascal and C, although modern implementations of C use some registers as well).

For simplicity, we will assume that up to $m \leq n$ function arguments are passed in registers, and that these registers are `r0`, `r1`, ..., `r`$(m-1)$, in that order (if some other registers are used, we can simply renumber them).[5]

[5]Our inclusion of `r0` here creates a minor conflict with our assumption that the ac-

2.1.6. Order of function arguments. If a function or primitive requires m arguments $x_1, \ldots, x_m$, they are pushed by the caller into the stack in the same order, starting from x_1. Therefore, when the function or primitive is invoked, its first argument x_1 is in $\mathtt{s}(m-1)$, its second argument x_2 is in $\mathtt{s}(m-2)$, and so on. The last argument x_m is in `s0` (i.e., at the top of the stack). It is the called function or primitive's responsibility to remove its arguments from the stack.

In this respect the TVM stack calling conventions—obeyed, at least, by TMV primitives—match those of Pascal and Forth, and are the opposite of those of C (in which the arguments are pushed into the stack in the reverse order, and are removed by the caller after it regains control, not the callee).

Of course, an implementation of a high-level language for TVM might choose some other calling conventions for its functions, different from the default ones. This might be useful for certain functions—for instance, if the total number of arguments depends on the value of the first argument, as happens for "variadic functions" such as `scanf` and `printf`. In such cases, the first one or several arguments are better passed near the top of the stack, not somewhere at some unknown location deep in the stack.

2.1.7. Arguments to arithmetic primitives on register machines. On a stack machine, built-in arithmetic primitives (such as `ADD` or `DIVMOD`) follow the same calling conventions as user-defined functions. In this respect, user-defined functions (for example, a function computing the square root of a number) might be considered as "extensions" or "custom upgrades" of the stack machine. This is one of the clearest advantages of stack machines (and of stack programming languages such as Forth) compared to register machines.

In contrast, arithmetic instructions (built-in operations) on register machines usually get their parameters from general-purpose registers encoded in the full opcode. A binary operation, such as `SUB`, thus requires two arguments, $\mathtt{r}(i)$ and $\mathtt{r}(j)$, with i and j specified by the instruction. A register $\mathtt{r}(k)$ for storing the result also must be specified. Arithmetic operations can take several possible forms, depending on whether i, j, and k are allowed to take arbitrary values:

- Three-address form — Allows the programmer to arbitrarily choose not only the two source registers $\mathtt{r}(i)$ and $\mathtt{r}(j)$, but also a separate

cumulator register, if present, is also `r0`; for simplicity, we will resolve this problem by assuming that the first argument to a function is passed in the accumulator.

destination register `r`(k). This form is common for most RISC processors, and for the XMM and AVX SIMD instruction sets in the x86-64 architecture.

- Two-address form — Uses one of the two operand registers (usually `r`(i)) to store the result of an operation, so that $k = i$ is never indicated explicitly. Only i and j are encoded inside the instruction. This is the most common form of arithmetic operations on register machines, and is quite popular on microprocessors (including the x86 family).

- One-address form — Always takes one of the arguments from the accumulator `r0`, and stores the result in `r0` as well; then $i = k = 0$, and only j needs to be specified by the instruction. This form is used by some simpler microprocessors (such as Intel 8080).

Note that this flexibility is available only for built-in operations, but not for user-defined functions. In this respect, register machines are not as easily "upgradable" as stack machines.[6]

2.1.8. Return values of functions. In stack machines such as TVM, when a function or primitive needs to return a result value, it simply pushes it into the stack (from which all arguments to the function have already been removed). Therefore, the caller will be able to access the result value through the top-of-stack "register" `s0`.

This is in complete accordance with Forth calling conventions, but differs slightly from Pascal and C calling conventions, where the accumulator register `r0` is normally used for the return value.

2.1.9. Returning several values. Some functions might want to return several values $y_1, \ldots, y_k$, with k not necessarily equal to one. In these cases, the k return values are pushed into the stack in their natural order, starting from y_1.

For example, the "divide with remainder" primitive `DIVMOD` needs to return two values, the quotient q and the remainder r. Therefore, `DIVMOD` pushes q and r into the stack, in that order, so that the quotient is available

[6]For instance, if one writes a function for extracting square roots, this function will always accept its argument and return its result in the same registers, in contrast with a hypothetical built-in square root instruction, which could allow the programmer to arbitrarily choose the source and destination registers. Therefore, a user-defined function is tremendously less flexible than a built-in instruction on a register machine.

thereafter at `s1` and the remainder at `s0`. The net effect of `DIVMOD` is to divide the original value of `s1` by the original value of `s0`, and return the quotient in `s1` and the remainder in `s0`. In this particular case the depth of the stack and the values of all other "stack registers" remain unchanged, because `DIVMOD` takes two arguments and returns two results. In general, the values of other "stack registers" that lie in the stack below the arguments passed and the values returned are shifted according to the change of the depth of the stack.

In principle, some primitives and user-defined functions might return a variable number of result values. In this respect, the remarks above about variadic functions (cf. **2.1.6**) apply: the total number of result values and their types should be determined by the values near the top of the stack. (For example, one might push the return values y_1, ..., y_k, and then push their total number k as an integer. The caller would then determine the total number of returned values by inspecting `s0`.)

In this respect TVM, again, faithfully observes Forth calling conventions.

2.1.10. Stack notation. When a stack of depth n contains values z_1, ..., z_n, in that order, with z_1 the deepest element and z_n the top of the stack, the contents of the stack are often represented by a list $z_1\ z_2\ \ldots z_n$, in that order. When a primitive transforms the original stack state S' into a new state S'', this is often written as $S' - S''$; this is the so-called *stack notation*. For example, the action of the division primitive `DIV` can be described by $S\ x\ y - S\ \lfloor x/y \rfloor$, where S is any list of values. This is usually abbreviated as $x\ y - \lfloor x/y \rfloor$, tacitly assuming that all other values deeper in the stack remain intact.

Alternatively, one can describe `DIV` as a primitive that runs on a stack S' of depth $n \geq 2$, divides `s1` by `s0`, and returns the floor-rounded quotient as `s0` of the new stack S'' of depth $n - 1$. The new value of $\mathtt{s}(i)$ equals the old value of $\mathtt{s}(i + 1)$ for $1 \leq i < n - 1$. These descriptions are equivalent, but saying that `DIV` transforms $x\ y$ into $\lfloor x/y \rfloor$, or $\ldots x\ y$ into $\ldots \lfloor x/y \rfloor$, is more concise.

The stack notation is extensively used throughout Appendix **A**, where all currently defined TVM primitives are listed.

2.1.11. Explicitly defining the number of arguments to a function. Stack machines usually pass the current stack in its entirety to the invoked primitive or function. That primitive or function accesses only the several values near the top of the stack that represent its arguments, and pushes the

return values in their place, by convention leaving all deeper values intact. Then the resulting stack, again in its entirety, is returned to the caller.

Most TVM primitives behave in this way, and we expect most user-defined functions to be implemented under such conventions. However, TVM provides mechanisms to specify how many arguments must be passed to a called function (cf. **4.1.10**). When these mechanisms are employed, the specified number of values are moved from the caller's stack into the (usually initially empty) stack of the called function, while deeper values remain in the caller's stack and are inaccessible to the callee. The caller can also specify how many return values it expects from the called function.

Such argument-checking mechanisms might be useful, for example, for a library function that calls user-provided functions passed as arguments to it.

2.2 Stack manipulation primitives

A stack machine, such as TVM, employs a lot of stack manipulation primitives to rearrange arguments to other primitives and user-defined functions, so that they become located near the top of the stack in correct order. This section discusses which stack manipulation primitives are necessary and sufficient for achieving this goal, and which of them are used by TVM. Some examples of code using these primitives can be found in Appendix **C**.

2.2.1. Basic stack manipulation primitives. The most important stack manipulation primitives used by TVM are the following:

- *Top-of-stack exchange operation*: `XCHG s0,s`(i) or `XCHG s`(i) — Exchanges values of `s0` and `s`(i). When $i = 1$, operation `XCHG s1` is traditionally denoted by `SWAP`. When $i = 0$, this is a `NOP` (an operation that does nothing, at least if the stack is non-empty).

- *Arbitrary exchange operation*: `XCHG s`(i)`,s`(j) — Exchanges values of `s`(i) and `s`(j). Notice that this operation is not strictly necessary, because it can be simulated by three top-of-stack exchanges: `XCHG s`(i); `XCHG s`(j); `XCHG s`(i). However, it is useful to have arbitrary exchanges as primitives, because they are required quite often.

- *Push operation*: `PUSH s`(i) — Pushes a copy of the (old) value of `s`(i) into the stack. Traditionally, `PUSH s0` is also denoted by `DUP` (it duplicates the value at the top of the stack), and `PUSH s1` by `OVER`.

- *Pop operation*: `POP` $s(i)$ — Removes the top-of-stack value and puts it into the (new) $\mathtt{s}(i-1)$, or the old $\mathtt{s}(i)$. Traditionally, `POP s0` is also denoted by `DROP` (it simply drops the top-of-stack value), and `POP s1` by `NIP`.

Some other "unsystematic" stack manipulation operations might be also defined (e.g., `ROT`, with stack notation $a\ b\ c - b\ c\ a$). While such operations are defined in stack languages like Forth (where `DUP`, `DROP`, `OVER`, `NIP` and `SWAP` are also present), they are not strictly necessary because the *basic stack manipulation primitives* listed above suffice to rearrange stack registers to allow any arithmetic primitives and user-defined functions to be invoked correctly.

2.2.2. Basic stack manipulation primitives suffice. A compiler or a human TVM-code programmer might use the basic stack primitives as follows.

Suppose that the function or primitive to be invoked is to be passed, say, three arguments x, y, and z, currently located in stack registers $\mathtt{s}(i)$, $\mathtt{s}(j)$, and $\mathtt{s}(k)$. In this circumstance, the compiler (or programmer) might issue operation `PUSH` $\mathtt{s}(i)$ (if a copy of x is needed after the call to this primitive) or `XCHG` $\mathtt{s}(i)$ (if it will not be needed afterwards) to put the first argument x into the top of the stack. Then, the compiler (or programmer) could use either `PUSH` $\mathtt{s}(j')$ or `XCHG` $\mathtt{s}(j')$, where $j' = j$ or $j+1$, to put y into the new top of the stack.[7]

Proceeding in this manner, we see that we can put the original values of x, y, and z—or their copies, if needed—into locations `s2`, `s1`, and `s0`, using a sequence of push and exchange operations (cf. **2.2.4** and **2.2.5** for a more detailed explanation). In order to generate this sequence, the compiler will need to know only the three values i, j and k, describing the old locations of variables or temporary values in question, and some flags describing whether each value will be needed thereafter or is needed only for this primitive or function call. The locations of other variables and temporary values will be affected in the process, but a compiler (or a human programmer) can easily track their new locations.

[7]Of course, if the second option is used, this will destroy the original arrangement of x in the top of the stack. In this case, one should either issue a `SWAP` before `XCHG` $\mathtt{s}(j')$, or replace the previous operation `XCHG` $\mathtt{s}(i)$ with `XCHG s1,` $\mathtt{s}(i)$, so that x is exchanged with `s1` from the beginning.

Similarly, if the results returned from a function need to be discarded or moved to other stack registers, a suitable sequence of exchange and pop operations will do the job. In the typical case of one return value in `s0`, this is achieved either by an `XCHG s(`i`)` or a `POP s(`i`)` (in most cases, a `DROP`) operation.[8]

Rearranging the result value or values before returning from a function is essentially the same problem as arranging arguments for a function call, and is achieved similarly.

2.2.3. Compound stack manipulation primitives. In order to improve the density of the TVM code and simplify development of compilers, compound stack manipulation primitives may be defined, each combining up to four exchange and push or exchange and pop basic primitives. Such compound stack operations might include, for example:

- `XCHG2 s(`i`),s(`j`)` — Equivalent to `XCHG s1,s(`i`); XCHG s(`j`)`.
- `PUSH2 s(`i`),s(`j`)` — Equivalent to `PUSH s(`i`); PUSH s(`$j+1$`)`.
- `XCPU s(`i`),s(`j`)` — Equivalent to `XCHG s(`i`); PUSH s(`j`)`.
- `PUXC s(`i`),s(`j`)` — Equivalent to `PUSH s(`i`); SWAP; XCHG s(`$j+1$`)`. When $j \neq i$ and $j \neq 0$, it is also equivalent to `XCHG s(`j`); PUSH s(`i`); SWAP`.
- `XCHG3 s(`i`),s(`j`),s(`k`)` — Equivalent to `XCHG s2,s(`i`); XCHG s1,s(`j`); XCHG s(`k`)`.
- `PUSH3 s(`i`),s(`j`),s(`k`)` — Equivalent to `PUSH s(`i`); PUSH s(`$j+1$`); PUSH s(`$k+2$`)`.

Of course, such operations make sense only if they admit a more compact encoding than the equivalent sequence of basic operations. For example, if all top-of-stack exchanges, `XCHG s1,s(`i`)` exchanges, and push and pop operations admit one-byte encodings, the only compound stack operations suggested above that might merit inclusion in the set of stack manipulation primitives are `PUXC`, `XCHG3`, and `PUSH3`.

[8]Notice that the most common `XCHG` $s(i)$ operation is not really required here if we do not insist on keeping the same temporary value or variable always in the same stack location, but rather keep track of its subsequent locations. We will move it to some other location while preparing the arguments to the next primitive or function call.

These compound stack operations essentially augment other primitives (instructions) in the code with the "true" locations of their operands, somewhat similarly to what happens with two-address or three-address register machine code. However, instead of encoding these locations inside the opcode of the arithmetic or another instruction, as is customary for register machines, we indicate these locations in a preceding compound stack manipulation operation. As already described in **2.1.7**, the advantage of such an approach is that user-defined functions (or rarely used specific primitives added in a future version of TVM) can benefit from it as well (cf. **C.3** for a more detailed discussion with examples).

2.2.4. Mnemonics of compound stack operations. The mnemonics of compound stack operations, some examples of which have been provided in **2.2.3**, are created as follows.

The $\gamma \geq 2$ formal arguments $\mathtt{s}(i_1)$, ..., $\mathtt{s}(i_\gamma)$ to such an operation O represent the values in the original stack that will end up in $\mathtt{s}(\gamma-1)$, ..., `s0` after the execution of this compound operation, at least if all i_ν, $1 \leq \nu \leq \gamma$, are distinct and at least γ. The mnemonic itself of the operation O is a sequence of γ two-letter strings `PU` and `XC`, with `PU` meaning that the corresponding argument is to be PUshed (i.e., a copy is to be created), and `XC` meaning that the value is to be eXChanged (i.e., no other copy of the original value is created). Sequences of several `PU` or `XC` strings may be abbreviated to one `PU` or `XC` followed by the number of copies. (For instance, we write `PUXC2PU` instead of `PUXCXCPU`.)

As an exception, if a mnemonic would consist of only `PU` or only `XC` strings, so that the compound operation is equivalent to a sequence of m PUSHes or eXCHanGes, the notation `PUSH`m or `XCHG`m is used instead of `PU`m or `XC`m.

2.2.5. Semantics of compound stack operations. Each compound γ-ary operation O $\mathtt{s}(i_1)$,...,$\mathtt{s}(i_\gamma)$ is translated into an equivalent sequence of basic stack operations by induction in γ as follows:

- As a base of induction, if $\gamma = 0$, the only nullary compound stack operation corresponds to an empty sequence of basic stack operations.
- Equivalently, we might begin the induction from $\gamma = 1$. Then `PU` $\mathtt{s}(i)$ corresponds to the sequence consisting of one basic operation `PUSH` $\mathtt{s}(i)$, and `XC` $\mathtt{s}(i)$ corresponds to the one-element sequence consisting of `XCHG` $\mathtt{s}(i)$.

- For $\gamma \geq 1$ (or for $\gamma \geq 2$, if we use $\gamma = 1$ as induction base), there are two subcases:

  1. $O\mathtt{s}(i_1), \ldots, \mathtt{s}(i_\gamma)$, with $O = \mathtt{XC}O'$, where O' is a compound operation of arity $\gamma - 1$ (i.e., the mnemonic of O' consists of $\gamma - 1$ strings `XC` and `PU`). Let α be the total quantity of `PU`shes in O, and β be that of e`XC`hanges, so that $\alpha + \beta = \gamma$. Then the original operation is translated into `XCHG` $\mathtt{s}(\beta - 1)$,$\mathtt{s}(i_1)$, followed by the translation of $O'\mathtt{s}(i_2), \ldots, \mathtt{s}(i_\gamma)$, defined by the induction hypothesis.
  2. $O\mathtt{s}(i_1), \ldots, \mathtt{s}(i_\gamma)$, with $O = \mathtt{PU}O'$, where O' is a compound operation of arity $\gamma - 1$. Then the original operation is translated into `PUSH` $\mathtt{s}(i_1)$; `XCHG` $\mathtt{s}(\beta)$, followed by the translation of $O'\mathtt{s}(i_2 + 1), \ldots, \mathtt{s}(i_\gamma + 1)$, defined by the induction hypothesis.[9]

2.2.6. Stack manipulation instructions are polymorphic. Notice that the stack manipulation instructions are almost the only "polymorphic" primitives in TVM—i.e., they work with values of arbitrary types (including the value types that will appear only in future revisions of TVM). For example, `SWAP` always interchanges the two top values of the stack, even if one of them is an integer and the other is a cell. Almost all other instructions, especially the data processing instructions (including arithmetic instructions), require each of their arguments to be of some fixed type (possibly different for different arguments).

2.3 Efficiency of stack manipulation primitives

Stack manipulation primitives employed by a stack machine, such as TVM, have to be implemented very efficiently, because they constitute more than half of all the instructions used in a typical program. In fact, TVM performs all these instructions in a (small) constant time, regardless of the values involved (even if they represent very large integers or very large trees of cells).

2.3.1. Implementation of stack manipulation primitives: using references for operations instead of objects. The efficiency of TVM's implementation of stack manipulation primitives results from the fact that a

[9] An alternative, arguably better, translation of $\mathtt{PU}O'\mathtt{s}(i_1), \ldots, \mathtt{s}(i_\gamma)$ consists of the translation of $O'\mathtt{s}(i_2), \ldots, \mathtt{s}(i_\gamma)$, followed by `PUSH` $\mathtt{s}(i_1 + \alpha - 1)$; `XCHG` $\mathtt{s}(\gamma - 1)$.

typical TVM implementation keeps in the stack not the value objects themselves, but only the references (pointers) to such objects. Therefore, a `SWAP` instruction only needs to interchange the references at `s0` and `s1`, not the actual objects they refer to.

2.3.2. Efficient implementation of `DUP` and `PUSH` instructions using copy-on-write. Furthermore, a `DUP` (or, more generally, `PUSH s`(i)) instruction, which appears to make a copy of a potentially large object, also works in small constant time, because it uses a copy-on-write technique of delayed copying: it copies only the reference instead of the object itself, but increases the "reference counter" inside the object, thus sharing the object between the two references. If an attempt to modify an object with a reference counter greater than one is detected, a separate copy of the object in question is made first (incurring a certain "non-uniqueness penalty" or "copying penalty" for the data manipulation instruction that triggered the creation of a new copy).

2.3.3. Garbage collecting and reference counting. When the reference counter of a TVM object becomes zero (for example, because the last reference to such an object has been consumed by a `DROP` operation or an arithmetic instruction), it is immediately freed. Because cyclic references are impossible in TVM data structures, this method of reference counting provides a fast and convenient way of freeing unused objects, replacing slow and unpredictable garbage collectors.

2.3.4. Transparency of the implementation: Stack values are "values", not "references". Regardless of the implementation details just discussed, all stack values are really "values", not "references", from the perspective of the TVM programmer, similarly to the values of all types in functional programming languages. Any attempt to modify an existing object referred to from any other objects or stack locations will result in a transparent replacement of this object by its perfect copy before the modification is actually performed.

In other words, the programmer should always act as if the objects themselves were directly manipulated by stack, arithmetic, and other data transformation primitives, and treat the previous discussion only as an explanation of the high efficiency of the stack manipulation primitives.

2.3.5. Absence of circular references. One might attempt to create a circular reference between two cells, A and B, as follows: first create A and

write some data into it; then create B and write some data into it, along with a reference to previously constructed cell A; finally, add a reference to B into A. While it may seem that after this sequence of operations we obtain a cell A, which refers to B, which in turn refers to A, this is not the case. In fact, we obtain a new cell A', which contains a copy of the data originally stored into cell A along with a reference to cell B, which contains a reference to (the original) cell A.

In this way the transparent copy-on-write mechanism and the "everything is a value" paradigm enable us to create new cells using only previously constructed cells, thus forbidding the appearance of circular references. This property also applies to all other data structures: for instance, the absence of circular references enables TVM to use reference counting to immediately free unused memory instead of relying on garbage collectors. Similarly, this property is crucial for storing data in the TON Blockchain.

3 Cells, memory, and persistent storage

This chapter briefly describes TVM cells, used to represent all data structures inside the TVM memory and its persistent storage, and the basic operations used to create cells, write (or serialize) data into them, and read (or deserialize) data from them.

3.1 Generalities on cells

This section presents a classification and general descriptions of cell types.

3.1.1. TVM memory and persistent storage consist of cells. Recall that the TVM memory and persistent storage consist of *(TVM) cells*. Each cell contains up to 1023 bits of data and up to four references to other cells.[10] Circular references are forbidden and cannot be created by means of TVM (cf. **2.3.5**). In this way, all cells kept in TVM memory and persistent storage constitute a directed acyclic graph (DAG).

3.1.2. Ordinary and exotic cells. Apart from the data and references, a cell has a *cell type*, encoded by an integer $-1 \ldots 255$. A cell of type -1 is called *ordinary*; such cells do not require any special processing. Cells of other types are called *exotic*, and may be *loaded*—automatically replaced by other cells when an attempt to deserialize them (i.e., to convert them into a *Slice* by a `CTOS` instruction) is made. They may also exhibit a non-trivial behavior when their hashes are computed.

The most common use for exotic cells is to represent some other cells—for instance, cells present in an external library, or pruned from the original tree of cells when a Merkle proof has been created.

The type of an exotic cell is stored as the first eight bits of its data. If an exotic cell has less than eight data bits, it is invalid.

3.1.3. The level of a cell. Every cell c has another attribute $\textsc{Lvl}(c)$ called its *(de Brujn) level*, which currently takes integer values in the range $0 \ldots 3$.

[10] From the perspective of low-level cell operations, these data bits and cell references are not intermixed. In other words, an (ordinary) cell essentially is a couple consisting of a list of up to 1023 bits and of a list of up to four cell references, without prescribing an order in which the references and the data bits should be deserialized, even though TL-B schemes appear to suggest such an order.

The level of an ordinary cell is always equal to the maximum of the levels of all its children c_i:

$$\textsc{Lvl}(c) = \max_{1 \le i \le r} \textsc{Lvl}(c_i) \quad , \tag{1}$$

for an ordinary cell c containing r references to cells $c_1, \ldots, c_r$. If $r = 0$, $\textsc{Lvl}(c) = 0$. Exotic cells may have different rules for setting their level.

A cell's level affects the number of *higher hashes* it has. More precisely, a level l cell has l higher hashes $\textsc{Hash}_1(c)$, ..., $\textsc{Hash}_l(c)$ in addition to its representation hash $\textsc{Hash}(c) = \textsc{Hash}_\infty(c)$. Cells of non-zero level appear inside *Merkle proofs* and *Merkle updates*, after some branches of the tree of cells representing a value of an abstract data type are pruned.

3.1.4. Standard cell representation. When a cell needs to be transferred by a network protocol or stored in a disk file, it must be *serialized*. The standard representation $\textsc{CellRepr}(c) = \textsc{CellRepr}_\infty(c)$ of a cell c as an octet (byte) sequence is constructed as follows:

1. Two descriptor bytes d_1 and d_2 are serialized first. Byte d_1 equals $r + 8s + 32l$, where $0 \le r \le 4$ is the quantity of cell references contained in the cell, $0 \le l \le 3$ is the level of the cell, and $0 \le s \le 1$ is 1 for exotic cells and 0 for ordinary cells. Byte d_2 equals $\lfloor b/8 \rfloor + \lceil b/8 \rceil$, where $0 \le b \le 1023$ is the quantity of data bits in c.

2. Then the data bits are serialized as $\lceil b/8 \rceil$ 8-bit octets (bytes). If b is not a multiple of eight, a binary `1` and up to six binary `0`s are appended to the data bits. After that, the data is split into $\lceil b/8 \rceil$ eight-bit groups, and each group is interpreted as an unsigned big-endian integer $0 \ldots 255$ and stored into an octet.

3. Finally, each of the r cell references is represented by 32 bytes containing the 256-bit *representation hash* $\textsc{Hash}(c_i)$, explained below in **3.1.5**, of the cell c_i referred to.

In this way, $2 + \lceil b/8 \rceil + 32r$ bytes of $\textsc{CellRepr}(c)$ are obtained.

3.1.5. The representation hash of a cell. The 256-bit *representation hash* or simply *hash* $\textsc{Hash}(c)$ of a cell c is recursively defined as the SHA256 of the standard representation of the cell c:

$$\textsc{Hash}(c) := \textsc{sha256}\big(\textsc{CellRepr}(c)\big) \tag{2}$$

Notice that cyclic cell references are not allowed and cannot be created by means of the TVM (cf. **2.3.5**), so this recursion always ends, and the representation hash of any cell is well-defined.

3.1.6. The higher hashes of a cell. Recall that a cell c of level l has l higher hashes $\text{Hash}_i(c)$, $1 \leq i \leq l$, as well. Exotic cells have their own rules for computing their higher hashes. Higher hashes $\text{Hash}_i(c)$ of an ordinary cell c are computed similarly to its representation hash, but using the higher hashes $\text{Hash}_i(c_j)$ of its children c_j instead of their representation hashes $\text{Hash}(c_j)$. By convention, we set $\text{Hash}_\infty(c) := \text{Hash}(c)$, and $\text{Hash}_i(c) := \text{Hash}_\infty(c) = \text{Hash}(c)$ for all $i > l$.[11]

3.1.7. Types of exotic cells. TVM currently supports the following cell types:

- Type -1: *Ordinary cell* — Contains up to 1023 bits of data and up to four cell references.
- Type 1: *Pruned branch cell* c — May have any level $1 \leq l \leq 3$. It contains exactly $8 + 256l$ data bits: first an 8-bit integer equal to 1 (representing the cell's type), then its l higher hashes $\text{Hash}_1(c)$, ..., $\text{Hash}_l(c)$. The level l of a pruned branch cell may be called its *de Brujn index*, because it determines the outer Merkle proof or Merkle update during the construction of which the branch has been pruned. An attempt to load a pruned branch cell usually leads to an exception.
- Type 2: *Library reference cell* — Always has level 0, and contains 8+256 data bits, including its 8-bit type integer 2 and the representation hash $\text{Hash}(c')$ of the library cell being referred to. When loaded, a library reference cell may be transparently replaced by the cell it refers to, if found in the current *library context*.
- Type 3: *Merkle proof cell* c — Has exactly one reference c_1 and level $0 \leq l \leq 3$, which must be one less than the level of its only child c_1:

$$\text{Lvl}(c) = \max(\text{Lvl}(c_1) - 1, 0) \tag{3}$$

[11] From a theoretical perspective, we might say that a cell c has an infinite sequence of hashes $\big(\text{Hash}_i(c)\big)_{i\geq 1}$, which eventually stabilizes: $\text{Hash}_i(c) \to \text{Hash}_\infty(c)$. Then the level l is simply the largest index i, such that $\text{Hash}_i(c) \neq \text{Hash}_\infty(c)$.

The $8 + 256$ data bits of a Merkle proof cell contain its 8-bit type integer 3, followed by $\text{Hash}_1(c_1)$ (assumed to be equal to $\text{Hash}(c_1)$ if $\text{Lvl}(c_1) = 0$). The higher hashes $\text{Hash}_i(c)$ of c are computed similarly to the higher hashes of an ordinary cell, but with $\text{Hash}_{i+1}(c_1)$ used instead of $\text{Hash}_i(c_1)$. When loaded, a Merkle proof cell is replaced by c_1.

- Type 4: *Merkle update cell* c — Has two children c_1 and c_2. Its level $0 \leq l \leq 3$ is given by

$$\text{Lvl}(c) = \max(\text{Lvl}(c_1) - 1, \text{Lvl}(c_2) - 1, 0) \tag{4}$$

A Merkle update behaves like a Merkle proof for both c_1 and c_2, and contains $8 + 256 + 256$ data bits with $\text{Hash}_1(c_1)$ and $\text{Hash}_1(c_2)$. However, an extra requirement is that *all pruned branch cells* c' *that are descendants of* c_2 *and are bound by* c *must also be descendants of* c_1.[12] When a Merkle update cell is loaded, it is replaced by c_2.

3.1.8. All values of algebraic data types are trees of cells. Arbitrary values of arbitrary algebraic data types (e.g., all types used in functional programming languages) can be serialized into trees of cells (of level 0), and such representations are used for representing such values within TVM. The copy-on-write mechanism (cf. **2.3.2**) allows TVM to identify cells containing the same data and references, and to keep only one copy of such cells. This actually transforms a tree of cells into a directed acyclic graph (with the additional property that all its vertices be accessible from a marked vertex called the "root"). However, this is a storage optimization rather than an essential property of TVM. From the perspective of a TVM code programmer, one should think of TVM data structures as trees of cells.

3.1.9. TVM code is a tree of cells. The TVM code itself is also represented by a tree of cells. Indeed, TVM code is simply a value of some complex algebraic data type, and as such, it can be serialized into a tree of cells.

The exact way in which the TVM code (e.g., TVM assembly code) is transformed into a tree of cells is explained later (cf. **4.1.4** and **5.2**), in sections discussing control flow instructions, continuations, and TVM instruction encoding.

[12] A pruned branch cell c' of level l is *bound* by a Merkle (proof or update) cell c if there are exactly l Merkle cells on the path from c to its descendant c', including c.

3.1.10. "Everything is a bag of cells" paradigm. As described in [1, 2.5.14], all the data used by the TON Blockchain, including the blocks themselves and the blockchain state, can be represented—and are represented—as collections, or "bags", of cells. We see that TVM's structure of data (cf. **3.1.8**) and code (cf. **3.1.9**) nicely fits into this "everything is a bag of cells" paradigm. In this way, TVM can naturally be used to execute smart contracts in the TON Blockchain, and the TON Blockchain can be used to store the code and persistent data of these smart contracts between invocations of TVM. (Of course, both TVM and the TON Blockchain have been designed so that this would become possible.)

3.2 Data manipulation instructions and cells

The next large group of TVM instructions consists of *data manipulation instructions*, also known as *cell manipulation instructions* or simply *cell instructions*. They correspond to memory access instructions of other architectures.

3.2.1. Classes of cell manipulation instructions. The TVM cell instructions are naturally subdivided into two principal classes:

- *Cell creation instructions* or *serialization instructions*, used to construct new cells from values previously kept in the stack and previously constructed cells.
- *Cell parsing instructions* or *deserialization instructions*, used to extract data previously stored into cells by cell creation instructions.

Additionally, there are *exotic cell instructions* used to create and inspect exotic cells (cf. **3.1.2**), which in particular are used to represent pruned branches of Merkle proofs and Merkle proofs themselves.

3.2.2. *Builder* and *Slice* values. Cell creation instructions usually work with *Builder* values, which can be kept only in the stack (cf. **1.1.3**). Such values represent partially constructed cells, for which fast operations for appending bitstrings, integers, other cells, and references to other cells can be defined. Similarly, cell parsing instructions make heavy use of *Slice* values, which represent either the remainder of a partially parsed cell, or a value (subcell) residing inside such a cell and extracted from it by a parsing instruction.

3.2.3. *Builder* and *Slice* values exist only as stack values. Notice that *Builder* and *Slice* objects appear only as values in a TVM stack. They cannot be stored in "memory" (i.e., trees of cells) or "persistent storage" (which is also a bag of cells). In this sense, there are far more *Cell* objects than *Builder* or *Slice* objects in a TVM environment, but, somewhat paradoxically, a TVM program sees *Builder* and *Slice* objects in its stack more often than *Cell*s. In fact, a TVM program does not have much use for *Cell* values, because they are immutable and opaque; all cell manipulation primitives require that a *Cell* value be transformed into either a *Builder* or a *Slice* first, before it can be modified or inspected.

3.2.4. TVM has no separate *Bitstring* value type. Notice that TVM offers no separate bitstring value type. Instead, bitstrings are represented by *Slice*s that happen to have no references at all, but can still contain up to 1023 data bits.

3.2.5. Cells and cell primitives are bit-oriented, not byte-oriented. An important point is that *TVM regards data kept in cells as sequences (strings, streams) of (up to 1023) bits, not of bytes.* In other words, TVM is a *bit-oriented machine*, not a byte-oriented machine. If necessary, an application is free to use, say, 21-bit integer fields inside records serialized into TVM cells, thus using fewer persistent storage bytes to represent the same data.

3.2.6. Taxonomy of cell creation (serialization) primitives. Cell creation primitives usually accept a *Builder* argument and an argument representing the value to be serialized. Additional arguments controlling some aspects of the serialization process (e.g., how many bits should be used for serialization) can be also provided, either in the stack or as an immediate value inside the instruction. The result of a cell creation primitive is usually another *Builder*, representing the concatenation of the original builder and the serialization of the value provided.

Therefore, one can suggest a classification of cell serialization primitives according to the answers to the following questions:

- Which is the type of values being serialized?
- How many bits are used for serialization? If this is a variable number, does it come from the stack, or from the instruction itself?

- What happens if the value does not fit into the prescribed number of bits? Is an exception generated, or is a success flag equal to zero silently returned in the top of stack?
- What happens if there is insufficient space left in the *Builder*? Is an exception generated, or is a zero success flag returned along with the unmodified original *Builder*?

The mnemonics of cell serialization primitives usually begin with `ST`. Subsequent letters describe the following attributes:

- The type of values being serialized and the serialization format (e.g., `I` for signed integers, `U` for unsigned integers).
- The source of the field width in bits to be used (e.g., `X` for integer serialization instructions means that the bit width n is supplied in the stack; otherwise it has to be embedded into the instruction as an immediate value).
- The action to be performed if the operation cannot be completed (by default, an exception is generated; "quiet" versions of serialization instructions are marked by a `Q` letter in their mnemonics).

This classification scheme is used to create a more complete taxonomy of cell serialization primitives, which can be found in **A.6.1**.

3.2.7. Integer serialization primitives. Integer serialization primitives can be classified according to the above taxonomy as well. For example:

- There are signed and unsigned (big-endian) integer serialization primitives.
- The size n of the bit field to be used ($1 \leq n \leq 257$ for signed integers, $0 \leq n \leq 256$ for unsigned integers) can either come from the top of stack or be embedded into the instruction itself.
- If the integer x to be serialized is not in the range $-2^{n-1} \leq x < 2^{n-1}$ (for signed integer serialization) or $0 \leq x < 2^n$ (for unsigned integer serialization), a range check exception is usually generated, and if n bits cannot be stored into the provided *Builder*, a cell overflow exception is generated.

- Quiet versions of serialization instructions do not throw exceptions; instead, they push `-1` on top of the resulting *Builder* upon success, or return the original *Builder* with `0` on top of it to indicate failure.

Integer serialization instructions have mnemonics like `STU 20` ("store an unsigned 20-bit integer value") or `STIXQ` ("quietly store an integer value of variable length provided in the stack"). The full list of these instructions—including their mnemonics, descriptions, and opcodes—is provided in **A.6.1**.

3.2.8. Integers in cells are big-endian by default. Notice that the default order of bits in *Integer*s serialized into *Cell*s is *big-endian*, not little-endian.[13] In this respect *TVM is a big-endian machine*. However, this affects only the serialization of integers inside cells. The internal representation of the *Integer* value type is implementation-dependent and irrelevant for the operation of TVM. Besides, there are some special primitives such as `STULE` for (de)serializing little-endian integers, which must be stored into an integral number of bytes (otherwise "little-endianness" does not make sense, unless one is also willing to revert the order of bits inside octets). Such primitives are useful for interfacing with the little-endian world—for instance, for parsing custom-format messages arriving to a TON Blockchain smart contract from the outside world.

3.2.9. Other serialization primitives. Other cell creation primitives serialize bitstrings (i.e., cell slices without references), either taken from the stack or supplied as literal arguments; cell slices (which are concatenated to the cell builder in an obvious way); other *Builder*s (which are also concatenated); and cell references (`STREF`).

3.2.10. Other cell creation primitives. In addition to the cell serialization primitives for certain built-in value types described above, there are simple primitives that create a new empty *Builder* and push it into the stack (`NEWC`), or transform a *Builder* into a *Cell* (`ENDC`), thus finishing the cell creation process. An `ENDC` can be combined with a `STREF` into a single instruction `ENDCST`, which finishes the creation of a cell and immediately stores a reference to it in an "outer" *Builder*. There are also primitives that obtain the quantity of data bits or references already stored in a *Builder*, and check how many data bits or references can be stored.

[13] Negative numbers are represented using two's complement. For instance, integer -17 is serialized by instruction `STI 8` into bitstring `xEF`.

3.2.11. Taxonomy of cell deserialisation primitives. Cell parsing, or deserialization, primitives can be classified as described in **3.2.6**, with the following modifications:

- They work with *Slice*s (representing the remainder of the cell being parsed) instead of *Builder*s.
- They return deserialized values instead of accepting them as arguments.
- They may come in two flavors, depending on whether they remove the deserialized portion from the *Slice* supplied ("fetch operations") or leave it unmodified ("prefetch operations").
- Their mnemonics usually begin with `LD` (or `PLD` for prefetch operations) instead of `ST`.

For example, an unsigned big-endian 20-bit integer previously serialized into a cell by a `STU 20` instruction is likely to be deserialized later by a matching `LDU 20` instruction.

Again, more detailed information about these instructions is provided in **A.6.2**.

3.2.12. Other cell slice primitives. In addition to the cell deserialisation primitives outlined above, TVM provides some obvious primitives for initializing and completing the cell deserialization process. For instance, one can convert a *Cell* into a *Slice* (`CTOS`), so that its deserialisation might begin; or check whether a *Slice* is empty, and generate an exception if it is not (`ENDS`); or deserialize a cell reference and immediately convert it into a *Slice* (`LDREFTOS`, equivalent to two instructions `LDREF` and `CTOS`).

3.2.13. Modifying a serialized value in a cell. The reader might wonder how the values serialized inside a cell may be modified. Suppose a cell contains three serialized 29-bit integers, (x, y, z), representing the coordinates of a point in space, and we want to replace y with $y' = y + 1$, leaving the other coordinates intact. How would we achieve this?

TVM does not offer any ways to modify existing values (cf. **2.3.4** and **2.3.5**), so our example can only be accomplished with a series of operations as follows:

1. Deserialize the original cell into three *Integer*s x, y, z in the stack (e.g., by `CTOS; LDI 29; LDI 29; LDI 29; ENDS`).

2. Increase y by one (e.g., by `SWAP; INC; SWAP`).

3. Finally, serialize the resulting *Integer*s into a new cell (e.g., by `XCHG s2; NEWC; STI 29; STI 29; STI 29; ENDC`).

3.2.14. Modifying the persistent storage of a smart contract. If the TVM code wants to modify its persistent storage, represented by the tree of cells rooted at `c4`, it simply needs to rewrite control register `c4` by the root of the tree of cells containing the new value of its persistent storage. (If only part of the persistent storage needs to be modified, cf. **3.2.13**.)

3.3 Hashmaps, or dictionaries

Hashmaps, or *dictionaries*, are a specific data structure represented by a tree of cells. Essentially, a hashmap represents a map from *keys*, which are bitstrings of either fixed or variable length, into *values* of an arbitrary type X, in such a way that fast lookups and modifications be possible. While any such structure might be inspected or modified with the aid of generic cell serialization and deserialization primitives, TVM introduces special primitives to facilitate working with these hashmaps.

3.3.1. Basic hashmap types. The two most basic hashmap types predefined in TVM are *HashmapE* n X or *HashmapE*(n, X), which represents a partially defined map from n-bit strings (called *keys*) for some fixed $0 \leq n \leq 1023$ into *values* of some type X, and *Hashmap*(n, X), which is similar to *HashmapE*(n, X) but is not allowed to be empty (i.e., it must contain at least one key-value pair).

Other hashmap types are also available—for example, one with keys of arbitrary length up to some predefined bound (up to 1023 bits).

3.3.2. Hashmaps as Patricia trees. The abstract representation of a hashmap in TVM is a *Patricia tree*, or a *compact binary trie*. It is a binary tree with edges labelled by bitstrings, such that the concatenation of all edge labels on a path from the root to a leaf equals a key of the hashmap. The corresponding value is kept in this leaf (for hashmaps with keys of fixed length), or optionally in the intermediate vertices as well (for hashmaps with keys of variable length). Furthermore, any intermediate vertex must have two children, and the label of the left child must begin with a binary zero, while the label of the right child must begin with a binary one. This enables us not to store the first bit of the edge labels explicitly.

It is easy to see that any collection of key-value pairs (with distinct keys) is represented by a unique Patricia tree.

3.3.3. Serialization of hashmaps. The serialization of a hashmap into a tree of cells (or, more generally, into a *Slice*) is defined by the following TL-B scheme:[14]

```
bit#_ _:(## 1) = Bit;

hm_edge#_ {n:#} {X:Type} {l:#} {m:#} label:(HmLabel ~l n)
          {n = (~m) + l} node:(HashmapNode m X) = Hashmap n X;

hmn_leaf#_ {X:Type} value:X = HashmapNode 0 X;
hmn_fork#_ {n:#} {X:Type} left:^(Hashmap n X)
           right:^(Hashmap n X) = HashmapNode (n+1) X;

hml_short$0 {n:#} len:(Unary ~n) s:n*bit = HmLabel ~n m;
hml_long$10 n:(#<= m) s:n*bit = HmLabel ~n m;
hml_same$11 v:bit n:(#<= m) = HmLabel ~n m;

unary_zero$0 = Unary ~0;
unary_succ$1 {n:#} x:(Unary ~n) = Unary ~(n+1);

hme_empty$0 {n:#} {X:Type} = HashmapE n X;
hme_root$1 {n:#} {X:Type} root:^(Hashmap n X) = HashmapE n X;

true#_ = True;
_ {n:#} _:(Hashmap n True) = BitstringSet n;
```

3.3.4. Brief explanation of TL-B schemes. A TL-B scheme, like the one above, includes the following components.

The right-hand side of each "equation" is a *type*, either simple (such as `Bit` or `True`) or parametrized (such as `Hashmap` n X). The parameters of a type must be either natural numbers (i.e., non-negative integers, which are required to fit into 32 bits in practice), such as n in `Hashmap` n X, or other types, such as X in `Hashmap` n X.

[14] A description of an older version of TL may be found at `https://core.telegram.org/mtproto/TL`.

The left-hand side of each equation describes a way to define, or even to serialize, a value of the type indicated in the right-hand side. Such a description begins with the name of a *constructor*, such as `hm_edge` or `hml_long`, immediately followed by an optional *constructor tag*, such as `#_` or `$10`, which describes the bitstring used to encode (serialize) the constructor in question. Such tags may be given in either binary (after a dollar sign) or hexadecimal notation (after a hash sign), using the conventions described in **1.0**. If a tag is not explicitly provided, TL-B computes a default 32-bit constructor tag by hashing the text of the "equation" defining this constructor in a certain fashion. Therefore, empty tags must be explicitly provided by `#_` or `$_`. All constructor names must be distinct, and constructor tags for the same type must constitute a prefix code (otherwise the deserialization would not be unique).

The constructor and its optional tag are followed by *field definitions*. Each field definition is of the form *ident* : *type-expr*, where *ident* is an identifier with the name of the field[15] (replaced by an underscore for anonymous fields), and *type-expr* is the field's type. The type provided here is a *type expression*, which may include simple types or parametrized types with suitable parameters. *Variables*—i.e., the (identifiers of the) previously defined fields of types # (natural numbers) or *Type* (type of types)—may be used as parameters for the parametrized types. The serialization process recursively serializes each field according to its type, and the serialization of a value ultimately consists of the concatenation of bitstrings representing the constructor (i.e., the constructor tag) and the field values.

Some fields may be *implicit*. Their definitions are surrounded by curly braces, which indicate that the field is not actually present in the serialization, but that its value must be deduced from other data (usually the parameters of the type being serialized).

Some occurrences of "variables" (i.e., already-defined fields) are prefixed by a tilde. This indicates that the variable's occurrence is used in the opposite way of the default behavior: in the left-hand side of the equation, it means that the variable will be deduced (computed) based on this occurrence, instead of substituting its previously computed value; in the right-hand side, conversely, it means that the variable will not be deduced from the type being serialized, but rather that it will be computed during the deserialization pro-

[15]The field's name is useful for representing values of the type being defined in human-readable form, but it does not affect the binary serialization.

cess. In other words, a tilde transforms an "input argument" into an "output argument", and vice versa.[16]

Finally, some equalities may be included in curly brackets as well. These are certain "equations", which must be satisfied by the "variables" included in them. If one of the variables is prefixed by a tilde, its value will be uniquely determined by the values of all other variables participating in the equation (which must be known at this point) when the definition is processed from the left to the right.

A caret (`^`) preceding a type X means that instead of serializing a value of type X as a bitstring inside the current cell, we place this value into a separate cell, and add a reference to it into the current cell. Therefore `^`X means "the type of references to cells containing values of type X".

Parametrized type `#<=` p with p : `#` (this notation means "p of type `#`", i.e., a natural number) denotes the subtype of the natural numbers type #, consisting of integers $0 \ldots p$; it is serialized into $\lceil \log_2(p+1) \rceil$ bits as an unsigned big-endian integer. Type `#` by itself is serialized as an unsigned 32-bit integer. Parametrized type `##` b with b : `#<=`31 is equivalent to `#<=` $2^b - 1$ (i.e., it is an unsigned b-bit integer).

3.3.5. Application to the serialization of hashmaps. Let us explain the net result of applying the general rules described in **3.3.4** to the TL-B scheme presented in **3.3.3**.

Suppose we wish to serialize a value of type *HashmapE* n X for some integer $0 \leq n \leq 1023$ and some type X (i.e., a dictionary with n-bit keys and values of type X, admitting an abstract representation as a Patricia tree (cf. **3.3.2**)).

First of all, if our dictionary is empty, it is serialized into a single binary `0`, which is the tag of nullary constructor `hme_empty`. Otherwise, its serialization consists of a binary `1` (the tag of `hme_root`), along with a reference to a cell containing the serialization of a value of type *Hashmap* n X (i.e., a necessarily non-empty dictionary).

The only way to serialize a value of type *Hashmap* n X is given by the `hm_edge` constructor, which instructs us to serialize first the label `label` of the edge leading to the root of the subtree under consideration (i.e., the common prefix of all keys in our (sub)dictionary). This label is of type `HmLabel` $l^{\perp}$ n, which means that it is a bitstring of length at most n, serialized in such a way that the true length l of the label, $0 \leq l \leq n$, becomes known from

[16] This is the "linear negation" operation $(-)^{\perp}$ of linear logic, hence our notation `~`.

the serialization of the label. (This special serialization method is described separately in **3.3.6**.)

The label must be followed by the serialization of a `node` of type *Hashmap-Node* m X, where $m = n - l$. It corresponds to a vertex of the Patricia tree, representing a non-empty subdictionary of the original dictionary with m-bit keys, obtained by removing from all the keys of the original subdictionary their common prefix of length l.

If $m = 0$, a value of type `HashmapNode` 0 X is given by the `hmn_leaf` constructor, which describes a leaf of the Patricia tree—or, equivalently, a subdictionary with 0-bit keys. A leaf simply consists of the corresponding `value` of type X and is serialized accordingly.

On the other hand, if $m > 0$, a value of type `HashmapNode` m X corresponds to a fork (i.e., an intermediate node) in the Patricia tree, and is given by the `hmn_fork` constructor. Its serialization consists of `left` and `right`, two references to cells containing values of type `Hashmap` $m - 1$ X, which correspond to the left and the right child of the intermediate node in question—or, equivalently, to the two subdictionaries of the original dictionary consisting of key-value pairs with keys beginning with a binary `0` or a binary `1`, respectively. Because the first bit of all keys in each of these subdictionaries is known and fixed, it is removed, and the resulting (necessarily non-empty) subdictionaries are recursively serialized as values of type `Hashmap` $m - 1$ X.

3.3.6. Serialization of labels. There are several ways to serialize a label of length at most n, if its exact length is $l \leq n$ (recall that the exact length must be deducible from the serialization of the label itself, while the upper bound n is known before the label is serialized or deserialized). These ways are described by the three constructors `hml_short`, `hml_long`, and `hml_same` of type `HmLabel` $l^{\perp}$ n:

- `hml_short` — Describes a way to serialize "short" labels, of small length $l \leq n$. Such a serialization consists of a binary `0` (the constructor tag of `hml_short`), followed by l binary `1`s and one binary `0` (the unary representation of the length l), followed by l bits comprising the label itself.

- `hml_long` — Describes a way to serialize "long" labels, of arbitrary length $l \leq n$. Such a serialization consists of a binary `10` (the constructor tag of `hml_long`), followed by the big-endian binary representation

of the length $0 \leq l \leq n$ in $\lceil \log_2(n+1) \rceil$ bits, followed by l bits comprising the label itself.

- `hml_same` — Describes a way to serialize "long" labels, consisting of l repetitions of the same bit v. Such a serialization consists of `11` (the constructor tag of `hml_same`), followed by the bit v, followed by the length l stored in $\lceil \log_2(n+1) \rceil$ bits as before.

Each label can always be serialized in at least two different fashions, using `hml_short` or `hml_long` constructors. Usually the shortest serialization (and in the case of a tie—the lexicographically smallest among the shortest) is preferred and is generated by TVM hashmap primitives, while the other variants are still considered valid.

This label encoding scheme has been designed to be efficient for dictionaries with "random" keys (e.g., hashes of some data), as well as for dictionaries with "regular" keys (e.g., big-endian representations of integers in some range).

3.3.7. An example of dictionary serialization. Consider a dictionary with three 16-bit keys 13, 17, and 239 (considered as big-endian integers) and corresponding 16-bit values 169, 289, and 57121.

In binary form:

```
0000000000001101 => 0000000010101001
0000000000010001 => 0000000100100001
0000000011101111 => 1101111100100001
```

The corresponding Patricia tree consists of a root A, two intermediate nodes B and C, and three leaf nodes D, E, and F, corresponding to 13, 17, and 239, respectively. The root A has only one child, B; the label on the edge AB is $00000000 = 0^8$. The node B has two children: its left child is an intermediate node C with the edge BC labelled by (0)00, while its right child is the leaf F with BF labelled by (1)1101111. Finally, C has two leaf children D and E, with CD labelled by (0)1101 and CE—by (1)0001.

The corresponding value of type `HashmapE 16 (## 16)` may be written in human-readable form as:

```
(hme_root$1
  root:^(hm_edge label:(hml_same$11 v:0 n:8) node:(hm_fork
    left:^(hm_edge label:(hml_short$0 len:$110 s:$00)
```

```
      node:(hm_fork
        left:^(hm_edge label:(hml_long$10 n:4 s:$1101)
          node:(hm_leaf value:169))
        right:^(hm_edge label:(hml_long$10 n:4 s:$0001)
          node:(hm_leaf value:289))))
    right:^(hm_edge label:(hml_long$10 n:7 s:$1101111)
      node:(hm_leaf value:57121)))))
```

The serialization of this data structure into a tree of cells consists of six cells with the following binary data contained in them:

```
A := 1
A.0 := 11 0 01000
A.0.0 := 0 110 00
A.0.0.0 := 10 100 1101 0000000010101001
A.0.0.1 := 10 100 0001 0000000100100001
A.0.1 := 10 111 1101111 1101111100100001
```

Here A is the root cell, $A.0$ is the cell at the first reference of A, $A.1$ is the cell at the second reference of A, and so on. This tree of cells can be represented more compactly using the hexadecimal notation described in **1.0**, using indentation to reflect the tree-of-cells structure:

```
C_
 C8
  62_
   A68054C_
   A08090C_
  BEFDF21
```

A total of 93 data bits and 5 references in 6 cells have been used to serialize this dictionary. Notice that a straightforward representation of three 16-bit keys and their corresponding 16-bit values would already require 96 bits (albeit without any references), so this particular serialization turns out to be quite efficient.

3.3.8. Ways to describe the serialization of type X. Notice that the built-in TVM primitives for dictionary manipulation need to know something about the serialization of type X; otherwise, they would not be able to work correctly with *Hashmap* n X, because values of type X are immediately

contained in the Patricia tree leaf cells. There are several options available to describe the serialization of type X:

- The simplest case is when $X = \hat{}Y$ for some other type Y. In this case the serialization of X itself always consists of one reference to a cell, which in fact must contain a value of type Y, something that is not relevant for dictionary manipulation primitives.

- Another simple case is when the serialization of any value of type X always consists of $0 \leq b \leq 1023$ data bits and $0 \leq r \leq 4$ references. Integers b and r can then be passed to a dictionary manipulation primitive as a simple description of X. (Notice that the previous case corresponds to $b = 0$, $r = 1$.)

- A more sophisticated case can be described by four integers $1 \leq b_0, b_1 \leq 1023$, $0 \leq r_0, r_1 \leq 4$, with b_i and r_i used when the first bit of the serialization equals i. When $b_0 = b_1$ and $r_0 = r_1$, this case reduces to the previous one.

- Finally, the most general description of the serialization of a type X is given by a *splitting function* $split_X$ for X, which accepts one *Slice* parameter s, and returns two *Slice*s, s' and s'', where s' is the only prefix of s that is the serialization of a value of type X, and s'' is the remainder of s. If no such prefix exists, the splitting function is expected to throw an exception. Notice that a compiler for a high-level language, which supports some or all algebraic TL-B types, is likely to automatically generate splitting functions for all types defined in the program.

3.3.9. A simplifying assumption on the serialization of X. One may notice that values of type X always occupy the remaining part of an `hm_edge`/`hme_leaf` cell inside the serialization of a *HashmapE n X*. Therefore, if we do not insist on strict validation of all dictionaries accessed, we may assume that everything left unparsed in an `hm_edge`/`hme_leaf` cell after deserializing its `label` is a value of type X. This greatly simplifies the creation of dictionary manipulation primitives, because in most cases they turn out not to need any information about X at all.

3.3.10. Basic dictionary operations. Let us present a classification of basic operations with dictionaries (i.e., values D of type *HashmapE n X*):

- Get(D, k) — Given $D : \mathit{HashmapE}(n, X)$ and a key $k : n \cdot \texttt{bit}$, returns the corresponding value $D[k] : X^?$ kept in D.
- Set(D, k, x) — Given $D : \mathit{HashmapE}(n, X)$, a key $k : n \cdot \texttt{bit}$, and a value $x : X$, sets $D'[k]$ to x in a copy D' of D, and returns the resulting dictionary D' (cf. **2.3.4**).
- Add(D, k, x) — Similar to Set, but adds the key-value pair (k, x) to D only if key k is absent in D.
- Replace(D, k, x) — Similar to Set, but changes $D'[k]$ to x only if key k is already present in D.
- GetSet, GetAdd, GetReplace — Similar to Set, Add, and Replace, respectively, but returns the old value of $D[k]$ as well.
- Delete(D, k) — Deletes key k from dictionary D, and returns the resulting dictionary D'.
- GetMin(D), GetMax(D) — Gets the minimal or maximal key k from dictionary D, along with the associated value $x : X$.
- RemoveMin(D), RemoveMax(D) — Similar to GetMin and GetMax, but also removes the key in question from dictionary D, and returns the modified dictionary D'. May be used to iterate over all elements of D, effectively using (a copy of) D itself as an iterator.
- GetNext(D, k) — Computes the minimal key $k' > k$ (or $k' \geq k$ in a variant) and returns it along with the corresponding value $x' : X$. May be used to iterate over all elements of D.
- GetPrev(D, k) — Computes the maximal key $k' < k$ (or $k' \leq k$ in a variant) and returns it along with the corresponding value $x' : X$.
- Empty(n) — Creates an empty dictionary $D : \mathit{HashmapE}(n, X)$.
- IsEmpty(D) — Checks whether a dictionary is empty.
- Create$(n, \{(k_i, x_i)\})$ — Given n, creates a dictionary from a list (k_i, x_i) of key-value pairs passed in stack.

- GetSubdict(D, l, k_0) — Given D : *HashmapE*(n, X) and some l-bit string $k_0 : l \cdot \texttt{bit}$ for $0 \leq l \leq n$, returns subdictionary $D' = D/k_0$ of D, consisting of keys beginning with k_0. The result D' may be of either type *HashmapE*(n, X) or type *HashmapE*$(n - l, X)$.

- ReplaceSubdict(D, l, k_0, D') — Given D : *HashmapE*(n, X), $0 \leq l \leq n$, $k_0 : l \cdot \texttt{bit}$, and D' : *HashmapE*$(n - l, X)$, replaces with D' the subdictionary D/k_0 of D consisting of keys beginning with k_0, and returns the resulting dictionary D'' : *HashmapE*(n, X). Some variants of ReplaceSubdict may also return the old value of the subdictionary D/k_0 in question.

- DeleteSubdict(D, l, k_0) — Equivalent to ReplaceSubdict with D' being an empty dictionary.

- Split(D) — Given D : *HashmapE*(n, X), returns $D_0 := D/0$ and $D_1 := D/1$: *HashmapE*$(n - 1, X)$, the two subdictionaries of D consisting of all keys beginning with 0 and 1, respectively.

- Merge(D_0, D_1) — Given D_0 and D_1 : *HashmapE*$(n-1, X)$, computes D : *HashmapE*(n, X), such that $D/0 = D_0$ and $D/1 = D_1$.

- Foreach(D, f) — Executes a function f with two arguments k and x, with (k, x) running over all key-value pairs of a dictionary D in lexicographical order.[17]

- ForeachRev(D, f) — Similar to Foreach, but processes all key-value pairs in reverse order.

- TreeReduce(D, o, f, g) — Given D : *HashmapE*(n, X), a value $o : X$, and two functions $f : X \to Y$ and $g : Y \times Y \to Y$, performs a "tree reduction" of D by first applying f to all the leaves, and then using g to compute the value corresponding to a fork starting from the values assigned to its children.[18]

[17] In fact, f may receive m extra arguments and return m modified values, which are passed to the next invocation of f. This may be used to implement "map" and "reduce" operations with dictionaries.

[18] Versions of this operation may be introduced where f and g receive an additional bitstring argument, equal to the key (for leaves) or to the common prefix of all keys (for forks) in the corresponding subtree.

3.3.11. Taxonomy of dictionary primitives. The dictionary primitives, described in detail in **A.9**, can be classified according to the following categories:

- Which dictionary operation (cf. **3.3.10**) do they perform?
- Are they specialized for the case $X = \hat{}Y$? If so, do they represent values of type Y by *Cell*s or by *Slice*s? (Generic versions always represent values of type X as *Slice*s.)
- Are the dictionaries themselves passed and returned as *Cell*s or as *Slice*s? (Most primitives represent dictionaries as *Slice*s.)
- Is the key length n fixed inside the primitive, or is it passed in the stack?
- Are the keys represented by *Slice*s, or by signed or unsigned *Integer*s?

In addition, TVM includes special serialization/deserialization primitives, such as `STDICT`, `LDDICT`, and `PLDDICT`. They can be used to extract a dictionary from a serialization of an encompassing object, or to insert a dictionary into such a serialization.

3.4 Hashmaps with variable-length keys

TVM provides some support for dictionaries, or hashmaps, with variable-length keys, in addition to its support for dictionaries with fixed-length keys (as described in **3.3** above).

3.4.1. Serialization of dictionaries with variable-length keys. The serialization of a *VarHashmap* into a tree of cells (or, more generally, into a *Slice*) is defined by a TL-B scheme, similar to that described in **3.3.3**:

```
vhm_edge#_ {n:#} {X:Type} {l:#} {m:#} label:(HmLabel ~l n)
           {n = (~m) + l} node:(VarHashmapNode m X)
           = VarHashmap n X;
vhmn_leaf$00 {n:#} {X:Type} value:X = VarHashmapNode n X;
vhmn_fork$01 {n:#} {X:Type} left:^(VarHashmap n X)
             right:^(VarHashmap n X) value:(Maybe X)
             = VarHashmapNode (n+1) X;
vhmn_cont$1 {n:#} {X:Type} branch:bit child:^(VarHashmap n X)
```

```
            value:X = VarHashmapNode (n+1) X;

nothing$0 {X:Type} = Maybe X;
just$1 {X:Type} value:X = Maybe X;

vhme_empty$0 {n:#} {X:Type} = VarHashmapE n X;
vhme_root$1 {n:#} {X:Type} root:^(VarHashmap n X)
            = VarHashmapE n X;
```

3.4.2. Serialization of prefix codes. One special case of a dictionary with variable-length keys is that of a *prefix code*, where the keys cannot be prefixes of each other. Values in such dictionaries may occur only in the leaves of a Patricia tree.

The serialization of a prefix code is defined by the following TL-B scheme:

```
phm_edge#_ {n:#} {X:Type} {l:#} {m:#} label:(HmLabel ~l n)
           {n = (~m) + l} node:(PfxHashmapNode m X)
           = PfxHashmap n X;

phmn_leaf$0 {n:#} {X:Type} value:X = PfxHashmapNode n X;
phmn_fork$1 {n:#} {X:Type} left:^(PfxHashmap n X)
            right:^(PfxHashmap n X) = PfxHashmapNode (n+1) X;

phme_empty$0 {n:#} {X:Type} = PfxHashmapE n X;
phme_root$1 {n:#} {X:Type} root:^(PfxHashmap n X)
            = PfxHashmapE n X;
```

4 Control flow, continuations, and exceptions

This chapter describes *continuations*, which may represent execution tokens and exception handlers in TVM. Continuations are deeply involved with the control flow of a TVM program; in particular, subroutine calls and conditional and iterated execution are implemented in TVM using special primitives that accept one or more continuations as their arguments.

We conclude this chapter with a discussion of the problem of recursion and of families of mutually recursive functions, exacerbated by the fact that cyclic references are not allowed in TVM data structures (including TVM code).

4.1 Continuations and subroutines

Recall (cf.**1.1.3**) that *Continuation* values represent "execution tokens" that can be executed later—for example, by `EXECUTE`=`CALLX` ("execute" or "call indirect") or `JMPX` ("jump indirect") primitives. As such, the continuations are responsible for the execution of the program, and are heavily used by control flow primitives, enabling subroutine calls, conditional expressions, loops, and so on.

4.1.1. Ordinary continuations. The most common kind of continuations are the *ordinary continuations*, containing the following data:

- A *Slice* `code` (cf. **1.1.3** and **3.2.2**), containing (the remainder of) the TVM code to be executed.
- A (possibly empty) *Stack* `stack`, containing the original contents of the stack for the code to be executed.
- A (possibly empty) list `save` of pairs $(\mathtt{c}(i), v_i)$ (also called "savelist"), containing the values of control registers to be restored before the execution of the code.
- A 16-bit integer value `cp`, selecting the TVM codepage used to interpret the TVM code from `code`.
- An optional non-negative integer `nargs`, indicating the number of arguments expected by the continuation.

4.1.2. Simple ordinary continuations. In most cases, the ordinary continuations are the simplest ones, having empty `stack` and `save`. They consist essentially of a reference `code` to (the remainder of) the code to be executed, and of the codepage `cp` to be used while decoding the instructions from this code.

4.1.3. Current continuation `cc`. The "current continuation" `cc` is an important part of the total state of TVM, representing the code being executed right now (cf. **1.1**). In particular, what we call "the current stack" (or simply "the stack") when discussing all other primitives is in fact the stack of the current continuation. All other components of the total state of TVM may be also thought of as parts of the current continuation `cc`; however, they may be extracted from the current continuation and kept separately as part of the total state for performance reasons. This is why we describe the stack, the control registers, and the codepage as separate parts of the TVM state in **1.4**.

4.1.4. Normal work of TVM, or the main loop. TVM usually performs the following operations:

If the current continuation `cc` is an ordinary one, it decodes the first instruction from the *Slice* `code`, similarly to the way other cells are deserialized by TVM `LD*` primitives (cf. **3.2** and **3.2.11**): it decodes the opcode first, and then the parameters of the instruction (e.g., 4-bit fields indicating "stack registers" involved for stack manipulation primitives, or constant values for "push constant" or "literal" primitives). The remainder of the *Slice* is then put into the `code` of the new `cc`, and the decoded operation is executed on the current stack. This entire process is repeated until there are no operations left in `cc.code`.

If the `code` is empty (i.e., contains no bits of data and no references), or if a (rarely needed) explicit subroutine return (`RET`) instruction is encountered, the current continuation is discarded, and the "return continuation" from control register `c0` is loaded into `cc` instead (this process is discussed in more detail starting in **4.1.6**).[19] Then the execution continues by parsing operations from the new current continuation.

4.1.5. Extraordinary continuations. In addition to the ordinary continuations considered so far (cf. **4.1.1**), TVM includes some *extraordinary contin-*

[19]If there are no bits of data left in `code`, but there is still exactly one reference, an implicit `JMP` to the cell at that reference is performed instead of an implicit `RET`.

uations, representing certain less common states. Examples of extraordinary continuations include:

- The continuation `ec_quit` with its parameter set to zero, which represents the end of the work of TVM. This continuation is the original value of `c0` when TVM begins executing the code of a smart contract.
- The continuation `ec_until`, which contains references to two other continuations (ordinary or not) representing the body of the loop being executed and the code to be executed after the loop.

Execution of an extraordinary continuation by TVM depends on its specific class, and differs from the operations for ordinary continuations described in **4.1.4**.[20]

4.1.6. Switching to another continuation: `JMP` and `RET`. The process of switching to another continuation c may be performed by such instructions as `JMPX` (which takes c from the stack) or `RET` (which uses `c0` as c). This process is slightly more complex than simply setting the value of `cc` to c: before doing this, either all values or the top n values in the current stack are moved to the stack of the continuation c, and only then is the remainder of the current stack discarded.

If all values need to be moved (the most common case), and if the continuation c has an empty stack (also the most common case; notice that extraordinary continuations are assumed to have an empty stack), then the new stack of c equals the stack of the current continuation, so we can simply transfer the current stack in its entirety to c. (If we keep the current stack as a separate part of the total state of TVM, we have to do nothing at all.)

4.1.7. Determining the number n of arguments passed to the next continuation c. By default, n equals the depth of the current stack. However, if c has an explicit value of `nargs` (number of arguments to be provided), then n is computed as n', equal to c.`nargs` minus the current depth of c's stack.

Furthermore, there are special forms of `JMPX` and `RET` that provide an explicit value n'', the number of parameters from the current stack to be passed to continuation c. If n'' is provided, it must be less than or equal to

[20]Technically, TVM may simply invoke a virtual method `run()` of the continuation currently in `cc`.

the depth of the current stack, or else a stack underflow exception occurs. If both n' and n'' are provided, we must have $n' \leq n''$, in which case $n = n'$ is used. If n'' is provided and n' is not, then $n = n''$ is used.

One could also imagine that the default value of n'' equals the depth of the original stack, and that n'' values are always removed from the top of the original stack even if only n' of them are actually moved to the stack of the next continuation c. Even though the remainder of the current stack is discarded afterwards, this description will become useful later.

4.1.8. Restoring control registers from the new continuation c. After the new stack is computed, the values of control registers present in c.`save` are restored accordingly, and the current codepage `cp` is also set to c.`cp`. Only then does TVM set `cc` equal to the new c and begin its execution.[21]

4.1.9. Subroutine calls: `CALLX` or `EXECUTE` primitives. The execution of continuations as subroutines is slightly more complicated than switching to continuations.

Consider the `CALLX` or `EXECUTE` primitive, which takes a continuation c from the (current) stack and executes it as a subroutine.

Apart from doing the stack manipulations described in **4.1.6** and **4.1.7** and setting the new control registers and codepage as described in **4.1.8**, these primitives perform several additional steps:

1. After the top n'' values are removed from the current stack (cf. **4.1.7**), the (usually empty) remainder is not discarded, but instead is stored in the (old) current continuation `cc`.

2. The old value of the special register `c0` is saved into the (previously empty) savelist `cc.save`.

3. The continuation `cc` thus modified is not discarded, but instead is set as the new `c0`, which performs the role of "next continuation" or "return continuation" for the subroutine being called.

4. After that, the switching to c continues as before. In particular, some control registers are restored from c.`save`, potentially overwriting the value of `c0` set in the previous step. (Therefore, a good optimization would be to check that $c0$ is present in c.`save` from the very beginning, and skip the three previous steps as useless in this case.)

[21] The already used savelist `cc.save` of the new `cc` is emptied before the execution starts.

In this way, the called subroutine can return control to the caller by switching the current continuation to the return continuation saved in `c0`. Nested subroutine calls work correctly because the previous value of `c0` ends up saved into the new `c0`'s control register savelist `c0.save`, from which it is restored later.

4.1.10. Determining the number of arguments passed to and/or return values accepted from a subroutine. Similarly to `JMPX` and `RET`, `CALLX` also has special (rarely used) forms, which allow us to explicitly specify the number n'' of arguments passed from the current stack to the called subroutine (by default, n'' equals the depth of the current stack, i.e., it is passed in its entirety). Furthermore, a second number n''' can be specified, used to set `nargs` of the modified `cc` continuation before storing it into the new `c0`; the new `nargs` equals the depth of the old stack minus n'' plus n'''. This means that the caller is willing to pass exactly n'' arguments to the called subroutine, and is willing to accept exactly n''' results in their stead.

Such forms of `CALLX` and `RET` are mostly intended for library functions that accept functional arguments and want to invoke them safely. Another application is related to the "virtualization support" of TVM, which enables TVM code to run other TVM code inside a "virtual TVM machine". Such virtualization techniques might be useful for implementing sophisticated payment channels in the TON Blockchain (cf. [1, 5]).

4.1.11. `CALLCC`: call with current continuation. Notice that TVM supports a form of the "call with current continuation" primitive. Namely, primitive `CALLCC` is similar to `CALLX` or `JMPX` in that it takes a continuation c from the stack and switches to it; however, `CALLCC` does not discard the previous current continuation c' (as `JMPX` does) and does not write c' to `c0` (as `CALLX` does), but rather pushes c' into the (new) stack as an extra argument to c. The primitive `JMPXDATA` does a similar thing, but pushes only the (remainder of the) code of the previous current continuation as a *Slice*.

4.2 Control flow primitives: conditional and iterated execution

4.2.1. Conditional execution: `IF`, `IFNOT`, `IFELSE`. An important modification of `EXECUTE` (or `CALLX`) consists in its conditional forms. For example, `IF` accepts an integer x and a continuation c, and executes c (in the same

way as EXECUTE would do it) only if x is non-zero; otherwise both values are simply discarded from the stack. Similarly, IFNOT accepts x and c, but executes c only if $x = 0$. Finally, IFELSE accepts x, c, and c', removes these values from the stack, and executes c if $x \neq 0$ or c' if $x = 0$.

4.2.2. Iterated execution and loops. More sophisticated modifications of EXECUTE include:

- REPEAT — Takes an integer n and a continuation c, and executes c n times.[22]
- WHILE — Takes c' and c'', executes c', and then takes the top value x from the stack. If x is non-zero, it executes c'' and then begins a new loop by executing c' again; if x is zero, it stops.
- UNTIL — Takes c, executes it, and then takes the top integer x from the stack. If x is zero, a new iteration begins; if x is non-zero, the previously executed code is resumed.

4.2.3. Constant, or literal, continuations. We see that we can create arbitrarily complex conditional expressions and loops in the TVM code, provided we have a means to push constant continuations into the stack. In fact, TVM includes special versions of "literal" or "constant" primitives that cut the next n bytes or bits from the remainder of the current code cc.code into a cell slice, and then push it into the stack not as a *Slice* (as a PUSHSLICE does) but as a simple ordinary *Continuation* (which has only code and cp).

The simplest of these primitives is PUSHCONT, which has an immediate argument n describing the number of subsequent bytes (in a byte-oriented version of TVM) or bits to be converted into a simple continuation. Another primitive is PUSHREFCONT, which removes the first cell reference from the current continuation cc.code, converts the cell referred to into a cell slice, and finally converts the cell slice into a simple continuation.

4.2.4. Constant continuations combined with conditional or iterated execution primitives. Because constant continuations are very often used as arguments to conditional or iterated execution primitives, combined

[22]The implementation of REPEAT involves an extraordinary continuation that remembers the remaining number of iterations, the body of the loop c, and the return continuation c'. (The latter term represents the remainder of the body of the function that invoked REPEAT, which would be normally stored in c0 of the new cc.)

versions of these primitives (e.g., `IFCONT` or `UNTILREFCONT`) may be defined in a future revision of TVM, which combine a `PUSHCONT` or `PUSHREFCONT` with another primitive. If one inspects the resulting code, `IFCONT` looks very much like the more customary "conditional-branch-forward" instruction.

4.3 Operations with continuations

4.3.1. Continuations are opaque. Notice that all continuations are *opaque*, at least in the current version of TVM, meaning that there is no way to modify a continuation or inspect its internal data. Almost the only use of a continuation is to supply it to a control flow primitive.

While there are some arguments in favor of including support for non-opaque continuations in TVM (along with opaque continuations, which are required for virtualization), the current revision offers no such support.

4.3.2. Allowed operations with continuations. However, some operations with opaque continuations are still possible, mostly because they are equivalent to operations of the kind "create a new continuation, which will do something special, and then invoke the original continuation". Allowed operations with continuations include:

- Push one or several values into the stack of a continuation c (thus creating a partial application of a function, or a closure).
- Set the saved value of a control register $\texttt{c}(i)$ inside the savelist $c.\texttt{save}$ of a continuation c. If there is already a value for the control register in question, this operation silently does nothing.

4.3.3. Example: operations with control registers. TVM has some primitives to set and inspect the values of control registers. The most important of them are `PUSH` $\texttt{c}(i)$ (pushes the current value of $\texttt{c}(i)$ into the stack) and `POP` $\texttt{c}(i)$ (sets the value of $\texttt{c}(i)$ from the stack, if the supplied value is of the correct type). However, there is also a modified version of the latter instruction, called `POPSAVE` $\texttt{c}(i)$, which saves the old value of $\texttt{c}(i)$ (for $i > 0$) into the continuation at `c0` as described in **4.3.2** before setting the new value.

4.3.4. Example: setting the number of arguments to a function in its code. The primitive `LEAVEARGS` n demonstrates another application of continuations in an operation: it leaves only the top n values of the current stack, and moves the remainder to the stack of the continuation in `c0`.

This primitive enables a called function to "return" unneeded arguments to its caller's stack, which is useful in some situations (e.g., those related to exception handling).

4.3.5. Boolean circuits. A continuation c may be thought of as a piece of code with two optional exit points kept in the savelist of c: the principal exit point given by $c.\texttt{c0} := c.\texttt{save}(\texttt{c0})$, and the auxiliary exit point given by $c.\texttt{c1} := c.\texttt{save}(\texttt{c1})$. If executed, a continuation performs whatever action it was created for, and then (usually) transfers control to the principal exit point, or, on some occasions, to the auxiliary exit point. We sometimes say that a continuation c with both exit points $c.\texttt{c0}$ and $c.\texttt{c1}$ defined is a *two-exit continuation*, or a *boolean circuit*, especially if the choice of the exit point depends on some internally-checked condition.

4.3.6. Composition of continuations. One can *compose* two continuations c and c' simply by setting $c.\texttt{c0}$ or $c.\texttt{c1}$ to c'. This creates a new continuation denoted by $c \circ_0 c'$ or $c \circ_1 c'$, which differs from c in its savelist. (Recall that if the savelist of c already has an entry corresponding to the control register in question, such an operation silently does nothing as explained in **4.3.2**).

By composing continuations, one can build chains or other graphs, possibly with loops, representing the control flow. In fact, the resulting graph resembles a flow chart, with the boolean circuits corresponding to the "condition nodes" (containing code that will transfer control either to `c0` or to `c1` depending on some condition), and the one-exit continuations corresponding to the "action nodes".

4.3.7. Basic continuation composition primitives. Two basic primitives for composing continuations are `COMPOS` (also known as `SETCONT c0` and `BOOLAND`) and `COMPOSALT` (also known as `SETCONT c1` and `BOOLOR`), which take c and c' from the stack, set $c.\texttt{c0}$ or $c.\texttt{c1}$ to c', and return the resulting continuation $c'' = c \circ_0 c'$ or $c \circ_1 c'$. All other continuation composition operations can be expressed in terms of these two primitives.

4.3.8. Advanced continuation composition primitives. However, TVM can compose continuations not only taken from stack, but also taken from `c0` or `c1`, or from the current continuation `cc`; likewise, the result may be pushed into the stack, stored into either `c0` or `c1`, or used as the new current continuation (i.e., the control may be transferred to it). Furthermore, TVM

can define conditional composition primitives, performing some of the above actions only if an integer value taken from the stack is non-zero.

For instance, `EXECUTE` can be described as $\texttt{cc} \leftarrow c \circ_0 \texttt{cc}$, with continuation c taken from the original stack. Similarly, `JMPX` is $\texttt{cc} \leftarrow c$, and `RET` (also known as `RETTRUE` in a boolean circuit context) is $\texttt{cc} \leftarrow \texttt{c0}$. Other interesting primitives include `THENRET` ($c' \leftarrow c \circ_0 \texttt{c0}$) and `ATEXIT` ($\texttt{c0} \leftarrow c \circ_0 \texttt{c0}$).

Finally, some "experimental" primitives also involve `c1` and $\circ_1$. For example:

- `RETALT` or `RETFALSE` does $\texttt{cc} \leftarrow \texttt{c1}$.
- Conditional versions of `RET` and `RETALT` may also be useful: `RETBOOL` takes an integer x from the stack, and performs `RETTRUE` if $x \neq 0$, `RETFALSE` otherwise.
- `INVERT` does $\texttt{c0} \leftrightarrow \texttt{c1}$; if the two continuations in `c0` and `c1` represent the two branches we should select depending on some boolean expression, `INVERT` negates this expression on the outer level.
- `INVERTCONT` does $c.\texttt{c0} \leftrightarrow c.\texttt{c1}$ to a continuation c taken from the stack.
- Variants of `ATEXIT` include `ATEXITALT` ($\texttt{c1} \leftarrow c \circ_1 \texttt{c1}$) and `SETEXITALT` ($\texttt{c1} \leftarrow (c \circ_0 \texttt{c0}) \circ_1 \texttt{c1}$).
- `BOOLEVAL` takes a continuation c from the stack and does $\texttt{cc} \leftarrow \big((c \circ_0 (\texttt{PUSH} -1)) \circ_1 (\texttt{PUSH0})\big) \circ_0 \texttt{cc}$. If c represents a boolean circuit, the net effect is to evaluate it and push either -1 or 0 into the stack before continuing.

4.4 Continuations as objects

4.4.1. Representing objects using continuations. Object-oriented programming in Smalltalk (or Objective C) style may be implemented with the aid of continuations. For this, an object is represented by a special continuation o. If it has any data fields, they can be kept in the stack of o, making o a partial application (i.e., a continuation with a non-empty stack).

When somebody wants to invoke a method m of o with arguments x_1, x_2, ..., x_n, she pushes the arguments into the stack, then pushes a magic number corresponding to the method m, and then executes o passing $n+1$ arguments (cf. **4.1.10**). Then o uses the top-of-stack integer m to select the branch with

the required method, and executes it. If o needs to modify its state, it simply computes a new continuation o' of the same sort (perhaps with the same code as o, but with a different initial stack). The new continuation o' is returned to the caller along with whatever other return values need to be returned.

4.4.2. Serializable objects. Another way of representing Smalltalk-style objects as continuations, or even as trees of cells, consists in using the `JMPREFDATA` primitive (a variant of `JMPXDATA`, cf. **4.1.11**), which takes the first cell reference from the code of the current continuation, transforms the cell referred to into a simple ordinary continuation, and transfers control to it, first pushing the remainder of the current continuation as a *Slice* into the stack. In this way, an object might be represented by a cell $\tilde{o}$ that contains `JMPREFDATA` at the beginning of its data, and the actual code of the object in the first reference (one might say that the first reference of cell $\tilde{o}$ is the *class* of object $\tilde{o}$). Remaining data and references of this cell will be used for storing the fields of the object.

Such objects have the advantage of being trees of cells, and not just continuations, meaning that they can be stored into the persistent storage of a TON smart contract.

4.4.3. Unique continuations and capabilities. It might make sense (in a future revision of TVM) to mark some continuations as *unique*, meaning that they cannot be copied, even in a delayed manner, by increasing their reference counter to a value greater than one. If an opaque continuation is unique, it essentially becomes a *capability*, which can either be used by its owner exactly once or be transferred to somebody else.

For example, imagine a continuation that represents the output stream to a printer (this is an example of a continuation used as an object, cf. **4.4.1**). When invoked with one integer argument n, this continuation outputs the character with code n to the printer, and returns a new continuation of the same kind reflecting the new state of the stream. Obviously, copying such a continuation and using the two copies in parallel would lead to some unintended side effects; marking it as unique would prohibit such adverse usage.

4.5 Exception handling

TVM's exception handling is quite simple and consists in a transfer of control to the continuation kept in control register `c2`.

4.5.1. Two arguments of the exception handler: exception parameter and exception number. Every exception is characterized by two arguments: the *exception number* (an *Integer*) and the *exception parameter* (any value, most often a zero *Integer*). Exception numbers 0–31 are reserved for TVM, while all other exception numbers are available for user-defined exceptions.

4.5.2. Primitives for throwing an exception. There are several special primitives used for throwing an exception. The most general of them, `THROWANY`, takes two arguments, v and $0 \leq n < 2^{16}$, from the stack, and throws the exception with number n and value v. There are variants of this primitive that assume v to be a zero integer, store n as a literal value, and/or are conditional on an integer value taken from the stack. User-defined exceptions may use arbitrary values as v (e.g., trees of cells) if needed.

4.5.3. Exceptions generated by TVM. Of course, some exceptions are generated by normal primitives. For example, an arithmetic overflow exception is generated whenever the result of an arithmetic operation does not fit into a signed 257-bit integer. In such cases, the arguments of the exception, v and n, are determined by TVM itself.

4.5.4. Exception handling. The exception handling itself consists in a control transfer to the exception handler—i.e., the continuation specified in control register `c2`, with v and n supplied as the two arguments to this continuation, as if a `JMP` to `c2` had been requested with $n'' = 2$ arguments (cf. **4.1.7** and **4.1.6**). As a consequence, v and n end up in the top of the stack of the exception handler. The remainder of the old stack is discarded.

Notice that if the continuation in `c2` has a value for `c2` in its savelist, it will be used to set up the new value of `c2` before executing the exception handler. In particular, if the exception handler invokes `THROWANY`, it will rethrow the original exception with the restored value of `c2`. This trick enables the exception handler to handle only some exceptions, and pass the rest to an outer exception handler.

4.5.5. Default exception handler. When an instance of TVM is created, `c2` contains a reference to the "default exception handler continuation", which is an `ec_fatal` extraordinary continuation (cf. **4.1.5**). Its execution leads to the termination of the execution of TVM, with the arguments v and n of the exception returned to the outside caller. In the context of the TON Blockchain, n will be stored as a part of the transaction's result.

4.5.6. `TRY` primitive. A `TRY` primitive can be used to implement C++-like exception handling. This primitive accepts two continuations, c and c'. It stores the old value of `c2` into the savelist of c', sets `c2` to c', and executes c just as `EXECUTE` would, but additionally saving the old value of `c2` into the savelist of the new `c0` as well. Usually a version of the `TRY` primitive with an explicit number of arguments n'' passed to the continuation c is used.

The net result is roughly equivalent to C++'s `try { ` c ` } catch(...) { ` c' ` }` operator.

4.5.7. List of predefined exceptions. Predefined exceptions of TVM correspond to exception numbers n in the range 0–31. They include:

- *Normal termination* ($n = 0$) — Should never be generated, but it is useful for some tricks.
- *Alternative termination* ($n = 1$) — Again, should never be generated.
- *Stack underflow* ($n = 2$) — Not enough arguments in the stack for a primitive.
- *Stack overflow* ($n = 3$) — More values have been stored on a stack than allowed by this version of TVM.
- *Integer overflow* ($n = 4$) — Integer does not fit into $-2^{256} \leq x < 2^{256}$, or a division by zero has occurred.
- *Range check error* ($n = 5$) — Integer out of expected range.
- *Invalid opcode* ($n = 6$) — Instruction cannot be decoded.
- *Type check error* ($n = 7$) — An argument to a primitive is of incorrect value type.
- *Cell overflow* ($n = 8$) — Error in one of the serialization primitives.
- *Cell underflow* ($n = 9$) — Deserialization error.
- *Dictionary error* ($n = 10$) — Error while deserializing a dictionary object.
- *Unknown error* ($n = 11$) — Unknown error, may be thrown by user programs.

- *Fatal error* ($n = 12$) — Thrown by TVM in situations deemed impossible.
- *Out of gas* ($n = 13$) — Thrown by TVM when the remaining gas (g_r) becomes negative. This exception usually cannot be caught and leads to an immediate termination of TVM.

Most of these exceptions have no parameter (i.e., use a zero integer instead). The order in which these exceptions are checked is outlined below in **4.5.8**.

4.5.8. Order of stack underflow, type check, and range check exceptions. All TVM primitives first check whether the stack contains the required number of arguments, generating a stack underflow exception if this is not the case. Only then are the type tags of the arguments and their ranges (e.g., if a primitive expects an argument not only to be an *Integer*, but also to be in the range from 0 to 256) checked, starting from the value in the top of the stack (the last argument) and proceeding deeper into the stack. If an argument's type is incorrect, a type-checking exception is generated; if the type is correct, but the value does not fall into the expected range, a range check exception is generated.

Some primitives accept a variable number of arguments, depending on the values of some small fixed subset of arguments located near the top of the stack. In this case, the above procedure is first run for all arguments from this small subset. Then it is repeated for the remaining arguments, once their number and types have been determined from the arguments already processed.

4.6 Functions, recursion, and dictionaries

4.6.1. The problem of recursion. The conditional and iterated execution primitives described in **4.2**—along with the unconditional branch, call, and return primitives described in **4.1**— enable one to implement more or less arbitrary code with nested loops and conditional expressions, with one notable exception: one can only create new constant continuations from parts of the current continuation. (In particular, one cannot invoke a subroutine from itself in this way.) Therefore, the code being executed—i.e., the current continuation—gradually becomes smaller and smaller.[23]

[23] An important point here is that the tree of cells representing a TVM program cannot have cyclic references, so using `CALLREF` along with a reference to a cell higher up the tree

4.6.2. Y-combinator solution: pass a continuation as an argument to itself. One way of dealing with the problem of recursion is by passing a copy of the continuation representing the body of a recursive function as an extra argument to itself. Consider, for example, the following code for a factorial function:

```
71      PUSHINT 1
9C      PUSHCONT {
22        PUSH s2
72        PUSHINT 2
B9        LESS
DC        IFRET
59        ROTREV
21        PUSH s1
A8        MUL
01        SWAP
A5        DEC
02        XCHG s2
20        DUP
D9        JMPX
        }
20      DUP
D8      EXECUTE
30      DROP
31      NIP
```

This roughly corresponds to defining an auxiliary function *body* with three arguments n, x, and f, such that $body(n, x, f)$ equals x if $n < 2$ and $f(n-1, nx, f)$ otherwise, then invoking $body(n, 1, body)$ to compute the factorial of n. The recursion is then implemented with the aid of the `DUP`; `EXECUTE` construction, or `DUP`; `JMPX` in the case of tail recursion. This trick is equivalent to applying Y-combinator to a function *body*.

4.6.3. A variant of Y-combinator solution. Another way of recursively computing the factorial, more closely following the classical recursive definition

$$fact(n) := \begin{cases} 1 & \text{if } n < 2, \\ n \cdot fact(n-1) & \text{otherwise} \end{cases} \tag{5}$$

would not work.

is as follows:

```
9D      PUSHCONT {
21        OVER
C102      LESSINT 2
92        PUSHCONT {
5B          2DROP
71          PUSHINT 1
          }
E0        IFJMP
21        OVER
A5        DEC
01        SWAP
20        DUP
D8        EXECUTE
A8        MUL
        }
20      DUP
D9      JMPX
```

This definition of the factorial function is two bytes shorter than the previous one, but it uses general recursion instead of tail recursion, so it cannot be easily transformed into a loop.

4.6.4. Comparison: non-recursive definition of the factorial function. Incidentally, a non-recursive definition of the factorial with the aid of a `REPEAT` loop is also possible, and it is much shorter than both recursive definitions:

```
71      PUSHINT 1
01      SWAP
20      DUP
94      PUSHCONT {
66        TUCK
A8        MUL
01        SWAP
A5        DEC
        }
E4      REPEAT
30      DROP
```

4.6.5. Several mutually recursive functions. If one has a collection $f_1, \ldots, f_n$ of mutually recursive functions, one can use the same trick by passing the whole collection of continuations $\{f_i\}$ in the stack as an extra n arguments to each of these functions. However, as n grows, this becomes more and more cumbersome, since one has to reorder these extra arguments in the stack to work with the "true" arguments, and then push their copies into the top of the stack before any recursive call.

4.6.6. Combining several functions into one tuple. One might also combine a collection of continuations representing functions $f_1, \ldots, f_n$ into a "tuple" $\mathbf{f} := (f_1, \ldots, f_n)$, and pass this tuple as one stack element $\mathbf{f}$. For instance, when $n \leq 4$, each function can be represented by a cell $\tilde{f}_i$ (along with the tree of cells rooted in this cell), and the tuple may be represented by a cell $\tilde{\mathbf{f}}$, which has references to its component cells $\tilde{f}_i$. However, this would lead to the necessity of "unpacking" the needed component from this tuple before each recursive call.

4.6.7. Combining several functions into a selector function. Another approach is to combine several functions $f_1, \ldots, f_n$ into one "selector function" f, which takes an extra argument i, $1 \leq i \leq n$, from the top of the stack, and invokes the appropriate function f_i. Stack machines such as TVM are well-suited to this approach, because they do not require the functions f_i to have the same number and types of arguments. Using this approach, one would need to pass only one extra argument, f, to each of these functions, and push into the stack an extra argument i before each recursive call to f to select the correct function to be called.

4.6.8. Using a dedicated register to keep the selector function. However, even if we use one of the two previous approaches to combine all functions into one extra argument, passing this argument to all mutually recursive functions is still quite cumbersome and requires a lot of additional stack manipulation operations. Because this argument changes very rarely, one might use a dedicated register to keep it and transparently pass it to all functions called. This is the approach used by TVM by default.

4.6.9. Special register `c3` for the selector function. In fact, TVM uses a dedicated register `c3` to keep the continuation representing the current or global "selector function", which can be used to invoke any of a family of mutually recursive functions. Special primitives `CALL` *nn* or `CALLDICT` *nn*

(cf. **A.7.7**) are equivalent to `PUSHINT` *nn*; `PUSH c3`; `EXECUTE`, and similarly `JMP` *nn* or `JMPDICT` *nn* are equivalent to `PUSHINT` *nn*; `PUSH c3`; `JMPX`. In this way a TVM program, which ultimately is a large collection of mutually recursive functions, may initialize `c3` with the correct selector function representing the family of all the functions in the program, and then use `CALL` *nn* to invoke any of these functions by its index (sometimes also called the *selector* of a function).

4.6.10. Initialization of `c3`. A TVM program might initialize `c3` by means of a `POP c3` instruction. However, because this usually is the very first action undertaken by a program (e.g., a smart contract), TVM makes some provisions for the automatic initialization of `c3`. Namely, `c3` is initialized by the code (the initial value of `cc`) of the program itself, and an extra zero (or, in some cases, some other predefined number *s*) is pushed into the stack before the program's execution. This is approximately equivalent to invoking `JMPDICT 0` (or `JMPDICT` *s*) at the very beginning of a program—i.e., the function with index zero is effectively the `main()` function for the program.

4.6.11. Creating selector functions and `switch` statements. TVM makes special provisions for simple and concise implementation of selector functions (which usually constitute the top level of a TVM program) or, more generally, arbitrary `switch` or `case` statements (which are also useful in `TVM` programs). The most important primitives included for this purpose are `IFBITJMP`, `IFNBITJMP`, `IFBITJMPREF`, and `IFNBITJMPREF` (cf. **A.7.2**). They effectively enable one to combine subroutines, kept either in separate cells or as subslices of certain cells, into a binary decision tree with decisions made according to the indicated bits of the integer passed in the top of the stack.

Another instruction, useful for the implementation of sum-product types, is `PLDUZ` (cf. **A.6.2**). This instruction preloads the first several bits of a *Slice* into an *Integer*, which can later be inspected by `IFBITJMP` and other similar instructions.

4.6.12. Alternative: using a hashmap to select the correct function. Yet another alternative is to use a *Hashmap* (cf. **3.3**) to hold the "collection" or "dictionary" of the code of all functions in a program, and use the hashmap lookup primitives (cf. **A.9**) to select the code of the required function, which can then be `BLESS`ed into a continuation (cf. **A.7.5**) and executed. Special combined "lookup, bless, and execute" primitives, such as `DICTIGETJMP` and `DICTIGETEXEC`, are also available (cf. **A.9.10**). This approach may be more

efficient for larger programs and `switch` statements.

5 Codepages and instruction encoding

This chapter describes the codepage mechanism, which allows TVM to be flexible and extendable while preserving backward compatibility with respect to previously generated code.

We also discuss some general considerations about instruction encodings (applicable to arbitrary machine code, not just TVM), as well as the implications of these considerations for TVM and the choices made while designing TVM's (experimental) codepage zero. The instruction encodings themselves are presented later in Appendix **A**.

5.1 Codepages and interoperability of different TVM versions

The *codepages* are an essential mechanism of backward compatibility and of future extensions to TVM. They enable transparent execution of code written for different revisions of TVM, with transparent interaction between instances of such code. The mechanism of the codepages, however, is general and powerful enough to enable some other originally unintended applications.

5.1.1. Codepages in continuations. Every ordinary continuation contains a 16-bit *codepage* field `cp` (cf. **4.1.1**), which determines the codepage that will be used to execute its code. If a continuation is created by a `PUSHCONT` (cf. **4.2.3**) or similar primitive, it usually inherits the current codepage (i.e., the codepage of `cc`).[24]

5.1.2. Current codepage. The current codepage `cp` (cf. **1.4**) is the codepage of the current continuation `cc`. It determines the way the next instruction will be decoded from `cc.code`, the remainder of the current continuation's code. Once the instruction has been decoded and executed, it determines the next value of the current codepage. In most cases, the current codepage is left unchanged.

On the other hand, all primitives that switch the current continuation load the new value of `cp` from the new current continuation. In this way, all code in continuations is always interpreted exactly as it was intended to be.

[24]This is not exactly true. A more precise statement is that usually the codepage of the newly-created continuation is a known function of the current codepage.

5.1.3. Different versions of TVM may use different codepages. Different versions of TVM may use different codepages for their code. For example, the original version of TVM might use codepage zero. A newer version might use codepage one, which contains all the previously defined opcodes, along with some newly defined ones, using some of the previously unused opcode space. A subsequent version might use yet another codepage, and so on.

However, a newer version of TVM will execute old code for codepage zero exactly as before. If the old code contained an opcode used for some new operations that were undefined in the original version of TVM, it will still generate an invalid opcode exception, because the new operations are absent in codepage zero.

5.1.4. Changing the behavior of old operations. New codepages can also change the effects of some operations present in the old codepages while preserving their opcodes and mnemonics.

For example, imagine a future 513-bit upgrade of TVM (replacing the current 257-bit design). It might use a 513-bit *Integer* type within the same arithmetic primitives as before. However, while the opcodes and instructions in the new codepage would look exactly like the old ones, they would work differently, accepting 513-bit integer arguments and results. On the other hand, during the execution of the same code in codepage zero, the new machine would generate exceptions whenever the integers used in arithmetic and other primitives do not fit into 257 bits.[25] In this way, the upgrade would not change the behavior of the old code.

5.1.5. Improving instruction encoding. Another application for codepages is to change instruction encodings, reflecting improved knowledge of the actual frequencies of such instructions in the code base. In this case, the new codepage will have exactly the same instructions as the old one, but with different encodings, potentially of differing lengths. For example, one might create an experimental version of the first version of TVM, using a

[25]This is another important mechanism of backward compatibility. All values of newly-added types, as well as values belonging to extended original types that do not belong to the original types (e.g., 513-bit integers that do not fit into 257 bits in the example above), are treated by all instructions (except stack manipulation instructions, which are naturally polymorphic, cf. **2.2.6**) in the old codepages as "values of incorrect type", and generate type-checking exceptions accordingly.

(prefix) bitcode instead of the original bytecode, aiming to achieve higher code density.

5.1.6. Making instruction encoding context-dependent. Another way of using codepages to improve code density is to use several codepages with different subsets of the whole instruction set defined in each of them, or with the whole instruction set defined, but with different length encodings for the same instructions in different codepages.

Imagine, for instance, a "stack manipulation" codepage, where stack manipulation primitives have short encodings at the expense of all other operations, and a "data processing" codepage, where all other operations are shorter at the expense of stack manipulation operations. If stack manipulation operations tend to come one after another, we can automatically switch to "stack manipulation" codepage after executing any such instruction. When a data processing instruction occurs, we switch back to "data processing" codepage. If conditional probabilities of the class of the next instruction depending on the class of the previous instruction are considerably different from corresponding unconditional probabilities, this technique—automatically switching into stack manipulation mode to rearrange the stack with shorter instructions, then switching back—might considerably improve the code density.

5.1.7. Using codepages for status and control flags. Another potential application of multiple codepages inside the same revision of TVM consists in switching between several codepages depending on the result of the execution of some instructions.

For example, imagine a version of TVM that uses two new codepages, 2 and 3. Most operations do not change the current codepage. However, the integer comparison operations will switch to codepage 2 if the condition is false, and to codepage 3 if it is true. Furthermore, a new operation `?EXECUTE`, similar to `EXECUTE`, will indeed be equivalent to `EXECUTE` in codepage 3, but will instead be a `DROP` in codepage 2. Such a trick effectively uses bit 0 of the current codepage as a status flag.

Alternatively, one might create a couple of codepages—say, 4 and 5—which differ only in their cell deserialisation primitives. For instance, in codepage 4 they might work as before, while in codepage 5 they might deserialize data not from the beginning of a *Slice*, but from its end. Two new instructions—say, `CLD` and `STD`—might be used for switching to codepage 4

or codepage 5. Clearly, we have now described a status flag, affecting the execution of some instructions in a certain new manner.

5.1.8. Setting the codepage in the code itself. For convenience, we reserve some opcode in all codepages—say, `FFFF` n—for the instruction `SETCP` n, with n from 0 to 255. Then by inserting such an instruction into the very beginning of (the main function of) a program (e.g., a TON Blockchain smart contract) or a library function, we can ensure that the code will always be executed in the intended codepage.

5.2 Instruction encoding

This section discusses the general principles of instruction encoding valid for all codepages and all versions of TVM. Later, **5.3** discusses the choices made for the experimental "codepage zero".

5.2.1. Instructions are encoded by a binary prefix code. All complete instructions (i.e., instructions along with all their parameters, such as the names of stack registers $s(i)$ or other embedded constants) of a TVM codepage are encoded by a *binary prefix code*. This means that a (finite) binary string (i.e., a bitstring) corresponds to each complete instruction, in such a way that binary strings corresponding to different complete instructions do not coincide, and no binary string among the chosen subset is a prefix of another binary string from this subset.

5.2.2. Determining the first instruction from a code stream. As a consequence of this encoding method, any binary string admits at most one prefix, which is an encoding of some complete instruction. In particular, the code `cc.code` of the current continuation (which is a *Slice*, and thus a bitstring along with some cell references) admits at most one such prefix, which corresponds to the (uniquely determined) instruction that TVM will execute first. After execution, this prefix is removed from the code of the current continuation, and the next instruction can be decoded.

5.2.3. Invalid opcode. If no prefix of `cc.code` encodes a valid instruction in the current codepage, an *invalid opcode exception* is generated (cf. **4.5.7**). However, the case of an empty `cc.code` is treated separately as explained in **4.1.4** (the exact behavior may depend on the current codepage).

5.2.4. Special case: end-of-code padding. As an exception to the above rule, some codepages may accept some values of `cc.code` that are too short to be valid instruction encodings as additional variants of `NOP`, thus effectively using the same procedure for them as for an empty `cc.code`. Such bitstrings may be used for padding the code near its end.

For example, if binary string `00000000` (i.e., `x00`, cf. **1.0.3**) is used in a codepage to encode `NOP`, its proper prefixes cannot encode any instructions. So this codepage may accept `0`, `00`, `000`, ..., `0000000` as variants of `NOP` if this is all that is left in `cc.code`, instead of generating an invalid opcode exception.

Such a padding may be useful, for example, if the `PUSHCONT` primitive (cf. **4.2.3**) creates only continuations with code consisting of an integral number of bytes, but not all instructions are encoded by an integral number of bytes.

5.2.5. TVM code is a bitcode, not a bytecode. Recall that TVM is a bit-oriented machine in the sense that its *Cell*s (and *Slice*s) are naturally considered as sequences of bits, not just of octets (bytes), cf. **3.2.5**. Because the TVM code is also kept in cells (cf. **3.1.9** and **4.1.4**), there is no reason to use only bitstrings of length divisible by eight as encodings of complete instructions. In other words, generally speaking, *the TVM code is a bitcode, not a bytecode.*

That said, some codepages (such as our experimental codepage zero) may opt to use a bytecode (i.e., to use only encodings consisting of an integral number of bytes)—either for simplicity, or for the ease of debugging and of studying memory (i.e., cell) dumps.[26]

5.2.6. Opcode space used by a complete instruction. Recall from coding theory that the lengths of bitstrings l_i used in a binary prefix code satisfy Kraft–McMillan inequality $\sum_i 2^{-l_i} \leq 1$. This is applicable in particular to the (complete) instruction encoding used by a TVM codepage. We say that *a particular complete instruction* (or, more precisely, *the encoding of a complete instruction*) *utilizes the portion* 2^{-l} *of the opcode space*, if it is encoded by an l-bit string. One can see that all complete instructions together utilize at most 1 (i.e., "at most the whole opcode space").

[26]If the cell dumps are hexadecimal, encodings consisting of an integral number of hexadecimal digits (i.e., having length divisible by four bits) might be equally convenient.

5.2.7. Opcode space used by an instruction, or a class of instructions. The above terminology is extended to instructions (considered with all admissible values of their parameters), or even classes of instructions (e.g., all arithmetic instructions). We say that an (incomplete) instruction, or a class of instructions, occupies portion α of the opcode space, if α is the sum of the portions of the opcode space occupied by all complete instructions belonging to that class.

5.2.8. Opcode space for bytecodes. A useful approximation of the above definitions is as follows: Consider all 256 possible values for the first byte of an instruction encoding. Suppose that k of these values correspond to the specific instruction or class of instructions we are considering. Then this instruction or class of instructions occupies approximately the portion $k/256$ of the opcode space.

This approximation shows why all instructions cannot occupy together more than the portion $256/256 = 1$ of the opcode space, at least without compromising the uniqueness of instruction decoding.

5.2.9. Almost optimal encodings. Coding theory tells us that in an optimally dense encoding, the portion of the opcode space used by a complete instruction (2^{-l}, if the complete instruction is encoded in l bits) should be approximately equal to the probability or frequency of its occurrence in real programs.[27] The same should hold for (incomplete) instructions, or primitives (i.e., generic instructions without specified values of parameters), and for classes of instructions.

5.2.10. Example: stack manipulation primitives. For instance, if stack manipulation instructions constitute approximately half of all instructions in a typical TVM program, one should allocate approximately half of the opcode space for encoding stack manipulation instructions. One might reserve the first bytes ("opcodes") `0x00`–`0x7f` for such instructions. If a quarter of these instructions are `XCHG`, it would make sense to reserve `0x00`–`0x1f` for `XCHG`s. Similarly, if half of all `XCHG`s involve the top of stack `s0`, it would make sense to use `0x00`–`0x0f` to encode `XCHG s0,s`(i).

5.2.11. Simple encodings of instructions. In most cases, *simple* encodings of complete instructions are used. Simple encodings begin with a fixed

[27] Notice that it is the probability of occurrence in the code that counts, not the probability of being executed. An instruction occurring in the body of a loop executed a million times is still counted only once.

bitstring called the *opcode* of the instruction, followed by, say, 4-bit fields containing the indices i of stack registers `s`(i) specified in the instruction, followed by all other constant (literal, immediate) parameters included in the complete instruction. While simple encodings may not be exactly optimal, they admit short descriptions, and their decoding and encoding can be easily implemented.

If a (generic) instruction uses a simple encoding with an l-bit opcode, then the instruction will utilize 2^{-l} portion of the opcode space. This observation might be useful for considerations described in **5.2.9** and **5.2.10**.

5.2.12. Optimizing code density further: Huffman codes. One might construct optimally dense binary code for the set of all complete instructions, provided their probabilities or frequences in real code are known. This is the well-known Huffman code (for the given probability distribution). However, such code would be highly unsystematic and hard to decode.

5.2.13. Practical instruction encodings. In practice, instruction encodings used in TVM and other virtual machines offer a compromise between code density and ease of encoding and decoding. Such a compromise may be achieved by selecting simple encodings (cf. **5.2.11**) for all instructions (maybe with separate simple encodings for some often used variants, such as `XCHG s0,s`(i) among all `XCHG s`(i)`,s`(j)), and allocating opcode space for such simple encodings using the heuristics outlined in **5.2.9** and **5.2.10**; this is the approach currently used in TVM.

5.3 Instruction encoding in codepage zero

This section provides details about the experimental instruction encoding for codepage zero, as described elsewhere in this document (cf. Appendix **A**) and used in the preliminary test version of TVM.

5.3.1. Upgradability. First of all, even if this preliminary version somehow gets into the production version of the TON Blockchain, the codepage mechanism (cf. **5.1**) enables us to introduce better versions later without compromising backward compatibility.[28] So in the meantime, we are free to experiment.

[28]Notice that any modifications after launch cannot be done unilaterally; rather they would require the support of at least two-thirds of validators.

5.3.2. Choice of instructions. We opted to include many "experimental" and not strictly necessary instructions in codepage zero just to see how they might be used in real code. For example, we have both the basic (cf. **2.2.1**) and the compound (cf. **2.2.3**) stack manipulation primitives, as well as some "unsystematic" ones such as `ROT` (mostly borrowed from Forth). If such primitives are rarely used, their inclusion just wastes some part of the opcode space and makes the encodings of other instructions slightly less effective, something we can afford at this stage of TVM's development.

5.3.3. Using experimental instructions. Some of these experimental instructions have been assigned quite long opcodes, just to fit more of them into the opcode space. One should not be afraid to use them just because they are long; if these instructions turn out to be useful, they will receive shorter opcodes in future revisions. Codepage zero is not meant to be fine-tuned in this respect.

5.3.4. Choice of bytecode. We opted to use a bytecode (i.e., to use encodings of complete instructions of lengths divisible by eight). While this may not produce optimal code density, because such a length restriction makes it more difficult to match portions of opcode space used for the encoding of instructions with estimated frequencies of these instructions in TVM code (cf. **5.2.11** and **5.2.9**), such an approach has its advantages: it admits a simpler instruction decoder and simplifies debugging (cf. **5.2.5**).

After all, we do not have enough data on the relative frequencies of different instructions right now, so our code density optimizations are likely to be very approximate at this stage. The ease of debugging and experimenting and the simplicity of implementation are more important at this point.

5.3.5. Simple encodings for all instructions. For similar reasons, we opted to use simple encodings for all instructions (cf. **5.2.11** and **5.2.13**), with separate simple encodings for some very frequently used subcases as outlined in **5.2.13**. That said, we tried to distribute opcode space using the heuristics described in **5.2.9** and **5.2.10**.

5.3.6. Lack of context-dependent encodings. This version of TVM also does not use context-dependent encodings (cf. **5.1.6**). They may be added at a later stage, if deemed useful.

5.3.7. The list of all instructions. The list of all instructions available in

codepage zero, along with their encodings and (in some cases) short descriptions, may be found in Appendix **A**.

References

[1] N. Durov, *Telegram Open Network*, 2017.

A Instructions and opcodes

This appendix lists all instructions available in the (experimental) codepage zero of TVM, as explained in **5.3**.

We list the instructions in lexicographical opcode order. However, the opcode space is distributed in such way as to make all instructions in each category (e.g., arithmetic primitives) have neighboring opcodes. So we first list a number of stack manipulation primitives, then constant primitives, arithmetic primitives, comparison primitives, cell primitives, continuation primitives, dictionary primitives, and finally application-specific primitives.

We use hexadecimal notation (cf. **1.0**) for bitstrings. Stack registers $\mathtt{s}(i)$ usually have $0 \leq i \leq 15$, and i is encoded in a 4-bit field (or, on a few rare occasions, in an 8-bit field). Other immediate parameters are usually 4-bit, 8-bit, or variable length.

The stack notation described in **2.1.10** is extensively used throughout this appendix.

A.1 Gas prices

The gas price for most primitives equals the *basic gas price*, computed as $P_b := 10 + b + 5r$, where b is the instruction length in bits and r is the number of cell references included in the instruction. When the gas price of an instruction differs from this basic price, it is indicated in parentheses after its mnemonics, either as *(x)*, meaning that the total gas price equals x, or as *(+x)*, meaning $P_b + x$. Apart from integer constants, the following expressions may appear:

- C_r — The total price of "reading" cells (i.e., transforming cell references into cell slices). Currently equal to 20 gas units per cell.
- L — The total price of loading cells. Depends on the loading action required.
- B_w — The total price of creating new *Builder*s. Currently equal to 100 gas units per builder.
- C_w — The total price of creating new *Cell*s from *Builder*s). Currently equal to 100 gas units per cell.

A.2 Stack manipulation primitives

This section includes both the basic (cf. **2.2.1**) and the compound (cf. **2.2.3**) stack manipulation primitives, as well as some "unsystematic" ones. Some compound stack manipulation primitives, such as `XCPU` or `XCHG2`, turn out to have the same length as an equivalent sequence of simpler operations. We have included these primitives regardless, so that they can easily be allocated shorter opcodes in a future revision of TVM—or removed for good.

Some stack manipulation instructions have two mnemonics: one Forth-style (e.g., `-ROT`), the other conforming to the usual rules for identifiers (e.g., `ROTREV`). Whenever a stack manipulation primitive (e.g., `PICK`) accepts an integer parameter n from the stack, it must be within the range $0 \ldots 255$; otherwise a range check exception happens before any further checks.

A.2.1. Basic stack manipulation primitives.

- `00` — `NOP`, does nothing.
- `01` — `XCHG s1`, also known as `SWAP`.
- `0`i — `XCHG s`(i) or `XCHG s0,s`(i), interchanges the top of the stack with `s`(i), $1 \leq i \leq 15$.
- `10`ij — `XCHG s`(i)`,s`(j), $1 \leq i < j \leq 15$, interchanges `s`(i) with `s`(j).
- `11`ii — `XCHG s0,s`(ii), with $0 \leq ii \leq 255$.
- `1`i — `XCHG s1,s`(i), $2 \leq i \leq 15$.
- `2`i — `PUSH s`(i), $0 \leq i \leq 15$, pushes a copy of the old `s`(i) into the stack.
- `20` — `PUSH s0`, also known as `DUP`.
- `21` — `PUSH s1`, also known as `OVER`.
- `3`i — `POP` $s(i)$, $0 \leq i \leq 15$, pops the old top-of-stack value into the old `s`(i).
- `30` — `POP s0`, also known as `DROP`, discards the top-of-stack value.
- `31` — `POP s1`, also known as `NIP`.

A.2.2. Compound stack manipulation primitives. Parameters i, j, and k of the following primitives all are 4-bit integers in the range $0 \ldots 15$.

- $4ijk$ — `XCHG3 s`(i)`,s`(j)`,s`(k), equivalent to `XCHG s2,s`(i); `XCHG s1,s`(j); `XCHG s0,s`(k), with $0 \leq i, j, k \leq 15$.
- $50ij$ — `XCHG2 s`(i)`,s`(j), equivalent to `XCHG s1,s`(i); `XCHG s`(j).
- $51ij$ — `XCPU s`(i)`,s`(j), equivalent to `XCHG s`(i); `PUSH s`(j).
- $52ij$ — `PUXC s`(i)`,s`$(j-1)$, equivalent to `PUSH s`(i); `SWAP`; `XCHG s`(j).
- $53ij$ — `PUSH2 s`(i)`,s`(j), equivalent to `PUSH s`(i); `PUSH s`$(j+1)$.
- $540ijk$ — `XCHG3 s`(i)`,s`(j)`,s`(k) (long form).
- $541ijk$ — `XC2PU s`(i)`,s`(j)`,s`(k), equivalent to `XCHG2 s`(i)`,s`(j); `PUSH s`(k).
- $542ijk$ — `XCPUXC s`(i)`,s`(j)`,s`$(k-1)$, equivalent to `XCHG s1,s`(i); `PUXC s`(j)`,s`$(k-1)$.
- $543ijk$ — `XCPU2 s`(i)`,s`(j)`,s`(k), equivalent to `XCHG s`(i); `PUSH2 s`(j)`,s`(k).
- $544ijk$ — `PUXC2 s`(i)`,s`$(j-1)$`,s`$(k-1)$, equivalent to `PUSH s`(i); `XCHG s2`; `XCHG2 s`(j)`,s`(k).
- $545ijk$ — `PUXCPU s`(i)`,s`$(j-1)$`,s`$(k-1)$, equivalent to `PUXC s`(i)`,s`$(j-1)$; `PUSH s`(k).
- $546ijk$ — `PU2XC s`(i)`,s`$(j-1)$`,s`$(k-2)$, equivalent to `PUSH s`(i); `SWAP`; `PUXC s`(j)`,s`$(k-1)$.
- $547ijk$ — `PUSH3 s`(i)`,s`(j)`,s`(k), equivalent to `PUSH s`(i); `PUSH2 s`$(j+1)$`,s`$(k+1)$.
- `54C_` — unused.

A.2.3. Exotic stack manipulation primitives.

- $55ij$ — `BLKSWAP` $i+1, j+1$, permutes two blocks `s`$(j+i+1) \ldots$ `s`$(j+1)$ and `s`$(j) \ldots$ `s0`, for $0 \leq i, j \leq 15$. Equivalent to `REVERSE` $i+1, j+1$; `REVERSE` $j+1$`,0`; `REVERSE` $i+j+2$`,0`.

- 56ii — PUSH s(ii) for $0 \leq ii \leq 255$.
- 57ii — POP s(ii) for $0 \leq ii \leq 255$.
- 58 — ROT ($a\ b\ c - b\ c\ a$), equivalent to BLKSWAP 1,2 or to XCHG2 s2,s1.
- 59 — ROTREV or -ROT ($a\ b\ c - c\ a\ b$), equivalent to BLKSWAP 2,1 or to XCHG2 s2,s2.
- 5A — SWAP2 or 2SWAP ($a\ b\ c\ d - c\ d\ a\ b$), equivalent to BLKSWAP 2,2 or to XCHG2 s3,s2.
- 5B — DROP2 or 2DROP ($a\ b -$), equivalent to DROP; DROP.
- 5C — DUP2 or 2DUP ($a\ b - a\ b\ a\ b$), equivalent to PUSH2 s1,s0.
- 5D — OVER2 or 2OVER ($a\ b\ c\ d - a\ b\ c\ d\ a\ b$), equivalent to PUSH2 s3,s2.
- 5Eij — REVERSE $i+2$,j, reverses the order of s($j+i+1$)... s(j) for $0 \leq i, j \leq 15$; equivalent to a sequence of $\lfloor i/2 \rfloor + 1$ XCHGs.
- 5F0i — BLKDROP i, equivalent to DROP performed i times.
- 5Fij — BLKPUSH i,j, equivalent to PUSH s(j) performed i times, $1 \leq i \leq 15$, $0 \leq j \leq 15$.
- 60 — PICK or PUSHX, pops integer i from the stack, then performs PUSH s(i).
- 61 — ROLL, pops integer i from the stack, then performs BLKSWAP 1,i.
- 62 — -ROLL or ROLLREV, pops integer i from the stack, then performs BLKSWAP i,1.
- 63 — BLKSWX, pops integers i,j from the stack, then performs BLKSWAP i,j.
- 64 — REVX, pops integers i,j from the stack, then performs REVERSE i,j.
- 65 — DROPX, pops integer i from the stack, then performs BLKDROP i.
- 66 — TUCK ($ab - bab$), equivalent to SWAP; OVER or to XCPU s1,s1.

- `67` — `XCHGX`, pops integer i from the stack, then performs `XCHG s`(i).
- `68` — `DEPTH`, pushes the current depth of the stack.
- `69` — `CHKDEPTH`, pops integer i from the stack, then checks whether there are at least i elements, generating a stack underflow exception otherwise.
- `6A` — `ONLYTOPX`, pops integer i from the stack, then removes all but the top i elements.
- `6B` — `ONLYX`, pops integer i from the stack, then leaves only the bottom i elements. Approximately equivalent to `DEPTH`; `SWAP`; `SUB`; `DROPX`.
- `6C`–`6F` — reserved for stack operations.

A.3 Constant, or literal primitives

The following primitives push into the stack one constant of some type and range.

A.3.1. Integer and boolean constants.

- `7`i — `PUSHINT` x with $-5 \leq x \leq 10$, pushes integer x into the stack; here i equals four lower-order bits of x (i.e., $i = x \bmod 16$).
- `70` — `ZERO`, `FALSE`, or `PUSHINT 0`, pushes a zero.
- `71` — `ONE` or `PUSHINT 1`.
- `72` — `TWO` or `PUSHINT 2`.
- `7A` — `TEN` or `PUSHINT` 10.
- `7F` — `TRUE` or `PUSHINT -1`.
- `80`xx — `PUSHINT` xx with $-128 \leq xx \leq 127$.
- `81`$xxxx$ — `PUSHINT` $xxxx$ with $-2^{15} \leq xxxx < 2^{15}$ a signed 16-bit big-endian integer.
- `81FC18` — `PUSHINT` -1000.

- 82$lxxx$ — PUSHINT xxx, where 5-bit $0 \leq l \leq 30$ determines the length $n = 8l + 19$ of signed big-endian integer xxx. The total length of this instruction is $l + 4$ bytes or $n + 13 = 8l + 32$ bits.
- 821005F5E100 — PUSHINT 10^8.
- 83xx — PUSHPOW2 $xx + 1$, (quietly) pushes 2^{xx+1} for $0 \leq xx \leq 255$.
- 83FF — PUSHNAN, pushes a NaN.
- 84xx — PUSHPOW2DEC $xx + 1$, pushes $2^{xx+1} - 1$ for $0 \leq xx \leq 255$.
- 85xx — PUSHNEGPOW2 $xx + 1$, pushes -2^{xx+1} for $0 \leq xx \leq 255$.
- 86, 87 — reserved for integer constants.

A.3.2. Constant slices, continuations, cells, and references.

- 88 — PUSHREF, pushes the first reference of cc.code into the stack as a *Cell* (and removes this reference from the current continuation).
- 89 — PUSHREFSLICE, similar to PUSHREF, but converts the cell into a *Slice*.
- 8A — PUSHREFCONT, similar to PUSHREFSLICE, but makes a simple ordinary *Continuation* out of the cell.
- 8Bxsss — PUSHSLICE sss, pushes the (prefix) subslice of cc.code consisting of its first $8x + 4$ bits and no references (i.e., essentially a bitstring), where $0 \leq x \leq 15$. A completion tag is assumed, meaning that all trailing zeroes and the last binary one (if present) are removed from this bitstring. If the original bitstring consists only of zeroes, an empty slice will be pushed.
- 8B08 — PUSHSLICE x8_, pushes an empty slice (bitstring ‘’).
- 8B04 — PUSHSLICE x4_, pushes bitstring ‘0’.
- 8B0C — PUSHSLICE xC_, pushes bitstring ‘1’.
- 8Crxxssss — PUSHSLICE ssss, pushes the (prefix) subslice of cc.code consisting of its first $1 \leq r + 1 \leq 4$ references and up to first $8xx + 1$ bits of data, with $0 \leq xx \leq 31$. A completion tag is also assumed.

- `8C01` is equivalent to `PUSHREFSLICE`.
- `8Drxxssssss` — `PUSHSLICE sssss`, pushes the subslice of `cc.code` consisting of $0 \le r \le 4$ references and up to $8xx + 6$ bits of data, with $0 \le xx \le 127$. A completion tag is assumed.
- `8DE_` — unused (reserved).
- `8F_rxxcccc` — `PUSHCONT cccc`, where *cccc* is the simple ordinary continuation made from the first $0 \le r \le 3$ references and the first $0 \le xx \le 127$ bytes of `cc.code`.
- `9xccc` — `PUSHCONT ccc`, pushes an x-byte continuation for $0 \le x \le 15$.

A.4 Arithmetic primitives

A.4.1. Addition, subtraction, multiplication.

- `A0` — `ADD` ($x\ y - x + y$), adds together two integers.
- `A1` — `SUB` ($x\ y - x - y$).
- `A2` — `SUBR` ($x\ y - y - x$), equivalent to `SWAP`; `SUB`.
- `A3` — `NEGATE` ($x - -x$), equivalent to `MULCONST` -1 or to `ZERO; SUBR`. Notice that it triggers an integer overflow exception if $x = -2^{256}$.
- `A4` — `INC` ($x - x + 1$), equivalent to `ADDCONST 1`.
- `A5` — `DEC` ($x - x - 1$), equivalent to `ADDCONST` -1.
- `A6`*cc* — `ADDCONST` *cc* ($x - x + cc$), $-128 \le cc \le 127$.
- `A7`*cc* — `MULCONST` *cc* ($x - x \cdot cc$), $-128 \le cc \le 127$.
- `A8` — `MUL` ($x\ y - xy$).

A.4.2. Division.

The general encoding of a `DIV`, `DIVMOD`, or `MOD` operation is `A9`*mscdf*, with an optional pre-multiplication and an optional replacement of the division or multiplication by a shift. Variable one- or two-bit fields m, s, c, d, and f are as follows:

- $0 \le m \le 1$ — Indicates whether there is pre-multiplication (`MULDIV` operation and its variants), possibly replaced by a left shift.
- $0 \le s \le 2$ — Indicates whether either the multiplication or the division have been replaced by shifts: $s = 0$—no replacement, $s = 1$—division replaced by a right shift, $s = 2$—multiplication replaced by a left shift (possible only for $m = 1$).
- $0 \le c \le 1$ — Indicates whether there is a constant one-byte argument tt for the shift operator (if $s \neq 0$). For $s = 0$, $c = 0$. If $c = 1$, then $0 \le tt \le 255$, and the shift is performed by $tt + 1$ bits.
- $1 \le d \le 3$ — Indicates which results of division are required: 1—only the quotient, 2—only the remainder, 3—both.
- $0 \le f \le 2$ — Rounding mode: 0—floor, 1—nearest integer, 2—ceiling (cf. **1.5.6**).

Examples:

- `A904` — `DIV` ($x\ y - q := \lfloor x/y \rfloor$).
- `A905` — `DIVR` ($x\ y - q' := \lfloor x/y + 1/2 \rfloor$).
- `A906` — `DIVC` ($x\ y - q'' := \lceil x/y \rceil$).
- `A908` — `MOD` ($x\ y - r$), where $q := \lfloor x/y \rfloor$, $r := x \bmod y := x - yq$.
- `A90C` — `DIVMOD` ($x\ y - q\ r$), where $q := \lfloor x/y \rfloor$, $r := x - yq$.
- `A90D` — `DIVMODR` ($x\ y - q'\ r'$), where $q' := \lfloor x/y + 1/2 \rfloor$, $r' := x - yq'$.
- `A90E` — `DIVMODC` ($x\ y - q''\ r''$), where $q'' := \lceil x/y \rceil$, $r'' := x - yq''$.
- `A924` — same as `RSHIFT`: ($x\ y - \lfloor x \cdot 2^{-y} \rfloor$) for $0 \le y \le 256$.
- `A934`tt — same as `RSHIFT` $tt + 1$: ($x - \lfloor x \cdot 2^{-tt-1} \rfloor$).
- `A938`tt — `MODPOW2` $tt + 1$: ($x - x \bmod 2^{tt+1}$).
- `A985` — `MULDIVR` ($x\ y\ z - q'$), where $q' = \lfloor xy/z + 1/2 \rfloor$.
- `A98C` — `MULDIVMOD` ($x\ y\ z - q\ r$), where $q := \lfloor x \cdot y/z \rfloor$, $r := x \cdot y \bmod z$ (same as `*/MOD` in Forth).

The most useful of these operations are `DIV`, `DIVMOD`, `MOD`, `DIVR`, `DIVC`, `MODPOW2` t, and `RSHIFTR` t (for integer arithmetic); and `MULDIVMOD`, `MULDIV`, `MULDIVR`, `LSHIFTDIVR` t, and `MULRSHIFTR` t (for fixed-point arithmetic).

A.4.3. Shifts, logical operations.

- `AA`cc — `LSHIFT` $cc+1$ (x – $x \cdot 2^{cc+1}$), $0 \le cc \le 255$.
- `AA00` — `LSHIFT 1`, equivalent to `MULCONST 2` or to Forth's `2*`.
- `AB`cc — `RSHIFT` $cc+1$ (x – $\lfloor x \cdot 2^{-cc-1} \rfloor$), $0 \le cc \le 255$.
- `AC` — `LSHIFT` (x y – $x \cdot 2^y$), $0 \le y \le 1023$.
- `AD` — `RSHIFT` (x y – $\lfloor x \cdot 2^{-y} \rfloor$), $0 \le y \le 1023$.
- `AE` — `POW2` (y – 2^y), $0 \le y \le 1023$, equivalent to `ONE`; `SWAP`; `LSHIFT`.
- `AF` — reserved.
- `B0` — `AND` (x y – $x\&y$), bitwise "and" of two signed integers x and y, sign-extended to infinity.
- `B1` — `OR` (x y – $x \vee y$), bitwise "or" of two integers.
- `B2` — `XOR` (x y – $x \oplus y$), bitwise "xor" of two integers.
- `B3` — `NOT` (x – $x \oplus -1 = -1 - x$), bitwise "not" of an integer.
- `B4`cc — `FITS` $cc+1$ (x – x), checks whether x is a $cc+1$-bit signed integer for $0 \le cc \le 255$ (i.e., whether $-2^{cc} \le x < 2^{cc}$). If not, either triggers an integer overflow exception, or replaces x with a `NaN` (quiet version).
- `B400` — `FITS 1` or `CHKBOOL` (x – x), checks whether x is a "boolean value" (i.e., either 0 or -1).
- `B5`cc — `UFITS` $cc+1$ (x – x), checks whether x is a $cc+1$-bit unsigned integer for $0 \le cc \le 255$ (i.e., whether $0 \le x < 2^{cc+1}$).
- `B500` — `UFITS 1` or `CHKBIT`, checks whether x is a binary digit (i.e., zero or one).

- `B600` — `FITSX` ($x\ c$ – x), checks whether x is a c-bit signed integer for $0 \leq c \leq 1023$.
- `B601` — `UFITSX` ($x\ c$ – x), checks whether x is a c-bit unsigned integer for $0 \leq c \leq 1023$.
- `B602` — `BITSIZE` (x – c), computes smallest $c \geq 0$ such that x fits into a c-bit signed integer ($-2^{c-1} \leq c < 2^{c-1}$).
- `B603` — `UBITSIZE` (x – c), computes smallest $c \geq 0$ such that x fits into a c-bit unsigned integer ($0 \leq x < 2^c$), or throws a range check exception.
- `B608` — `MIN` ($x\ y$ – x or y), computes the minimum of two integers x and y.
- `B609` — `MAX` ($x\ y$ – x or y), computes the maximum of two integers x and y.
- `B60A` — `MINMAX` or `INTSORT2` ($x\ y$ – $x\ y$ or $y\ x$), sorts two integers. Quiet version of this operation returns two `NaN`s if any of the arguments are `NaN`s.
- `B60B` — `ABS` (x – $|x|$), computes the absolute value of an integer x.

A.4.4. Quiet arithmetic primitives. We opted to make all arithmetic operations "non-quiet" (signaling) by default, and create their quiet counterparts by means of a prefix. Such an encoding is definitely sub-optimal. It is not yet clear whether it should be done in this way, or in the opposite way by making all arithmetic operations quiet by default, or whether quiet and non-quiet operations should be given opcodes of equal length; this can only be settled by practice.

- `B7xx` — `QUIET` prefix, transforming any arithmetic operation into its "quiet" variant, indicated by prefixing a `Q` to its mnemonic. Such operations return `NaN`s instead of throwing integer overflow exceptions if the results do not fit in *Integer*s, or if one of their arguments is a `NaN`. Notice that this does not extend to shift amounts and other parameters that must be within a small range (e.g., 0–1023). Also notice that this does not disable type-checking exceptions if a value of a type other than *Integer* is supplied.

- B7A0 — QADD ($x\ y - x + y$), always works if x and y are *Integer*s, but returns a NaN if the addition cannot be performed.
- B7A904 — QDIV ($x\ y - \lfloor x/y \rfloor$), returns a NaN if $y = 0$, or if $y = -1$ and $x = -2^{256}$, or if either of x or y is a NaN.
- B7B0 — QAND ($x\ y - x\&y$), bitwise "and" (similar to AND), but returns a NaN if either x or y is a NaN instead of throwing an integer overflow exception. However, if one of the arguments is zero, and the other is a NaN, the result is zero.
- B7B1 — QOR ($x\ y - x \vee y$), bitwise "or". If $x = -1$ or $y = -1$, the result is always -1, even if the other argument is a NaN.
- B7B507 — QUFITS 8 ($x - x'$), checks whether x is an unsigned byte (i.e., whether $0 \leq x < 2^8$), and replaces x with a NaN if this is not the case; leaves x intact otherwise (i.e., if x is an unsigned byte).

A.5 Comparison primitives

A.5.1. Integer comparison. All integer comparison primitives return integer -1 ("true") or 0 ("false") to indicate the result of the comparison. We do not define their "boolean circuit" counterparts, which would transfer control to c0 or c1 depending on the result of the comparison. If needed, such instructions can be simulated with the aid of RETBOOL.

Quiet versions of integer comparison primitives are also available, encoded with the aid of the QUIET prefix (B7). If any of the integers being compared are NaNs, the result of a quiet comparison will also be a NaN ("undefined"), instead of a -1 ("yes") or 0 ("no"), thus effectively supporting ternary logic.

- B8 — SGN ($x - \mathrm{sgn}(x)$), computes the sign of an integer x: -1 if $x < 0$, 0 if $x = 0$, 1 if $x > 0$.
- B9 — LESS ($x\ y - x < y$), returns -1 if $x < y$, 0 otherwise.
- BA — EQUAL ($x\ y - x = y$), returns -1 if $x = y$, 0 otherwise.
- BB — LEQ ($x\ y - x \leq y$).
- BC — GREATER ($x\ y - x > y$).

- `BD` — `NEQ` ($x\ y - x \neq y$), equivalent to `EQUAL`; `NOT`.
- `BE` — `GEQ` ($x\ y - x \geq y$), equivalent to `LESS`; `NOT`.
- `BF` — `CMP` ($x\ y - \mathrm{sgn}(x-y)$), computes the sign of $x-y$: -1 if $x<y$, 0 if $x=y$, 1 if $x>y$. No integer overflow can occur here unless x or y is a `NaN`.
- `C0`yy — `EQINT` yy ($x - x = yy$) for $-2^7 \leq yy < 2^7$.
- `C000` — `ISZERO`, checks whether an integer is zero. Corresponds to Forth's `0=`.
- `C1`yy — `LESSINT` yy ($x - x < yy$) for $-2^7 \leq yy < 2^7$.
- `C100` — `ISNEG`, checks whether an integer is negative. Corresponds to Forth's `0<`.
- `C101` — `ISNPOS`, checks whether an integer is non-positive.
- `C2`yy — `GTINT` yy ($x - x > yy$) for $-2^7 \leq yy < 2^7$.
- `C200` — `ISPOS`, checks whether an integer is positive. Corresponds to Forth's `0>`.
- `C2FF` — `ISNNEG`, checks whether an integer is non-negative.
- `C3`yy — `NEQINT` yy ($x - x \neq yy$) for $-2^7 \leq yy < 2^7$.
- `C4` — `ISNAN` ($x - x =$ `NaN`), checks whether x is a `NaN`.
- `C5` — `CHKNAN` ($x - x$), throws an arithmetic overflow exception if x is a `NaN`.
- `C6` — reserved for integer comparison.

A.5.2. Other comparison.

Most of these "other comparison" primitives actually compare the data portions of *Slice*s as bitstrings.

- `C700` — `SEMPTY` ($s - s = \emptyset$), checks whether a *Slice* s is empty (i.e., contains no bits of data and no cell references).

- C701 — SDEMPTY (s – $s \approx \emptyset$), checks whether *Slice* s has no bits of data.
- C702 — SREMPTY (s – $r(s) = 0$), checks whether *Slice* s has no references.
- C703 — SDFIRST (s – $s_0 = 1$), checks whether the first bit of *Slice* s is a one.
- C704 — SDLEXCMP (s s' – c), compares the data of s lexicographically with the data of s', returning -1, 0, or 1 depending on the result.
- C705 — SDEQ (s s' – $s \approx s'$), checks whether the data parts of s and s' coincide, equivalent to SDLEXCMP; ISZERO.
- C708 — SDPFX (s s' – ?), checks whether s is a prefix of s'.
- C709 — SDPFXREV (s s' – ?), checks whether s' is a prefix of s, equivalent to SWAP; SDPFX.
- C70A — SDPPFX (s s' – ?), checks whether s is a proper prefix of s' (i.e., a prefix distinct from s').
- C70B — SDPPFXREV (s s' – ?), checks whether s' is a proper prefix of s.
- C70C — SDSFX (s s' – ?), checks whether s is a suffix of s'.
- C70D — SDSFXREV (s s' – ?), checks whether s' is a suffix of s.
- C70E — SDPSFX (s s' – ?), checks whether s is a proper suffix of s'.
- C70F — SDPSFXREV (s s' – ?), checks whether s' is a proper suffix of s.
- C710 — SDCNTLEAD0 (s – n), returns the number of leading zeroes in s.
- C711 — SDCNTLEAD1 (s – n), returns the number of leading ones in s.
- C712 — SDCNTTRAIL0 (s – n), returns the number of trailing zeroes in s.
- C713 — SDCNTTRAIL1 (s – n), returns the number of trailing ones in s.

A.6 Cell primitives

The cell primitives are mostly either *cell serialization primitives*, which work with *Builder*s, or *cell deserialization primitives*, which work with *Slice*s.

A.6.1. Cell serialization primitives. All these primitives first check whether there is enough space in the Builder, and only then check the range of the value being serialized.

- `C8` — `NEWC` (– b), creates a new empty *Builder*.
- `C9` — `ENDC` (b – c), converts a *Builder* into an ordinary *Cell*.
- `CA`cc — `STI` $cc+1$ (x b – b'), stores a signed $cc+1$-bit integer x into *Builder* b for $0 \leq cc \leq 255$, throws a range check exception if x does not fit into $cc+1$ bits.
- `CB`cc — `STU` $cc+1$ (x b – b'), stores an unsigned $cc+1$-bit integer x into *Builder* b. In all other respects it is similar to `STI`.
- `CC` — `STREF` (c b – b'), stores a reference to *Cell* c into *Builder* b.
- `CD` — `STBREFR` or `ENDCST` (b b'' – b), equivalent to `ENDC`; `SWAP`; `STREF`.
- `CE` — `STSLICE` (s b – b'), stores *Slice* s into *Builder* b.
- `CF00` — `STIX` (x b l – b'), stores a signed l-bit integer x into b for $0 \leq l \leq 257$.
- `CF01` — `STUX` (x b l – b'), stores an unsigned l-bit integer x into b for $0 \leq l \leq 256$.
- `CF02` — `STIXR` (b x l – b'), similar to `STIX`, but with arguments in a different order.
- `CF03` — `STUXR` (b x l – b'), similar to `STUX`, but with arguments in a different order.
- `CF04` — `STIXQ` (x b l – x b f or b' 0), a quiet version of `STIX`. If there is no space in b, sets $b'=b$ and $f=-1$. If x does not fit into l bits, sets $b'=b$ and $f=1$. If the operation succeeds, b' is the new *Builder* and $f=0$. However, $0 \leq l \leq 257$, with a range check exception if this is not so.

- CF05 — STUXQ ($x\ b\ l$ – $b'\ f$).
- CF06 — STIXRQ ($b\ x\ l$ – $b\ x\ f$ or $b'\ 0$).
- CF07 — STUXRQ ($b\ x\ l$ – $b\ x\ f$ or $b'\ 0$).
- CF08cc — a longer version of STI $cc+1$.
- CF09cc — a longer version of STU $cc+1$.
- CF0Acc — STIR $cc+1$ ($b\ x$ – b'), equivalent to SWAP; STI $cc+1$.
- CF0Bcc — STUR $cc+1$ ($b\ x$ – b'), equivalent to SWAP; STU $cc+1$.
- CF0Ccc — STIQ $cc+1$ ($x\ b$ – $x\ b\ f$ or $b'\ 0$).
- CF0Dcc — STUQ $cc+1$ ($x\ b$ – $x\ b\ f$ or $b'\ 0$).
- CF0Ecc — STIRQ $cc+1$ ($b\ x$ – $b\ x\ f$ or $b'\ 0$).
- CF0Fcc — STURQ $cc+1$ ($b\ x$ – $b\ x\ f$ or $b'\ 0$).
- CF10 — a longer version of STREF ($c\ b$ – b').
- CF11 — STBREF ($b'\ b$ – b''), equivalent to SWAP; STBREFREV.
- CF12 — a longer version of STSLICE ($s\ b$ – b').
- CF13 — STB ($b'\ b$ – b''), appends all data from *Builder* b' to *Builder* b.
- CF14 — STREFR ($b\ c$ – b').
- CF15 — STBREFR ($b\ b'$ – b''), a longer encoding of STBREFR.
- CF16 — STSLICER ($b\ s$ – b').
- CF17 — STBR ($b\ b'$ – b''), concatenates two *Builder*s, equivalent to SWAP; STB.
- CF18 — STREFQ ($c\ b$ – $c\ b\ -1$ or $b'\ 0$).
- CF19 — STBREFQ ($b'\ b$ – $b'\ b\ -1$ or $b''\ 0$).
- CF1A — STSLICEQ ($s\ b$ – $s\ b\ -1$ or $b'\ 0$).

- CF1B — STBQ (b' b – b' b -1 or b'' 0).
- CF1C — STREFRQ (b c – b c -1 or b' 0).
- CF1D — STBREFRQ (b b' – b b' -1 or b'' 0).
- CF1E — STSLICERQ (b s – b s -1 or b'' 0).
- CF1F — STBRQ (b b' – b b' -1 or b'' 0).
- CF20 — STREFCONST, equivalent to PUSHREF; STREFR.
- CF21 — STREF2CONST, equivalent to STREFCONST; STREFCONST.
- CF23 — ENDXC (b x – c), if $x \neq 0$, creates a *special* or *exotic* cell (cf. **3.1.2**) from *Builder* b. The type of the exotic cell must be stored in the first 8 bits of b. If $x = 0$, it is equivalent to ENDC. Otherwise some validity checks on the data and references of b are performed before creating the exotic cell.
- CF28 — STILE4 (x b – b'), stores a little-endian signed 32-bit integer.
- CF29 — STULE4 (x b – b'), stores a little-endian unsigned 32-bit integer.
- CF2A — STILE8 (x b – b'), stores a little-endian signed 64-bit integer.
- CF2B — STULE8 (x b – b'), stores a little-endian unsigned 64-bit integer.
- CF31 — BBITS (b – x), returns the number of data bits already stored in *Builder* b.
- CF32 — BREFS (b – y), returns the number of cell references already stored in b.
- CF33 — BBITREFS (b – x y), returns the numbers of both data bits and cell references in b.
- CF35 — BREMBITS (b – x'), returns the number of data bits that can still be stored in b.
- CF36 — BREMREFS (b – y').
- CF37 — BREMBITREFS (b – x' y').

- CF38*cc* — BCHKBITS $cc+1$ (b –), checks whether $cc+1$ bits can be stored into b, where $0 \leq cc \leq 255$.
- CF39 — BCHKBITS (b x –), checks whether x bits can be stored into b, $0 \leq x \leq 1023$. If there is no space for x more bits in b, or if x is not within the range $0 \ldots 1023$, throws an exception.
- CF3A — BCHKREFS (b y –), checks whether y references can be stored into b, $0 \leq y \leq 7$.
- CF3B — BCHKBITREFS (b x y –), checks whether x bits and y references can be stored into b, $0 \leq x \leq 1023$, $0 \leq y \leq 7$.
- CF3C*cc* — BCHKBITSQ $cc+1$ (b – ?), checks whether $cc+1$ bits can be stored into b, where $0 \leq cc \leq 255$.
- CF3D — BCHKBITSQ (b x – ?), checks whether x bits can be stored into b, $0 \leq x \leq 1023$.
- CF3E — BCHKREFSQ (b y – ?), checks whether y references can be stored into b, $0 \leq y \leq 7$.
- CF3F — BCHKBITREFSQ (b x y – ?), checks whether x bits and y references can be stored into b, $0 \leq x \leq 1023$, $0 \leq y \leq 7$.
- CF40 — STZEROES (b n – b'), stores n binary zeroes into *Builder* b.
- CF41 — STONES (b n – b'), stores n binary ones into *Builder* b.
- CF42 — STSAME (b n x – b'), stores n binary $x$es ($0 \leq x \leq 1$) into *Builder* b.
- CFC0_*xysss* — STSLICECONST *sss* (b – b'), stores a constant subslice *sss* consisting of $0 \leq x \leq 3$ references and up to $8y+1$ data bits, with $0 \leq y \leq 7$. Completion bit is assumed.
- CF81 — STSLICECONST ‘0’ (b – b'), stores one binary zero.
- CF83 — STSLICECONST ‘1’ (b – b'), stores one binary one.
- CFA2 — equivalent to STREFCONST.
- CFA3 — almost equivalent to STSLICECONST ‘1’; STREFCONST.

- `CFC2` — equivalent to `STREF2CONST`.
- `CFE2` — `STREF3CONST`.

A.6.2. Cell deserialization primitives.

- `D0` — `CTOS` (c – s), converts a *Cell* into a *Slice*. Notice that c must be either an ordinary cell, or an exotic cell (cf. **3.1.2**) which is automatically *loaded* to yield an ordinary cell c', converted into a *Slice* afterwards.
- `D1` — `ENDS` (s –), removes a *Slice* s from the stack, and throws an exception if it is not empty.
- `D2`cc — `LDI` $cc+1$ (s – x s'), loads (i.e., parses) a signed $cc+1$-bit integer x from *Slice* s, and returns the remainder of s as s'.
- `D3`cc — `LDU` $cc+1$ (s – x s'), loads an unsigned $cc+1$-bit integer x from *Slice* s.
- `D4` — `LDREF` (s – c s'), loads a cell reference c from s.
- `D5` — `LDREFRTOS` (s – s' s''), equivalent to `LDREF`; `SWAP`; `CTOS`.
- `D6`cc — `LDSLICE` $cc+1$ (s – s'' s'), cuts the next $cc+1$ bits of s into a separate *Slice* s''.
- `D700` — `LDIX` (s l – x s'), loads a signed l-bit ($0 \leq l \leq 257$) integer x from *Slice* s, and returns the remainder of s as s'.
- `D701` — `LDUX` (s l – x s'), loads an unsigned l-bit integer x from (the first l bits of) s, with $0 \leq l \leq 256$.
- `D702` — `PLDIX` (s l – x), preloads a signed l-bit integer from *Slice* s, for $0 \leq l \leq 257$.
- `D703` — `PLDUX` (s l – x), preloads an unsigned l-bit integer from s, for $0 \leq l \leq 256$.
- `D704` — `LDIXQ` (s l – x s' -1 or s 0), quiet version of `LDIX`: loads a signed l-bit integer from s similarly to `LDIX`, but returns a success flag, equal to -1 on success or to 0 on failure (if s does not have l bits), instead of throwing a cell underflow exception.

- D705 — LDUXQ ($s\ l$ – $x\ s'$ -1 or $s\ 0$), quiet version of LDUX.
- D706 — PLDIXQ ($s\ l$ – x -1 or 0), quiet version of PLDIX.
- D707 — PLDUXQ ($s\ l$ – x -1 or 0), quiet version of PLDUX.
- D708cc — LDI $cc+1$ (s – $x\ s'$), a longer encoding for LDI.
- D709cc — LDU $cc+1$ (s – $x\ s'$), a longer encoding for LDU.
- D70Acc — PLDI $cc+1$ (s – x), preloads a signed $cc+1$-bit integer from *Slice* s.
- D70Bcc — PLDU $cc+1$ (s – x), preloads an unsigned $cc+1$-bit integer from s.
- D70Ccc — LDIQ $cc+1$ (s – $x\ s'$ -1 or $s\ 0$), a quiet version of LDI.
- D70Dcc — LDUQ $cc+1$ (s – $x\ s'$ -1 or $s\ 0$), a quiet version of LDU.
- D70Ecc — PLDIQ $cc+1$ (s – x -1 or 0), a quiet version of PLDI.
- D70Fcc — PLDUQ $cc+1$ (s – x -1 or 0), a quiet version of PLDU.
- D714_c — PLDUZ $32(c+1)$ (s – $s\ x$), preloads the first $32(c+1)$ bits of *Slice* s into an unsigned integer x, for $0 \leq c \leq 7$. If s is shorter than necessary, missing bits are assumed to be zero. This operation is intended to be used along with IFBITJMP and similar instructions.
- D718 — LDSLICEX ($s\ l$ – $s''\ s'$), loads the first $0 \leq l \leq 1023$ bits from *Slice* s into a separate *Slice* s'', returning the remainder of s as s'.
- D719 — PLDSLICEX ($s\ l$ – s''), returns the first $0 \leq l \leq 1023$ bits of s as s''.
- D71A — LDSLICEXQ ($s\ l$ – $s''\ s'$ -1 or $s\ 0$), a quiet version of LDSLICEX.
- D71B — PLDSLICEXQ ($s\ l$ – s' -1 or 0), a quiet version of LDSLICEXQ.
- D71Ccc — LDSLICE $cc+1$ (s – $s''\ s'$), a longer encoding for LDSLICE.
- D71Dcc — PLDSLICE $cc+1$ (s – s''), returns the first $0 < cc+1 \leq 256$ bits of s as s''.

- D71Ecc — LDSLICEQ $cc+1$ (s – s'' s' -1 or s 0), a quiet version of LDSLICE.
- D71Fcc — PLDSLICEQ $cc+1$ (s – s'' -1 or 0), a quiet version of PLDSLICE.
- D720 — SDCUTFIRST (s l – s'), returns the first $0 \leq l \leq 1023$ bits of s. It is equivalent to PLDSLICEX.
- D721 — SDSKIPFIRST (s l – s'), returns all but the first $0 \leq l \leq 1023$ bits of s. It is equivalent to LDSLICEX; NIP.
- D722 — SDCUTLAST (s l – s'), returns the last $0 \leq l \leq 1023$ bits of s.
- D723 — SDSKIPLAST (s l – s'), returns all but the last $0 \leq l \leq 1023$ bits of s.
- D724 — SDSUBSTR (s l l' – s'), returns $0 \leq l' \leq 1023$ bits of s starting from offset $0 \leq l \leq 1023$, thus extracting a bit substring out of the data of s.
- D726 — SDBEGINSX (s s' – s''), checks whether s begins with (the data bits of) s', and removes s' from s on success. On failure throws a cell deserialization exception. Primitive SPFXREV can be considered a quiet version of SDBEGINSX.
- D727 — SDBEGINSXQ (s s' – s'' -1 or s 0), a quiet version of SDBEGINSX.
- D72A_$xsss$ — SDBEGINS (s – s''), checks whether s begins with constant bitstring sss of length $8x+3$ (with continuation bit assumed), where $0 \leq x \leq 127$, and removes sss from s on success.
- D72802 — SDBEGINS ‘0’ (s – s''), checks whether s begins with a binary zero.
- D72806 — SDBEGINS ‘1’ (s – s''), checks whether s begins with a binary one.
- D72E_$xsss$ — SDBEGINSQ (s – s'' -1 or s 0), a quiet version of SDBEGINS.
- D730 — SCUTFIRST (s l r – s'), returns the first $0 \leq l \leq 1023$ bits and first $0 \leq r \leq 4$ references of s.

- D731 — SSKIPFIRST ($s\ l\ r$ – s').
- D732 — SCUTLAST ($s\ l\ r$ – s'), returns the last $0 \leq l \leq 1023$ data bits and last $0 \leq r \leq 4$ references of s.
- D733 — SSKIPLAST ($s\ l\ r$ – s').
- D734 — SUBSLICE ($s\ l\ r\ l'\ r'$ – s'), returns $0 \leq l' \leq 1023$ bits and $0 \leq r' \leq 4$ references from *Slice* s, after skipping the first $0 \leq l \leq 1023$ bits and first $0 \leq r \leq 4$ references.
- D736 — SPLIT ($s\ l\ r$ – $s'\ s''$), splits the first $0 \leq l \leq 1023$ data bits and first $0 \leq r \leq 4$ references from s into s', returning the remainder of s as s''.
- D737 — SPLITQ ($s\ l\ r$ – $s'\ s''$ -1 or s 0), a quiet version of SPLIT.
- D739 — XCTOS (c – s ?), transforms an ordinary or exotic cell into a *Slice*, as if it were an ordinary cell. A flag is returned indicating whether c is exotic. If that be the case, its type can later be deserialized from the first eight bits of s.
- D73A — XLOAD (c – c'), loads an exotic cell c and returns an ordinary cell c'. If c is already ordinary, does nothing. If c cannot be loaded, throws an exception.
- D73B — XLOADQ (c – c' -1 or c 0), loads an exotic cell c as XLOAD, but returns 0 on failure.
- D741 — SCHKBITS ($s\ l$ –), checks whether there are at least l data bits in *Slice* s. If this is not the case, throws a cell deserialisation (i.e., cell underflow) exception.
- D742 — SCHKREFS ($s\ r$ –), checks whether there are at least r references in *Slice* s.
- D743 — SCHKBITREFS ($s\ l\ r$ –), checks whether there are at least l data bits and r references in *Slice* s.
- D745 — SCHKBITSQ ($s\ l$ – ?), checks whether there are at least l data bits in *Slice* s.

- D746 — SCHKREFSQ ($s\ r$ – ?), checks whether there are at least r references in *Slice* s.
- D747 — SCHKBITREFSQ ($s\ l\ r$ – ?), checks whether there are at least l data bits and r references in *Slice* s.
- D749 — SBITS (s – l), returns the number of data bits in *Slice* s.
- D74A — SREFS (s – r), returns the number of references in *Slice* s.
- D74B — SBITREFS (s – $l\ r$), returns both the number of data bits and the number of references in s.
- D750 — LDILE4 (s – $x\ s'$), loads a little-endian signed 32-bit integer.
- D751 — LDULE4 (s – $x\ s'$), loads a little-endian unsigned 32-bit integer.
- D752 — LDILE8 (s – $x\ s'$), loads a little-endian signed 64-bit integer.
- D753 — LDULE8 (s – $x\ s'$), loads a little-endian unsigned 64-bit integer.
- D754 — PLDILE4 (s – x), preloads a little-endian signed 32-bit integer.
- D755 — PLDULE4 (s – x), preloads a little-endian unsigned 32-bit integer.
- D756 — PLDILE8 (s – x), preloads a little-endian signed 64-bit integer.
- D757 — PLDULE8 (s – x), preloads a little-endian unsigned 64-bit integer.
- D758 — LDILE4Q (s – $x\ s'\ -1$ or $s\ 0$), quietly loads a little-endian signed 32-bit integer.
- D759 — LDULE4Q (s – $x\ s'\ -1$ or $s\ 0$), quietly loads a little-endian unsigned 32-bit integer.
- D75A — LDILE8Q (s – $x\ s'\ -1$ or $s\ 0$), quietly loads a little-endian signed 64-bit integer.
- D75B — LDULE8Q (s – $x\ s'\ -1$ or $s\ 0$), quietly loads a little-endian unsigned 64-bit integer.

- D75C — PLDILE4Q (s – x -1 or 0), quietly preloads a little-endian signed 32-bit integer.
- D75D — PLDULE4Q (s – x -1 or 0), quietly preloads a little-endian unsigned 32-bit integer.
- D75E — PLDILE8Q (s – x -1 or 0), quietly preloads a little-endian signed 64-bit integer.
- D75F — PLDULE8Q (s – x -1 or 0), quietly preloads a little-endian unsigned 64-bit integer.
- D760 — LDZEROES (s – n s'), returns the count n of leading zero bits in s, and removes these bits from s.
- D761 — LDONES (s – n s'), returns the count n of leading one bits in s, and removes these bits from s.
- D762 — LDSAME (s x – n s'), returns the count n of leading bits equal to $0 \leq x \leq 1$ in s, and removes these bits from s.

A.7 Continuation and control flow primitives

A.7.1. Unconditional control flow primitives.

- D8 — EXECUTE or CALLX (c –), *calls* or *executes* continuation c (i.e., cc $\leftarrow c \circ_0$ cc).
- D9 — JMPX (c –), *jumps*, or transfers control, to continuation c (i.e., cc $\leftarrow c \circ_0$ c0, or rather cc $\leftarrow (c \circ_0$ c0$) \circ_1$ c1). The remainder of the previous current continuation cc is discarded.
- DApr — CALLXARGS p,r (c –), *calls* continuation c with p parameters and expecting r return values, $0 \leq p \leq 15$, $0 \leq r \leq 15$.
- DB0p — CALLXARGS p,-1 (c –), *calls* continuation c with $0 \leq p \leq 15$ parameters, expecting an arbitrary number of return values.
- DB1p — JMPXARGS p (c –), *jumps* to continuation c, passing only the top $0 \leq p \leq 15$ values from the current stack to it (the remainder of the current stack is discarded).

- DB2r — RETARGS r, *returns* to c0, with $0 \leq r \leq 15$ return values taken from the current stack.
- DB30 — RET or RETTRUE, *returns* to the continuation at c0 (i.e., performs cc $\leftarrow$ c0). The remainder of the current continuation cc is discarded. Approximately equivalent to PUSH c0; JMPX.
- DB31 — RETALT or RETFALSE, *returns* to the continuation at c1 (i.e., cc $\leftarrow$ c1). Approximately equivalent to PUSH c1; JMPX.
- DB32 — BRANCH or RETBOOL (f –), performs RETTRUE if integer $f \neq 0$, or RETFALSE if $f = 0$.
- DB34 — CALLCC (c –), *call with current continuation*, transfers control to c, pushing the old value of cc into c's stack (instead of discarding it or writing it into new c0).
- DB35 — JMPXDATA (c –), similar to CALLCC, but the remainder of the current continuation (the old value of cc) is converted into a *Slice* before pushing it into the stack of c.
- DB36pr — CALLCCARGS p,r (c –), similar to CALLXARGS, but pushes the old value of cc (along with the top $0 \leq p \leq 15$ values from the original stack) into the stack of newly-invoked continuation c, setting cc.nargs to $-1 \leq r \leq 14$.
- DB38 — CALLXVARARGS (c p r –), similar to CALLXARGS, but takes $-1 \leq p, r \leq 255$ from the stack. The next three operations also take p and r from the stack.
- DB39 — RETVARARGS (p r –), similar to RETARGS.
- DB3A — JMPXVARARGS (c p r –), similar to JMPXARGS.
- DB3B — CALLCCVARARGS (c p r –), similar to CALLCCARGS.
- DB3C — CALLREF, equivalent to PUSHREFCONT; CALLX.
- DB3D — JMPREF, equivalent to PUSHREFCONT; JMPX.
- DB3E — JMPREFDATA, equivalent to PUSHREFCONT; JMPXDATA.

- DB3F — RETDATA, equivalent to PUSH c0; JMPXDATA. In this way, the remainder of the current continuation is converted into a *Slice* and returned to the caller.

A.7.2. Conditional control flow primitives.

- DC — IFRET (f –), performs a RET, but only if integer f is non-zero. If f is a NaN, throws an integer overflow exception.
- DD — IFNOTRET (f –), performs a RET, but only if integer f is zero.
- DE — IF (f c –), performs EXECUTE for c (i.e., *executes* c), but only if integer f is non-zero. Otherwise simply discards both values.
- DF — IFNOT (f c –), executes continuation c, but only if integer f is zero. Otherwise simply discards both values.
- E0 — IFJMP (f c –), jumps to c (similarly to JMPX), but only if f is non-zero.
- E1 — IFNOTJMP (f c –), jumps to c (similarly to JMPX), but only if f is zero.
- E2 — IFELSE (f c c' –), if integer f is non-zero, executes c, otherwise executes c'. Equivalent to CONDSELCHK; EXECUTE.
- E300 — IFREF (f –), equivalent to PUSHREFCONT; IF.
- E301 — IFNOTREF (f –), equivalent to PUSHREFCONT; IFNOT.
- E302 — IFJMPREF (f –), equivalent to PUSHREFCONT; IFJMP.
- E303 — IFNOTJMPREF (f –), equivalent to PUSHREFCONT; IFNOTJMP.
- E304 — CONDSEL (f x y – x or y), if integer f is non-zero, returns x, otherwise returns y. Notice that no type checks are performed on x and y; as such, it is more like a conditional stack operation. Roughly equivalent to ROT; ISZERO; INC; ROLL; NIP.
- E305 — CONDSELCHK (f x y – x or y), same as CONDSEL, but first checks whether x and y have the same type.
- E308 — IFRETALT (f –), performs RETALT if integer $f \neq 0$.

- E309 — IFNOTRETALT (f –), performs RETALT if integer $f = 0$.
- E39_n — IFBITJMP n (x c – x), checks whether bit $0 \leq n \leq 31$ is set in integer x, and if so, performs JMPX to continuation c. Value x is left in the stack.
- E3B_n — IFNBITJMP n (x c – x), jumps to c if bit $0 \leq n \leq 31$ is not set in integer x.
- E3D_n — IFBITJMPREF n (x – x), performs a JMPREF if bit $0 \leq n \leq 31$ is set in integer x.
- E3F_n — IFNBITJMPREF n (x – x), performs a JMPREF if bit $0 \leq n \leq 31$ is not set in integer x.

A.7.3. Control flow primitives: loops.

- E4 — REPEAT (n c –), executes continuation c n times, if integer n is non-negative. If $n \geq 2^{31}$ or $n < -2^{31}$, generates a range check exception. Notice that a RET inside the code of c works as a continue, not as a break. One should use either alternative (experimental) loops or alternative RETALT (along with a SETEXITALT before the loop) to break out of a loop.
- E5 — REPEATEND (n –), similar to REPEAT, but it is applied to the current continuation cc.
- E6 — UNTIL (c –), executes continuation c, then pops an integer x from the resulting stack. If x is zero, performs another iteration of this loop.
- E7 — UNTILEND (–), similar to UNTIL, but executes the current continuation cc in a loop. When the loop exit condition is satisfied, performs a RET.
- E8 — WHILE (c' c –), executes c' and pops an integer x from the resulting stack. If x is zero, exists the loop and transfers control to the original cc. If x is non-zero, executes c, and then begins a new iteration.
- E9 — WHILEEND (c' –), similar to WHILE, but uses the current continuation cc as the loop body.

- `EA` — `AGAIN` (c –), similar to `REPEAT`, but executes c infinitely many times. A `RET` only begins a new iteration of the infinite loop, which can be exited only by an exception, or a `RETALT` (or an explicit `JMPX`).

- `EB` — `AGAINEND` (–), similar to `AGAIN`, but performed with respect to the current continuation `cc`.

A.7.4. Manipulating the stack of continuations.

- `EC`rn — `SETCONTARGS` r`,`n ($x_1\ x_2 \ldots x_r\ c - c'$), similar to `SETCONTARGS` r, but sets c.`nargs` to the final size of the stack of c' plus n. In other words, transforms c into a *closure* or a *partially applied function*, with $0 \le n \le 14$ arguments missing.

- `EC0`n — `SETNUMARGS` n or `SETCONTARGS` 0`,`n ($c - c'$), sets c.`nargs` to n plus the current depth of c's stack, where $0 \le n \le 14$. If c.`nargs` is already set to a non-negative value, does nothing.

- `EC`r`F` — `SETCONTARGS` r or `SETCONTARGS` r`,`-1 ($x_1\ x_2 \ldots x_r\ c - c'$), pushes $0 \le r \le 15$ values $x_1 \ldots x_r$ into the stack of (a copy of) the continuation c, starting with x_1. If the final depth of c's stack turns out to be greater than c.`nargs`, a stack overflow exception is generated.

- `ED0`p — `RETURNARGS` p (–), leaves only the top $0 \le p \le 15$ values in the current stack (somewhat similarly to `ONLYTOPX`), with all the unused bottom values not discarded, but saved into continuation `c0` in the same way as `SETCONTARGS` does.

- `ED10` — `RETURNVARARGS` (p –), similar to `RETURNARGS`, but with Integer $0 \le p \le 255$ taken from the stack.

- `ED11` — `SETCONTVARARGS` ($x_1\ x_2 \ldots x_r\ c\ r\ n - c'$), similar to `SETCONTARGS`, but with $0 \le r \le 255$ and $-1 \le n \le 255$ taken from the stack.

- `ED12` — `SETNUMVARARGS` ($c\ n$ – c'), where $-1 \le n \le 255$. If $n = -1$, this operation does nothing ($c' = c$). Otherwise its action is similar to `SETNUMARGS` n, but with n taken from the stack.

A.7.5. Creating simple continuations and closures.

- `ED1E` — `BLESS` (s – c), transforms a *Slice* s into a simple ordinary continuation c, with c.`code` $= s$ and an empty stack and savelist.

- ED1F — BLESSVARARGS $(x_1 \dots x_r\ s\ r\ n - c)$, equivalent to ROT; BLESS; ROTREV; SETCONTVARARGS.

- EErn — BLESSARGS r, n $(x_1 \dots x_r\ s - c)$, where $0 \leq r \leq 15$, $-1 \leq n \leq 14$, equivalent to BLESS; SETCONTARGS r, n. The value of n is represented inside the instruction by the 4-bit integer n mod 16.

- EE0n — BLESSNUMARGS n or BLESSARGS 0,n $(s - c)$, also transforms a *Slice* s into a *Continuation* c, but sets c.nargs to $0 \leq n \leq 14$.

A.7.6. Operations with continuation savelists and control registers.

- ED4i — PUSH c(i) or PUSHCTR c(i) $(- x)$, pushes the current value of control register c(i). If the control register is not supported in the current codepage, or if it does not have a value, an exception is triggered.

- ED44 — PUSH c4 or PUSHROOT, pushes the "global data root" cell reference, thus enabling access to persistent smart-contract data.

- ED5i — POP c(i) or POPCTR c(i) $(x -)$, pops a value x from the stack and stores it into control register c(i), if supported in the current codepage. Notice that if a control register accepts only values of a specific type, a type-checking exception may occur.

- ED54 — POP c4 or POPROOT, sets the "global data root" cell reference, thus allowing modification of persistent smart-contract data.

- ED6i — SETCONT c(i) or SETCONTCTR c(i) $(x\ c - c')$, stores x into the savelist of continuation c as c(i), and returns the resulting continuation c'. Almost all operations with continuations may be expressed in terms of SETCONTCTR, POPCTR, and PUSHCTR.

- ED7i — SETRETCTR c(i) $(x -)$, equivalent to PUSH c0; SETCONTCTR c(i); POP c0.

- ED8i — SETALTCTR c(i) $(x -)$, equivalent to PUSH c1; SETCONTCTR c(i); POP c0.

- ED9i — POPSAVE c(i) or POPCTRSAVE c(i) $(x -)$, similar to POP c(i), but also saves the old value of c(i) into continuation c0. Equivalent (up to exceptions) to SAVECTR c(i); POP c(i).

- EDAi — SAVE c(i) or SAVECTR c(i) (–), saves the current value of c(i) into the savelist of continuation c0. If an entry for c(i) is already present in the savelist of c0, nothing is done. Equivalent to PUSH c(i); SETRETCTR $c(i)$.

- EDBi — SAVEALT c(i) or SAVEALTCTR c(i) (x –), similar to SAVE c(i), but saves the current value of c(i) into the savelist of c1, not c0.

- EDCi — SAVEBOTH c(i) or SAVEBOTHCTR c(i) (x –), equivalent to DUP; SAVE c(i); SAVEALT c(i).

- EDE0 — PUSHCTRX (i – x), similar to PUSHCTR c(i), but with i, $0 \le i \le 255$, taken from the stack. Notice that this primitive is one of the few "exotic" primitives, which are not polymorphic like stack manipulation primitives, and at the same time do not have well-defined types of parameters and return values, because the type of x depends on i.

- EDE1 — POPCTRX (x i –), similar to POPCTR c(i), but with $0 \le i \le 255$ from the stack.

- EDE2 — SETCONTCTRX (x c i – c'), similar to SETCONTCTR c(i), but with $0 \le i \le 255$ from the stack.

- EDF0 — COMPOS or BOOLAND (c c' – c''), computes the composition $c \circ_0 c'$, which has the meaning of "perform c, and, if successful, perform c'" (if c is a boolean circuit) or simply "perform c, then c'". Equivalent to SWAP; SETCONT c0.

- EDF1 — COMPOSALT or BOOLOR (c c' – c''), computes the alternative composition $c \circ_1 c'$, which has the meaning of "perform c, and, if not successful, perform c'" (if c is a boolean circuit). Equivalent to SWAP; SETCONT c1.

- EDF2 — COMPOSBOTH (c c' – c''), computes $(c \circ_0 c') \circ_1 c'$, which has the meaning of "compute boolean circuit c, then compute c', regardless of the result of c".

- EDF3 — ATEXIT (c –), sets c0 $\leftarrow c \circ_0$ c0. In other words, c will be executed before exiting current subroutine.

- EDF4 — ATEXITALT (c –), sets c1 $\leftarrow c \circ_1$ c1. In other words, c will be executed before exiting current subroutine by its alternative return path.
- EDF5 — SETEXITALT (c –), sets c1 $\leftarrow (c \circ_0$ c0$) \circ_1$ c1. In this way, a subsequent RETALT will first execute c, then transfer control to the original c0. This can be used, for instance, to exit from nested loops.
- EDF6 — THENRET (c – c'), computes $c' := c \circ_0$ c0
- EDF7 — THENRETALT (c – c'), computes $c' := c \circ_0$ c1
- EDF8 — INVERT (–), interchanges c0 and c1.
- EDF9 — BOOLEVAL (c – ?), performs cc $\leftarrow \big(c \circ_0 ($(PUSH $-1) \circ_0$ cc$)\big) \circ_1$ ((PUSH 0) $\circ_0$ cc). If c represents a boolean circuit, the net effect is to evaluate it and push either -1 or 0 into the stack before continuing.
- EErn — BLESSARGS r, n ($x_1 \ldots x_r$ s – c), described in **A.7.4**.

A.7.7. Dictionary subroutine calls and jumps.

- F0n — CALL n or CALLDICT n (– n), calls the continuation in c3, pushing integer $0 \le n \le 255$ into its stack as an argument. Approximately equivalent to PUSHINT n; PUSH c3; EXECUTE.
- F12_n — CALL n for $0 \le n < 2^{14}$ (– n), an encoding of CALL n for larger values of n.
- F16_n — JMP n or JMPDICT n (– n), jumps to the continuation in c3, pushing integer $0 \le n < 2^{14}$ as its argument. Approximately equivalent to PUSHINT n; PUSH c3; JMPX.
- F1A_n — PREPARE n (– n c), equivalent to PUSHINT n; PUSH c3, for $0 \le n < 2^{14}$. In this way, CALL n is approximately equivalent to PREPARE n; EXECUTE, and JMP n is approximately equivalent to PREPARE n; JMPX. One might use, for instance, CALLARGS or CALLCC instead of EXECUTE here.

A.8 Exception generating and handling primitives

A.8.1. Throwing exceptions.

- F22_*nn* — THROW *nn* (– 0 *nn*), throws exception $0 \leq nn \leq 63$ with parameter zero. In other words, it transfers control to the continuation in c2, pushing 0 and *nn* into its stack, and discarding the old stack altogether.
- F26_*nn* — THROWIF *nn* (*f* –), throws exception $0 \leq nn \leq 63$ with parameter zero only if integer $f \neq 0$.
- F2A_*nn* — THROWIFNOT *nn* (*f* –), throws exception $0 \leq nn \leq 63$ with parameter zero only if integer $f = 0$.
- F2C4_*nn* — THROW *nn* for $0 \leq nn < 2^{11}$, an encoding of THROW *nn* for larger values of *nn*.
- F2CC_*nn* — THROWARG *nn* (*x* – *x nn*), throws exception $0 \leq nn < 2^{11}$ with parameter *x*, by copying *x* and *nn* into the stack of c2 and transferring control to c2.
- F2D4_*nn* — THROWIF *nn* (*f* –) for $0 \leq nn < 2^{11}$.
- F2DC_*nn* — THROWARGIF *nn* (*x f* –), throws exception $0 \leq nn < 2^{11}$ with parameter *x* only if integer $f \neq 0$.
- F2E4_*nn* — THROWIFNOT *nn* (*f* –) for $0 \leq nn < 2^{11}$.
- F2EC_*nn* — THROWARGIFNOT *nn* (*x f* –), throws exception $0 \leq nn < 2^{11}$ with parameter *x* only if integer $f = 0$.
- F2F0 — THROWANY (*n* – 0 *n*), throws exception $0 \leq n < 2^{16}$ with parameter zero. Approximately equivalent to PUSHINT 0; SWAP; THROWARGANY.
- F2F1 — THROWARGANY (*x n* – *x n*), throws exception $0 \leq n < 2^{16}$ with parameter *x*, transferring control to the continuation in c2. Approximately equivalent to PUSH c2; JMPXARGS 2.
- F2F2 — THROWANYIF (*n f* –), throws exception $0 \leq n < 2^{16}$ with parameter zero only if $f \neq 0$.

- F2F3 — THROWARGANYIF (x n f –), throws exception $0 \leq n < 2^{16}$ with parameter x only if $f \neq 0$.
- F2F4 — THROWANYIFNOT (n f –), throws exception $0 \leq n < 2^{16}$ with parameter zero only if $f = 0$.
- F2F5 — THROWARGANYIFNOT (x n f –), throws exception $0 \leq n < 2^{16}$ with parameter x only if $f = 0$.

A.8.2. Catching and handling exceptions.

- F2FF — TRY (c c' –), sets c2 to c', first saving the old value of c2 both into the savelist of c' and into the savelist of the current continuation, which is stored into c.c0 and c'.c0. Then runs c similarly to EXECUTE. If c does not throw any exceptions, the original value of c2 is automatically restored on return from c. If an exception occurs, the execution is transferred to c', but the original value of c2 is restored in the process, so that c' can re-throw the exception by THROWANY if it cannot handle it by itself.
- F3*pr* — TRYARGS p,r (c c' –), similar to TRY, but with CALLARGS p,r internally used instead of EXECUTE. In this way, all but the top $0 \leq p \leq 15$ stack elements will be saved into current continuation's stack, and then restored upon return from either c or c', with the top $0 \leq r \leq 15$ values of the resulting stack of c or c' copied as return values.

A.9 Dictionary manipulation primitives

TVM's dictionary support is discussed at length in **3.3**. The basic operations with dictionaries are listed in **3.3.10**, while the taxonomy of dictionary manipulation primitives is provided in **3.3.11**. Here we use the concepts and notation introduced in those sections.

A.9.1. Dictionary creation.

- 8B04 — NEWDICT (– s), returns a *Slice* representing a new empty dictionary. It is an alternative mnemonics for PUSHSLICE x4.
- F400 — DICTEMPTY (s – ?), returns a flag indicating whether a dictionary is empty. Equivalent to PLDU 1; DEC.

A.9.2. Dictionary serialization and deserialization.

- `CE` — `STDICT` ($s\ b$ – b'), stores dictionary s into *Builder* b, returing the resulting *Builder* b'. It is actually a synonym for `STSLICE`.
- `F401` — `SKIPDICT` or `SKIPOPTREF` (s – s'), equivalent to `LDDICT`; `NIP`.
- `F402` — `LDOPTREF` (s – s' c -1 or s' 0), performs `LDI 1`, and then a `LDREF` if the previous operation returns -1. May be applied to dictionaries or to values of arbitrary $(\hat{}Y)^?$ types.
- `F403` — `PLDOPTREF` (s – c -1 or 0), similar to `LDOPTREF`, but does not return the remainder of *Slice* s.
- `F404` — `LDDICT` (s – s' s''), loads (parses) a dictionary s' from *Slice* s, and returns the remainder of s as s''. This is a "split function" for all *HashmapE*(n, X) dictionary types.
- `F405` — `PLDDICT` (s – s'), preloads a dictionary s' from *Slice* s. Approximately equivalent to `LDDICT`; `DROP`.
- `F406` — `LDDICTQ` (s – s' s'' -1 or s 0), a quiet version of `LDDICT`.
- `F407` — `PLDDICTQ` (s – s' -1 or 0), a quiet version of `PLDDICT`.

A.9.3. Get dictionary operations.

- `F40A` — `DICTGET` (k s n – x -1 or 0), looks up key k (represented by a *Slice*, the first $0 \leq n \leq 1023$ data bits of which are used as a key) in dictionary s of type *HashmapE*(n, X) with n-bit keys. On success, returns the value found as a *Slice* x.
- `F40B` — `DICTGETREF` (k s n – c -1 or 0), similar to `DICTGET`, but with a `LDREF`; `ENDS` applied to x on success. This operation is useful for dictionaries of type *HashmapE*$(n, \hat{}Y)$.
- `F40C` — `DICTIGET` (i s n – x -1 or 0), similar to `DICTGET`, but with a signed (big-endian) n-bit *Integer* i as a key. If i does not fit into n bits, returns 0. If i is a `NaN`, throws an integer overflow exception.
- `F40D` — `DICTIGETREF` (i s n – c -1 or 0), combines `DICTIGET` with `DICTGETREF`: it uses signed n-bit *Integer* i as a key and returns a *Cell* instead of a *Slice* on success.

- F40E — DICTUGET (i s n – x -1 or 0), similar to DICTIGET, but with *unsigned* (big-endian) n-bit *Integer* i used as a key.
- F40F — DICTUGETREF (i s n – c -1 or 0), similar to DICTIGETREF, but with an unsigned n-bit *Integer* key i.

A.9.4. Set/Replace/Add **dictionary operations.** The mnemonics of the following dictionary primitives are constructed in a systematic fashion according to the regular expression DICT[, I, U](SET, REPLACE, ADD)[GET][REF] depending on the type of the key used (a *Slice* or a signed or unsigned *Integer*), the dictionary operation to be performed, and the way the values are accepted and returned (as *Cell*s or as *Slice*s). Therefore, we provide a detailed description only for some primitives, assuming that this information is sufficient for the reader to understand the precise action of the remaining primitives.

- F412 — DICTSET (x k s n – s'), sets the value associated with n-bit key k (represented by a *Slice* as in DICTGET) in dictionary s (also represented by a *Slice*) to value x (again a *Slice*), and returns the resulting dictionary as s'.
- F413 — DICTSETREF (c k s n – s'), similar to DICTSET, but with the value set to a reference to *Cell* c.
- F414 — DICTISET (x i s n – s'), similar to DICTSET, but with the key represented by a (big-endian) signed n-bit integer i. If i does not fit into n bits, a range check exception is generated.
- F415 — DICTISETREF (c i s n – s'), similar to DICTSETREF, but with the key a signed n-bit integer as in DICTISET.
- F416 — DICTUSET (x i s n – s'), similar to DICTISET, but with i an *unsigned* n-bit integer.
- F417 — DICTUSETREF (c i s n – s'), similar to DICTISETREF, but with i unsigned.
- F41A — DICTSETGET (x k s n – s' y -1 or s' 0), combines DICTSET with DICTGET: it sets the value corresponding to key k to x, but also returns the old value y associated with the key in question, if present.

- F41B — DICTSETGETREF ($c\ k\ s\ n - s'\ c'\ -1$ or $s'\ 0$), combines DICTSETREF with DICTGETREF similarly to DICTSETGET.
- F41C — DICTISETGET ($x\ i\ s\ n - s'\ y\ -1$ or $s'\ 0$), similar to DICTSETGET, but with the key represented by a big-endian signed n-bit *Integer* i.
- F41D — DICTISETGETREF ($c\ i\ s\ n - s'\ c'\ -1$ or $s'\ 0$), a version of DICTSETGETREF with signed *Integer* i as a key.
- F41E — DICTUSETGET ($x\ i\ s\ n - s'\ y\ -1$ or $s'\ 0$), similar to DICTISETGET, but with i an unsigned n-bit integer.
- F41F — DICTUSETGETREF ($c\ i\ s\ n - s'\ c'\ -1$ or $s'\ 0$).
- F422 — DICTREPLACE ($x\ k\ s\ n - s'\ -1$ or $s\ 0$), a Replace operation, which is similar to DICTSET, but sets the value of key k in dictionary s to x only if the key k was already present in s.
- F423 — DICTREPLACEREF ($c\ k\ s\ n - s'\ -1$ or $s\ 0$), a Replace counterpart of DICTSETREF.
- F424 — DICTIREPLACE ($x\ i\ s\ n - s'\ -1$ or $s\ 0$), a version of DICTREPLACE with signed n-bit *Integer* i used as a key.
- F425 — DICTIREPLACEREF ($c\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F426 — DICTUREPLACE ($x\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F427 — DICTUREPLACEREF ($c\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F42A — DICTREPLACEGET ($x\ k\ s\ n - s'\ y\ -1$ or $s\ 0$), a Replace counterpart of DICTSETGET: on success, also returns the old value associated with the key in question.
- F42B — DICTREPLACEGETREF ($c\ k\ s\ n - s'\ c'\ -1$ or $s\ 0$).
- F42C — DICTIREPLACEGET ($x\ i\ s\ n - s'\ y\ -1$ or $s\ 0$).
- F42D — DICTIREPLACEGETREF ($c\ i\ s\ n - s'\ c'\ -1$ or $s\ 0$).
- F42E — DICTUREPLACEGET ($x\ i\ s\ n - s'\ y\ -1$ or $s\ 0$).
- F42F — DICTUREPLACEGETREF ($c\ i\ s\ n - s'\ c'\ -1$ or $s\ 0$).

- F432 — DICTADD ($x\ k\ s\ n - s'\ -1$ or $s\ 0$), an Add counterpart of DICTSET: sets the value associated with key k in dictionary s to x, but only if it is not already present in s.
- F433 — DICTADDREF ($c\ k\ s\ n - s'\ -1$ or $s\ 0$).
- F434 — DICTIADD ($x\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F435 — DICTIADDREF ($c\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F436 — DICTUADD ($x\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F437 — DICTUADDREF ($c\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F43A — DICTADDGET ($x\ k\ s\ n - s'\ -1$ or $s\ y\ 0$), an Add counterpart of DICTSETGET: sets the value associated with key k in dictionary s to x, but only if key k is not already present in s. Otherwise, just returns the old value y without changing the dictionary.
- F43B — DICTADDGETREF ($c\ k\ s\ n - s'\ -1$ or $s\ c'\ 0$), an Add counterpart of DICTSETGETREF.
- F43C — DICTIADDGET ($x\ i\ s\ n - s'\ -1$ or $s\ y\ 0$).
- F43D — DICTIADDGETREF ($c\ i\ s\ n - s'\ -1$ or $s\ c'\ 0$).
- F43E — DICTUADDGET ($x\ i\ s\ n - s'\ -1$ or $s\ y\ 0$).
- F43F — DICTUADDGETREF ($c\ i\ s\ n - s'\ -1$ or $s\ c'\ 0$).

A.9.5. Builder-accepting variants of Set dictionary operations. The following primitives accept the new value as a *Builder* b instead of a *Slice* x, which often is more convenient if the value needs to be serialized from several components computed in the stack. (This is reflected by appending a B to the mnemonics of the corresponding Set primitives that work with *Slice*s.) The net effect is roughly equivalent to converting b into a *Slice* by ENDC; CTOS and executing the corresponding primitive listed in **A.9.4**.

- F441 — DICTSETB ($b\ k\ s\ n - s'$).
- F442 — DICTISETB ($b\ i\ s\ n - s'$).
- F443 — DICTUSETB ($b\ i\ s\ n - s'$).

- F445 — DICTSETGETB ($b\ k\ s\ n - s'\ y\ -1$ or $s'\ 0$).
- F446 — DICTISETGETB ($b\ i\ s\ n - s'\ y\ -1$ or $s'\ 0$).
- F447 — DICTUSETGETB ($b\ i\ s\ n - s'\ y\ -1$ or $s'\ 0$).
- F449 — DICTREPLACEB ($b\ k\ s\ n - s'\ -1$ or $s\ 0$).
- F44A — DICTIREPLACEB ($b\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F44B — DICTUREPLACEB ($b\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F44D — DICTREPLACEGETB ($b\ k\ s\ n - s'\ y\ -1$ or $s\ 0$).
- F44E — DICTIREPLACEGETB ($b\ i\ s\ n - s'\ y\ -1$ or $s\ 0$).
- F44F — DICTUREPLACEGETB ($b\ i\ s\ n - s'\ y\ -1$ or $s\ 0$).
- F451 — DICTADDB ($b\ k\ s\ n - s'\ -1$ or $s\ 0$).
- F452 — DICTIADDB ($b\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F453 — DICTUADDB ($b\ i\ s\ n - s'\ -1$ or $s\ 0$).
- F455 — DICTADDGETB ($b\ k\ s\ n - s'\ -1$ or $s\ y\ 0$).
- F456 — DICTIADDGETB ($b\ i\ s\ n - s'\ -1$ or $s\ y\ 0$).
- F457 — DICTUADDGETB ($b\ i\ s\ n - s'\ -1$ or $s\ y\ 0$).

A.9.6. Delete dictionary operations.

- F459 — DICTDEL ($k\ s\ n - s'\ -1$ or $s\ 0$), deletes n-bit key, represented by a *Slice* k, from dictionary s. If the key is present, returns the modified dictionary s' and the success flag -1. Otherwise, returns the original dictionary s and 0.
- F45A — DICTIDEL ($i\ s\ n - s'\ ?$), a version of DICTDEL with the key represented by a signed n-bit *Integer* i. If i does not fit into n bits, simply returns $s\ 0$ ("key not found, dictionary unmodified").
- F45B — DICTUDEL ($i\ s\ n - s'\ ?$), similar to DICTIDEL, but with i an unsigned n-bit integer.

- F462 — DICTDELGET (k s n – s' x -1 or s 0), deletes n-bit key, represented by a *Slice* k, from dictionary s. If the key is present, returns the modified dictionary s', the original value x associated with the key k (represented by a *Slice*), and the success flag -1. Otherwise, returns the original dictionary s and 0.
- F463 — DICTDELGETREF (k s n – s' c -1 or s 0), similar to DICTDELGET, but with LDREF; ENDS applied to x on success, so that the value returned c is a *Cell*.
- F464 — DICTIDELGET (i s n – s' x -1 or s 0), a variant of primitive DICTDELGET with signed n-bit integer i as a key.
- F465 — DICTIDELGETREF (i s n – s' c -1 or s 0), a variant of primitive DICTIDELGET returning a *Cell* instead of a *Slice*.
- F466 — DICTUDELGET (i s n – s' x -1 or s 0), a variant of primitive DICTDELGET with unsigned n-bit integer i as a key.
- F467 — DICTUDELGETREF (i s n – s' c -1 or s 0), a variant of primitive DICTUDELGET returning a *Cell* instead of a *Slice*.

A.9.7. Prefix code dictionary operations. These are some basic operations for constructing prefix code dictionaries (cf. **3.4.2**). The primary application for prefix code dictionaries is deserializing TL-B serialized data structures, or, more generally, parsing prefix codes. Therefore, most prefix code dictionaries will be constant and created at compile time, not by the following primitives.

Some Get operations for prefix code dictionaries may be found in **A.9.10**. Other prefix code dictionary operations include:

- F470 — PFXDICTSET (x k s n – s' -1 or s 0).
- F471 — PFXDICTREPLACE (x k s n – s' -1 or s 0).
- F472 — PFXDICTADD (x k s n – s' -1 or s 0).
- F473 — PFXDICTDEL (k s n – s' -1 or s 0).

These primitives are completely similar to their non-prefix code counterparts DICTSET etc (cf. **A.9.4**), with the obvious difference that even a Set may fail in a prefix code dictionary, so a success flag must be returned by PFXDICTSET as well.

A.9.8. Variants of GetNext and GetPrev operations.

- F474 — DICTGETNEXT (k s n – x' k' -1 or 0), computes the minimal key k' in dictionary s that is lexicographically greater than k, and returns k' (represented by a *Slice*) along with associated value x' (also represented by a *Slice*).
- F475 — DICTGETNEXTEQ (k s n – x' k' -1 or 0), similar to DICTGETNEXT, but computes the minimal key k' that is lexicographically greater than or equal to k.
- F476 — DICTGETPREV (k s n – x' k' -1 or 0), similar to DICTGETNEXT, but computes the maximal key k' lexicographically smaller than k.
- F477 — DICTGETPREVEQ (k s n – x' k' -1 or 0), similar to DICTGETPREV, but computes the maximal key k' lexicographically smaller than or equal to k.
- F478 — DICTIGETNEXT (i s n – x' i' -1 or 0), similar to DICTGETNEXT, but interprets all keys in dictionary s as big-endian signed n-bit integers, and computes the minimal key i' that is larger than *Integer* i (which does not necessarily fit into n bits).
- F479 — DICTIGETNEXTEQ (i s n – x' i' -1 or 0).
- F47A — DICTIGETPREV (i s n – x' i' -1 or 0).
- F47B — DICTIGETPREVEQ (i s n – x' i' -1 or 0).
- F47C — DICTUGETNEXT (i s n – x' i' -1 or 0), similar to DICTGETNEXT, but interprets all keys in dictionary s as big-endian unsigned n-bit integers, and computes the minimal key i' that is larger than *Integer* i (which does not necessarily fit into n bits, and is not necessarily non-negative).
- F47D — DICTUGETNEXTEQ (i s n – x' i' -1 or 0).
- F47E — DICTUGETPREV (i s n – x' i' -1 or 0).
- F47F — DICTUGETPREVEQ (i s n – x' i' -1 or 0).

A.9.9. GetMin, GetMax, RemoveMin, RemoveMax operations.

- F482 — DICTMIN ($s\ n$ – $x\ k$ -1 or 0), computes the minimal key k (represented by a *Slice* with n data bits) in dictionary s, and returns k along with the associated value x.
- F483 — DICTMINREF ($s\ n$ – $c\ k$ -1 or 0), similar to DICTMIN, but returns the only reference in the value as a *Cell* c.
- F484 — DICTIMIN ($s\ n$ – $x\ i$ -1 or 0), somewhat similar to DICTMIN, but computes the minimal key i under the assumption that all keys are big-endian signed n-bit integers. Notice that the key and value returned may differ from those computed by DICTMIN and DICTUMIN.
- F485 — DICTIMINREF ($s\ n$ – $c\ i$ -1 or 0).
- F486 — DICTUMIN ($s\ n$ – $x\ i$ -1 or 0), similar to DICTMIN, but returns the key as an unsigned n-bit *Integer* i.
- F487 — DICTUMINREF ($s\ n$ – $c\ i$ -1 or 0).
- F48A — DICTMAX ($s\ n$ – $x\ k$ -1 or 0), computes the maximal key k (represented by a *Slice* with n data bits) in dictionary s, and returns k along with the associated value x.
- F48B — DICTMAXREF ($s\ n$ – $c\ k$ -1 or 0).
- F48C — DICTIMAX ($s\ n$ – $x\ i$ -1 or 0).
- F48D — DICTIMAXREF ($s\ n$ – $c\ i$ -1 or 0).
- F48E — DICTUMAX ($s\ n$ – $x\ i$ -1 or 0).
- F48F — DICTUMAXREF ($s\ n$ – $c\ i$ -1 or 0).
- F492 — DICTREMMIN ($s\ n$ – $s'\ x\ k$ -1 or s 0), computes the minimal key k (represented by a *Slice* with n data bits) in dictionary s, removes k from the dictionary, and returns k along with the associated value x and the modified dictionary s'.
- F493 — DICTREMMINREF ($s\ n$ – $s'\ c\ k$ -1 or s 0), similar to DICTREMMIN, but returns the only reference in the value as a *Cell* c.

- F494 — DICTIREMMIN ($s\ n$ – $s'\ x\ i\ -1$ or $s\ 0$), somewhat similar to DICTREMMIN, but computes the minimal key i under the assumption that all keys are big-endian signed n-bit integers. Notice that the key and value returned may differ from those computed by DICTREMMIN and DICTUREMMIN.
- F495 — DICTIREMMINREF ($s\ n$ – $s'\ c\ i\ -1$ or $s\ 0$).
- F496 — DICTUREMMIN ($s\ n$ – $s'\ x\ i\ -1$ or $s\ 0$), similar to DICTREMMIN, but returns the key as an unsigned n-bit *Integer* i.
- F497 — DICTUREMMINREF ($s\ n$ – $s'\ c\ i\ -1$ or $s\ 0$).
- F49A — DICTREMMAX ($s\ n$ – $s'\ x\ k\ -1$ or $s\ 0$), computes the maximal key k (represented by a *Slice* with n data bits) in dictionary s, removes k from the dictionary, and returns k along with the associated value x and the modified dictionary s'.
- F49B — DICTREMMAXREF ($s\ n$ – $s'\ c\ k\ -1$ or $s\ 0$).
- F49C — DICTIREMMAX ($s\ n$ – $s'\ x\ i\ -1$ or $s\ 0$).
- F49D — DICTIREMMAXREF ($s\ n$ – $s'\ c\ i\ -1$ or $s\ 0$).
- F49E — DICTUREMMAX ($s\ n$ – $s'\ x\ i\ -1$ or $s\ 0$).
- F49F — DICTUREMMAXREF ($s\ n$ – $s'\ c\ i\ -1$ or $s\ 0$).

A.9.10. Special Get dictionary and prefix code dictionary operations, and constant dictionaries.

- F4A0 — DICTIGETJMP ($i\ s\ n$ –), similar to DICTIGET (cf. **A.9.3**), but with s' BLESSed into a continuation with a subsequent JMPX to it on success. On failure, does nothing. This is useful for implementing switch/case constructions.
- F4A1 — DICTUGETJMP ($i\ s\ n$ –), similar to DICTIGETJMP, but performs DICTUGET instead of DICTIGET.
- F4A2 — DICTIGETEXEC ($i\ s\ n$ –), similar to DICTIGETJMP, but with EXECUTE instead of JMPX.

- F4A3 — DICTUGETEXEC (i s n –), similar to DICTUGETJMP, but with EXECUTE instead of JMPX.

- F4A6_n — DICTPUSHCONST n (– s n), pushes a non-empty constant dictionary s (as a *Slice*) along with its key length $0 \leq n \leq 1023$, stored as a part of the instruction. The dictionary itself is created from the first of remaining references of the current continuation. In this way, the complete DICTPUSHCONST instruction can be obtained by first serializing xF4A8_, then the non-empty dictionary itself (one 1 bit and a cell reference), and then the unsigned 10-bit integer n (as if by a STU 10 instruction). An empty dictionary can be pushed by a NEWDICT primitive (cf. **A.9.1**) instead.

- F4A8 — PFXDICTGETQ (s' s n – s'' x s''' -1 or s' 0), looks up the unique prefix of *Slice* s' present in the prefix code dictionary (cf. **3.4.2**) represented by *Slice* s and $0 \leq n \leq 1023$. If found, the prefix of s' is returned as s'', and the corresponding value (also a *Slice*) as x. The remainder of s' is returned as a *Slice* s'''. If no prefix of s' is a key in prefix code dictionary s, returns the unchanged s' and a zero failure flag.

- F4A9 — PFXDICTGET (s' s n – s'' x s'''), similar to PFXDICTGET, but throws a cell deserialization failure exception on failure.

- F4AA — PFXDICTGETJMP (s' s n – s'' s''' or s'), similar to PFXDICTGETQ, but on success BLESSes the value x into a *Continuation* and transfers control to it as if by a JMPX. On failure, returns s' unchanged and continues execution.

- F4AB — PFXDICTGETEXEC (s' s n – s'' s'''), similar to PFXDICTGETJMP, but EXECutes the continuation found instead of jumping to it. On failure, throws a cell deserialization exception.

- F4AE_n — PFXDICTCONSTGETJMP n or PFXDICTSWITCH n (s' – s'' s''' or s'), combines DICTPUSHCONST n for $0 \leq n \leq 1023$ with PFXDICTGETJMP.

B Formal properties and specifications of TVM

This appendix discusses certain formal properties of TVM that are necessary for executing smart contracts in the TON Blockchain and validating such executions afterwards.

B.1 Serialization of the TVM state

Recall that a virtual machine used for executing smart contracts in a blockchain must be *deterministic*, otherwise the validation of each execution would require the inclusion of all intermediate steps of the execution into a block, or at least of the choices made when indeterministic operations have been performed.

Furthermore, the *state* of such a virtual machine must be (uniquely) serializable, so that even if the state itself is not usually included in a block, its *hash* is still well-defined and can be included into a block for verification purposes.

B.1.1. TVM stack values. TVM stack values can be serialized as follows:

```
vm_stk_tinyint#01 value:int64 = VmStackValue;
vm_stk_int#0201_ value:int257 = VmStackValue;
vm_stk_nan#02FF = VmStackValue;
vm_stk_cell#03 cell:^Cell = VmStackValue;
_ cell:^Cell st_bits:(## 10) end_bits:(## 10)
  { st_bits <= end_bits }
  st_ref:(#<= 4) end_ref:(#<= 4)
  { st_ref <= end_ref } = VmCellSlice;
vm_stk_slice#04 _:VmCellSlice  = VmStackValue;
vm_stk_builder#05 cell:^Cell = VmStackValue;
vm_stk_cont#06 cont:VmCont = VmStackValue;
```

Of these, `vm_stk_tinyint` is never used by TVM in codepage zero; it is used only in restricted modes.

B.1.2. TVM stack. The TVM stack can be serialized as follows:

```
vm_stack#_ depth:(## 24) stack:(VmStackList depth) = VmStack;
vm_stk_cons#_ {n:#} head:VmStackValue tail:^(VmStackList n)
  = VmStackList (n + 1);
vm_stk_nil#_ = VmStackList 0;
```

B.1.3. TVM control registers. Control registers in TVM can be serialized as follows:

```
_ cregs:(HashmapE 4 VmStackValue) = VmSaveList;
```

B.1.4. TVM gas limits. Gas limits in TVM can be serialized as follows:

```
gas_limits#_ remaining:int64 _:^[
  max_limit:int64 cur_limit:int64 credit:int64 ]
  = VmGasLimits;
```

B.1.5. TVM library environment. The TVM library environment can be serialized as follows:

```
_ libraries:(HashMapE 256 ^Cell) = VmLibraries;
```

B.1.6. TVM continuations. Continuations in TVM can be serialized as follows:

```
vmc_std$00 nargs:(## 22) stack:(Maybe VmStack) save:VmSaveList
   cp:int16 code:VmCellSlice = VmCont;
vmc_envelope$01 nargs:(## 22) stack:(Maybe VmStack)
   save:VmSaveList next:^VmCont = VmCont;
vmc_quit$1000 exit_code:int32 = VmCont;
vmc_quit_exc$1001 = VmCont;
vmc_until$1010 body:^VmCont after:^VmCont = VmCont;
vmc_again$1011 body:^VmCont = VmCont;
vmc_while_cond$1100 cond:^VmCont body:^VmCont
   after:^VmCont = VmCont;
vmc_while_body$1101 cond:^VmCont body:^VmCont
   after:^VmCont = VmCont;
vmc_pushint$1111 value:int32 next:^VmCont = VmCont;
```

B.1.7. TVM state. The total state of TVM can be serialized as follows:

```
vms_init$00 cp:int16 step:int32 gas:GasLimits
  stack:(Maybe VmStack) save:VmSaveList code:VmCellSlice
  lib:VmLibraries = VmState;
vms_exception$01 cp:int16 step:int32 gas:GasLimits
  exc_no:int32 exc_arg:VmStackValue
  save:VmSaveList lib:VmLibraries = VmState;
```

```
vms_running$10 cp:int16 step:int32 gas:GasLimits stack:VmStack
  save:VmSaveList code:VmCellSlice lib:VmLibraries
  = VmState;
vms_finished$11 cp:int16 step:int32 gas:GasLimits
  exit_code:int32 no_gas:Boolean stack:VmStack
  save:VmSaveList lib:VmLibraries = VmState;
```

When TVM is initialized, its state is described by a `vms_init`, usually with `step` set to zero. The step function of TVM does nothing to a `vms_finished` state, and transforms all other states into `vms_running`, `vms_exception`, or `vms_finished`, with `step` increased by one.

B.2 Step function of TVM

A formal specification of TVM would be completed by the definition of a *step function* $f : \mathit{VmState} \to \mathit{VmState}$. This function deterministically transforms a valid VM state into a valid subsequent VM state, and is allowed to throw exceptions or return an invalid subsequent state if the original state was invalid.

B.2.1. A high-level definition of the step function. We might present a very long formal definition of the TVM step function in a high-level functional programming language. Such a specification, however, would mostly be useful as a reference for the (human) developers. We have chosen another approach, better adapted to automated formal verification by computers.

B.2.2. An operational definition of the step function. Notice that the step function f is a well-defined computable function from trees of cells into trees of cells. As such, it can be computed by a universal Turing machine. Then a program P computing f on such a machine would provide a machine-checkable specification of the step function f. This program P effectively is an *emulator* of TVM on this Turing machine.

B.2.3. A reference implementation of the TVM emulator inside TVM. We see that the step function of TVM may be defined by a reference implementation of a TVM emulator on another machine. An obvious idea is to use TVM itself, since it is well-adapted to working with trees of cells. However, an emulator of TVM inside itself is not very useful if we have doubts about a particular implementation of TVM and want to check it. For

instance, if such an emulator interpreted a `DICTISET` instruction simply by invoking this instruction itself, then a bug in the underlying implementation of TVM would remain unnoticed.

B.2.4. Reference implementation inside a minimal version of TVM. We see that using TVM itself as a host machine for a reference implementation of TVM emulator would yield little insight. A better idea is to define a *stripped-down version of TVM*, which supports only the bare minimum of primitives and 64-bit integer arithmetic, and provide a reference implementation P of the TVM step function f for this stripped-down version of TVM.

In that case, one must carefully implement and check only a handful of primitives to obtain a stripped-down version of TVM, and compare the reference implementation P running on this stripped-down version to the full custom TVM implementation being verified. In particular, if there are any doubts about the validity of a specific run of a custom TVM implementation, they can now be easily resolved with the aid of the reference implementation.

B.2.5. Relevance for the TON Blockchain. The TON Blockchain adopts this approach to validate the runs of TVM (e.g., those used for processing inbound messages by smart contracts) when the validators' results do not match one another. In this case, a reference implementation of TVM, stored inside the masterchain as a configurable parameter (thus defining the current revision of TVM), is used to obtain the correct result.

B.2.6. Codepage -1. *Codepage* -1 of TVM is reserved for the stripped-down version of TVM. Its main purpose is to execute the reference implementation of the step function of the full TVM. This codepage contains only special versions of arithmetic primitives working with "tiny integers" (64-bit signed integers); therefore, TVM's 257-bit *Integer* arithmetic must be defined in terms of 64-bit arithmetic. Elliptic curve cryptography primitives are also implemented directly in codepage -1, without using any third-party libraries. Finally, a reference implementation of the SHA256 hash function is also provided in codepage -1.

B.2.7. Codepage -2. This bootstrapping process could be iterated even further, by providing an emulator of the stripped-down version of TVM written for an even simpler version of TVM that supports only boolean values (or integers 0 and 1)—a "codepage -2". All 64-bit arithmetic used in codepage -1 would then need to be defined by means of boolean operations, thus

providing a reference implementation for the stripped-down version of TVM used in codepage −1. In this way, if some of the TON Blockchain validators did not agree on the results of their 64-bit arithmetic, they could regress to this reference implementation to find the correct answer.[29]

[29]The preliminary version of TVM does not use codepage −2 for this purpose. This may change in the future.

C Code density of stack and register machines

This appendix extends the general consideration of stack manipulation primitives provided in **2.2**, explaining the choice of such primitives for TVM, with a comparison of stack machines and register machines in terms of the quantity of primitives used and the code density. We do this by comparing the machine code that might be generated by an optimizing compiler for the same source files, for different (abstract) stack and register machines.

It turns out that the stack machines (at least those equipped with the basic stack manipulation primitives described in **2.2.1**) have far superior code density. Furthermore, the stack machines have excellent extendability with respect to additional arithmetic and arbitrary data processing operations, especially if one considers machine code automatically generated by optimizing compilers.

C.1 Sample leaf function

We start with a comparison of machine code generated by an (imaginary) optimizing compiler for several abstract register and stack machines, corresponding to the same high-level language source code that contains the definition of a leaf function (i.e., a function that does not call any other functions). For both the register machines and stack machines, we observe the notation and conventions introduced in **2.1**.

C.1.1. Sample source file for a leaf function. The source file we consider contains one function f that takes six (integer) arguments, a, b, c, d, e, f, and returns two (integer) values, x and y, which are the solutions of the system of two linear equations

$$\begin{cases} ax + by &= e \\ cx + dy &= f \end{cases} \tag{6}$$

The source code of the function, in a programming language similar to C, might look as follows:

```
(int, int) f(int a, int b, int c, int d, int e, int f) {
  int D = a*d - b*c;
  int Dx = e*d - b*f;
  int Dy = a*f - e*c;
```

```
  return (Dx / D, Dy / D);
}
```

We assume (cf. **2.1**) that the register machines we consider accept the six parameters $a \ldots f$ in registers `r0`...`r5`, and return the two values x and y in `r0` and `r1`. We also assume that the register machines have 16 registers, and that the stack machine can directly access `s0` to `s15` by its stack manipulation primitives; the stack machine will accept the parameters in `s5` to `s0`, and return the two values in `s0` and `s1`, somewhat similarly to the register machine. Finally, we assume at first that the register machine is allowed to destroy values in all registers (which is slightly unfair towards the stack machine); this assumption will be revisited later.

C.1.2. Three-address register machine. The machine code (or rather the corresponding assembly code) for a three-address register machine (cf. **2.1.7**) might look as follows:

```
IMUL r6,r0,r3  // r6 := r0 * r3 = ad
IMUL r7,r1,r2  // r7 := bc
SUB r6,r6,r7   // r6 := ad-bc = D
IMUL r3,r4,r3  // r3 := ed
IMUL r1,r1,r5  // r1 := bf
SUB r3,r3,r1   // r3 := ed-bf = Dx
IMUL r1,r0,r5  // r1 := af
IMUL r7,r4,r2  // r7 := ec
SUB r1,r1,r7   // r1 := af-ec = Dy
IDIV r0,r3,r6  // x := Dx/D
IDIV r1,r1,r6  // y := Dy/D
RET
```

We have used 12 operations and at least 23 bytes (each operation uses $3 \times 4 = 12$ bits to indicate the three registers involved, and at least 4 bits to indicate the operation performed; thus we need two or three bytes to encode each operation). A more realistic estimate would be 34 (three bytes for each arithmetic operation) or 31 bytes (two bytes for addition and subtraction, three bytes for multiplication and division).

C.1.3. Two-address register machine. The machine code for a two-address register machine might look as follows:

```
MOV r6,r0   // r6 := r0 = a
MOV r7,r1   // r7 := b
IMUL r6,r3  // r6 := r6*r3 = ad
IMUL r7,r2  // r7 := bc
IMUL r3,r4  // r3 := de
IMUL r1,r5  // r1 := bf
SUB r6,r7   // r6 := ad-bc = D
IMUL r5,r0  // r5 := af
SUB r3,r1   // r3 := de-bf = Dx
IMUL r2,r4  // r2 := ce
MOV r0,r3   // r0 := Dx
SUB r5,r2   // r5 := af-ce = Dy
IDIV r0,r6  // r0 := x = Dx/D
MOV r1,r5   // r1 := Dy
IDIV r1,r6  // r1 := Dy/D
RET
```

We have used 16 operations; optimistically assuming each of them (with the exception of `RET`) can be encoded by two bytes, this code would require 31 bytes.[30]

C.1.4. One-address register machine. The machine code for a one-address register machine might look as follows:

```
MOV r8,r0  // r8 := r0 = a
XCHG r1    // r0 <-> r1; r0 := b, r1 := a
MOV r6,r0  // r6 := b
IMUL r2    // r0 := r0*r2; r0 := bc
MOV r7,r0  // r7 := bc
MOV r0,r8  // r0 := a
IMUL r3    // r0 := ad
```

[30]It is interesting to compare this code with that generated by optimizing C compilers for the x86-64 architecture.

First of all, the integer division operation for x86-64 uses the one-address form, with the (double-length) dividend to be supplied in accumulator pair `r2:r0`. The quotient is also returned in `r0`. As a consequence, two single-to-double extension operations (`CDQ` or `CQO`) and at least one move operation need to be added.

Secondly, the encoding used for arithmetic and move operations is less optimistic than in our example above, requiring about three bytes per operation on average. As a result, we obtain a total of 43 bytes for 32-bit integers, and 68 bytes for 64-bit integers.

```
SUB r7     // r0 := ad-bc = D
XCHG r1    // r1 := D,  r0 := b
IMUL r5    // r0 := bf
XCHG r3    // r0 := d,  r3 := bf
IMUL r4    // r0 := de
SUB r3     // r0 := de-bf = Dx
IDIV r1    // r0 := Dx/D = x
XCHG r2    // r0 := c,  r2 := x
IMUL r4    // r0 := ce
XCHG r5    // r0 := f,  r5 := ce
IMUL r8    // r0 := af
SUB r5     // r0 := af-ce = Dy
IDIV r1    // r0 := Dy/D = y
MOV r1,r0  // r1 := y
MOV r0,r2  // r0 := x
RET
```

We have used 23 operations; if we assume one-byte encoding for all arithmetic operations and `XCHG`, and two-byte encodings for `MOV`, the total size of the code will be 29 bytes. Notice, however, that to obtain the compact code shown above we had to choose a specific order of computation, and made heavy use of the commutativity of multiplication. (For example, we compute bc before af, and $af - bc$ immediately after af.) It is not clear whether a compiler would be able to make all such optimizations by itself.

C.1.5. Stack machine with basic stack primitives. The machine code for a stack machine equipped with basic stack manipulation primitives described in **2.2.1** might look as follows:

```
PUSH s5    // a b c d e f a
PUSH s3    // a b c d e f a d
IMUL       // a b c d e f ad
PUSH s5    // a b c d e f ad b
PUSH s5    // a b c d e f ad b c
IMUL       // a b c d e f ad bc
SUB        // a b c d e f ad-bc
XCHG s3    // a b c ad-bc e f d
PUSH s2    // a b c ad-bc e f d e
IMUL       // a b c ad-bc e f de
```

```
XCHG s5    // a de c ad-bc e f b
PUSH s1    // a de c ad-bc e f b f
IMUL       // a de c ad-bc e f bf
XCHG s1,s5 // a f c ad-bc e de bf
SUB        // a f c ad-bc e de-bf
XCHG s3    // a f de-bf ad-bc e c
IMUL       // a f de-bf ad-bc ec
XCHG s3    // a ec de-bf ad-bc f
XCHG s1,s4 // ad-bc ec de-bf a f
IMUL       // D ec Dx af
XCHG s1    // D ec af Dx
XCHG s2    // D Dx af ec
SUB        // D Dx Dy
XCHG s1    // D Dy Dx
PUSH s2    // D Dy Dx D
IDIV       // D Dy x
XCHG s2    // x Dy D
IDIV       // x y
RET
```

We have used 29 operations; assuming one-byte encodings for all stack operations involved (including `XCHG s1,s`(i)), we have used 29 code bytes as well. Notice that with one-byte encoding, the "unsystematic" operation `ROT` (equivalent to `XCHG s1; XCHG s2`) would reduce the operation and byte count to 28. This shows that such "unsystematic" operations, borrowed from Forth, may indeed reduce the code size on some occasions.

Notice as well that we have implicitly used the commutativity of multiplication in this code, computing $de - bf$ instead of $ed - bf$ as specified in high-level language source code. If we were not allowed to do so, an extra `XCHG s1` would need to be inserted before the third `IMUL`, increasing the total size of the code by one operation and one byte.

The code presented above might have been produced by a rather unsophisticated compiler that simply computed all expressions and subexpressions in the order they appear, then rearranged the arguments near the top of the stack before each operation as outlined in **2.2.2**. The only "manual" optimization done here involves computing ec before af; one can check that the other order would lead to slightly shorter code of 28 operations and bytes (or 29, if we are not allowed to use the commutativity of multiplication), but

the ROT optimization would not be applicable.

C.1.6. Stack machine with compound stack primitives. A stack machine with compound stack primitives (cf. **2.2.3**) would not significantly improve code density of the code presented above, at least in terms of bytes used. The only difference is that, if we were not allowed to use commutativity of multiplication, the extra XCHG s1 inserted before the third IMUL might be combined with two previous operations XCHG s3, PUSH s2 into one compound operation PUXC s2,s3; we provide the resulting code below. To make this less redundant, we show a version of the code that computes subexpression af before ec as specified in the original source file. We see that this replaces six operations (starting from line 15) with five other operations, and disables the ROT optimization:

```
PUSH s5    // a b c d e f a
PUSH s3    // a b c d e f a d
IMUL       // a b c d e f ad
PUSH s5    // a b c d e f ad b
PUSH s5    // a b c d e f ad b c
IMUL       // a b c d e f ad bc
SUB        // a b c d e f ad-bc
PUXC s2,s3 // a b c ad-bc e f e d
IMUL       // a b c ad-bc e f ed
XCHG s5    // a ed c ad-bc e f b
PUSH s1    // a ed c ad-bc e f b f
IMUL       // a ed c ad-bc e f bf
XCHG s1,s5 // a f c ad-bc e ed bf
SUB        // a f c ad-bc e ed-bf
XCHG s4    // a ed-bf c ad-bc e f
XCHG s1,s5 // e Dx c D a f
IMUL       // e Dx c D af
XCHG s2    // e Dx af D c
XCHG s1,s4 // D Dx af e c
IMUL       // D Dx af ec
SUB        // D Dx Dy
XCHG s1    // D Dy Dx
PUSH s2    // D Dy Dx D
IDIV       // D Dy x
XCHG s2    // x Dy D
```

```
IDIV       // x y
RET
```

We have used a total of 27 operations and 28 bytes, the same as the previous version (with the `ROT` optimization). However, we did not use the commutativity of multiplication here, so we can say that compound stack manipulation primitives enable us to reduce the code size from 29 to 28 bytes.

Yet again, notice that the above code might have been generated by an unsophisticated compiler. Manual optimizations might lead to more compact code; for instance, we could use compound operations such as `XCHG3` to prepare in advance not only the correct values of `s0` and `s1` for the next arithmetic operation, but also the value of `s2` for the arithmetic operation after that. The next section provides an example of such an optimization.

C.1.7. Stack machine with compound stack primitives and manually optimized code. The previous version of code for a stack machine with compound stack primitives can be manually optimized as follows.

By interchanging `XCHG` operations with preceding `XCHG`, `PUSH`, and arithmetic operations whenever possible, we obtain code fragment `XCHG s2,s6`; `XCHG s1,s0`; `XCHG s0,s5`, which can then be replaced by compound operation `XCHG3 s6,s0,s5`. This compound operation would admit a two-byte encoding, thus leading to 27-byte code using only 21 operations:

```
PUSH2 s5,s2    // a b c d e f a d
IMUL           // a b c d e f ad
PUSH2 s5,s4    // a b c d e f ad b c
IMUL           // a b c d e f ad bc
SUB            // a b c d e f ad-bc
PUXC s2,s3     // a b c ad-bc e f e d
IMUL           // a b c D e f ed
XCHG3 s6,s0,s5 //  (same as XCHG s2,s6; XCHG s1,s0; XCHG s0,s5)
               // e f c D a ed b
PUSH s5        // e f c D a ed b f
IMUL           // e f c D a ed bf
SUB            // e f c D a ed-bf
XCHG s4        // e Dx c D a f
IMUL           // e Dx c D af
XCHG2 s4,s2    // D Dx af e c
IMUL           // D Dx af ec
```

```
SUB             // D Dx Dy
XCPU s1,s2      // D Dy Dx D
IDIV            // D Dy x
XCHG s2         // x Dy D
IDIV            // x y
RET
```

It is interesting to note that this version of stack machine code contains only 9 stack manipulation primitives for 11 arithmetic operations. It is not clear, however, whether an optimizing compiler would be able to reorganize the code in such a manner by itself.

C.2 Comparison of machine code for sample leaf function

Table **1** summarizes the properties of machine code corresponding to the same source file described in **C.1.1**, generated for a hypothetical three-address register machine (cf. **C.1.2**), with both "optimistic" and "realistic" instruction encodings; a two-address machine (cf. **C.1.3**); a one-address machine (cf. **C.1.4**); and a stack machine, similar to TVM, using either only the basic stack manipulation primitives (cf. **C.1.5**) or both the basic and the composite stack primitives (cf. **C.1.7**).

The meaning of the columns in Table **1** is as follows:

- "Operations" — The quantity of instructions used, split into "data" (i.e., register move and exchange instructions for register machines, and stack manipulation instructions for stack machines) and "arithmetic" (instructions for adding, subtracting, multiplying and dividing integer numbers). The "total" is one more than the sum of these two, because there is also a one-byte `RET` instruction at the end of machine code.

- "Code bytes" — The total amount of code bytes used.

- "Opcode space" — The portion of "opcode space" (i.e., of possible choices for the first byte of the encoding of an instruction) used by data and arithmetic instructions in the assumed instruction encoding. For example, the "optimistic" encoding for the three-address machine assumes two-byte encodings for all arithmetic instructions *op* `r`(i)`,` `r`(j)`,` `r`(k). Each arithmetic instruction would then consume portion

	Operations			Code bytes			Opcode space		
Machine	data	arith	total	data	arith	**total**	data	**arith**	total
3-addr. (opt.)	0	11	12	0	22	**23**	0/256	**64/256**	65/256
3-addr. (real.)	0	11	12	0	30	**31**	0/256	**34/256**	35/256
2-addr.	4	11	16	8	22	**31**	1/256	**4/256**	6/256
1-addr.	11	11	23	17	11	**29**	17/256	**64/256**	82/256
stack (basic)	16	11	28	16	11	**28**	64/256	**4/256**	69/256
stack (comp.)	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 1: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address, and stack machines, generated for a sample leaf function (cf. **C.1.1**). The two most important columns, reflecting **code density** and **extendability** to other operations, are marked by bold font. Smaller values are better in both of these columns.

16/256 = 1/16 of the opcode space. Notice that for the stack machine we have assumed one-byte encodings for `XCHG s`(i), `PUSH s`(i) and `POP s`(i) in all cases, augmented by `XCHG s1,s`(i) for the basic stack instructions case only. As for the compound stack operations, we have assumed two-byte encodings for `PUSH3`, `SWAP3`, `XCHG2`, `XCPU`, `PUXC`, `PUSH2`, but not for `XCHG s1,s`(i).

The "code bytes" column reflects the density of the code for the specific sample source. However, "opcode space" is also important, because it reflects the extendability of the achieved density to other classes of operations (e.g., if one were to complement arithmetic operations with string manipulation operations and so on). Here the "arithmetic" subcolumn is more important than the "data" subcolumn, because no further data manipulation operations would be required for such extensions.

We see that the three-address register machine with the "optimistic" encoding, assuming two-byte encodings for all three-register arithmetic operations, achieves the best code density, requiring only 23 bytes. However, this comes at a price: each arithmetic operation consumes 1/16 of the opcode space, so the four operations already use a quarter of the opcode space. At most 11 other operations, arithmetic or not, might be added to this architecture while preserving such high code density. On the other hand, when we consider the "realistic" encoding for the three-address machine, using two-byte encodings only for the most frequently used addition/subtraction operations (and longer encodings for less frequently used multiplication/division operations, reflecting the fact that the possible extension operations would likely fall in this class), then the three-address machine ceases to offer such

attractive code density.

In fact, the two-address machine becomes equally attractive at this point: it is capable of achieving the same code size of 31 bytes as the three-address machine with the "realistic" encoding, using only 6/256 of the opcode space for this! However, 31 bytes is the worst result in this table.

The one-address machine uses 29 bytes, slightly less than the two-address machine. However, it utilizes a quarter of the opcode space for its arithmetic operations, hampering its extendability. In this respect it is similar to the three-address machine with the "optimistic" encoding, but requires 29 bytes instead of 23! So there is no reason to use the one-address machine at all, in terms of extendability (reflected by opcode space used for arithmetic operations) compared to code density.

Finally, the stack machine wins the competition in terms of code density (27 or 28 bytes), losing only to the three-address machine with the "optimistic" encoding (which, however, is terrible in terms of extendability).

To summarize: the two-address machine and stack machine achieve the best extendability with respect to additional arithmetic or data processing instructions (using only 1/256 of code space for each such instruction), while the stack machine additionally achieves the best code density by a small margin. The stack machine utilizes a significant part of its code space (more than a quarter) for data (i.e., stack) manipulation instructions; however, this does not seriously hamper extendability, because the stack manipulation instructions occupy a constant part of the opcode stace, regardless of all other instructions and extensions.

While one might still be tempted to use a two-address register machine, we will explain shortly (cf. **C.3**) why the two-address register machine offers worse code density and extendability in practice than it appears based on this table.

As for the choice between a stack machine with only basic stack manipulation primitives or one supporting compound stack primitives as well, the case for the more sophisticated stack machine appears to be weaker: it offers only one or two fewer bytes of code at the expense of using considerably more opcode space for stack manipulation, and the optimized code using these additional instructions is hard for programmers to write and for compilers to automatically generate.

C.2.1. Register calling conventions: some registers must be preserved by functions. Up to this point, we have considered the machine

code of only one function, without taking into account the interplay between this function and other functions in the same program.

Usually a program consists of more than one function, and when a function is not a "simple" or "leaf" function, it must call other functions. Therefore, it becomes important whether a called function preserves all or at least some registers. If it preserves all registers except those used to return results, the caller can safely keep its local and temporary variables in certain registers; however, the callee needs to save all the registers it will use for its temporary values somewhere (usually into the stack, which also exists on register machines), and then restore the original values. On the other hand, if the called function is allowed to destroy all registers, it can be written in the manner described in **C.1.2**, **C.1.3**, and **C.1.4**, but the caller will now be responsible for saving all its temporary values into the stack before the call, and restoring these values afterwards.

In most cases, calling conventions for register machines require preservation of some but not all registers. We will assume that $m \le n$ registers will be preserved by functions (unless they are used for return values), and that these registers are $\mathtt{r}(n-m) \ldots \mathtt{r}(n-1)$. Case $m = 0$ corresponds to the case "the callee is free to destroy all registers" considered so far; it is quite painful for the caller. Case $m = n$ corresponds to the case "the callee must preserve all registers"; it is quite painful for the callee, as we will see in a moment. Usually a value of m around $n/2$ is used in practice.

The following sections consider cases $m = 0$, $m = 8$, and $m = 16$ for our register machines with $n = 16$ registers.

C.2.2. Case $m = 0$: no registers to preserve. This case has been considered and summarized in **C.2** and Table **1** above.

C.2.3. Case $m = n = 16$: all registers must be preserved. This case is the most painful one for the called function. It is especially difficult for leaf functions like the one we have been considering, which do not benefit at all from the fact that other functions preserve some registers when called—they do not call any functions, but instead must preserve all registers themselves.

In order to estimate the consequences of assuming $m = n = 16$, we will assume that all our register machines are equipped with a stack, and with one-byte instructions `PUSH` $\mathtt{r}(i)$ and `POP` $\mathtt{r}(i)$, which push or pop a register into/from the stack. For example, the three-address machine code provided in **C.1.2** destroys the values in registers `r2`, `r3`, `r6`, and `r7`; this means that the code of this function must be augmented by four instructions `PUSH`

		Operations			Code bytes			Opcode space		
Machine	r	data	arith	total	data	arith	**total**	data	**arith**	total
3-addr. (opt.)	*4*	8	11	20	8	22	**31**	32/256	**64/256**	97/256
3-addr. (real.)	*4*	8	11	20	8	30	**39**	32/256	**34/256**	67/256
2-addr.	*5*	14	11	26	18	22	**41**	33/256	**4/256**	38/256
1-addr.	*6*	23	11	35	29	11	**41**	49/256	**64/256**	114/256
stack (basic)	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
stack (comp.)	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 2: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address, and stack machines, generated for a sample leaf function (cf. **C.1.1**), assuming all of the 16 registers must be preserved by called functions ($m = n = 16$). The new column labeled r denotes the number of registers to be saved and restored, leading to $2r$ more operations and code bytes compared to Table **1**. Newly-added `PUSH` and `POP` instructions for register machines also utilize 32/256 of the opcode space. The two rows corresponding to stack machines remain unchanged.

`r2`; `PUSH r3`; `PUSH r6`; `PUSH r7` at the beginning, and by four instructions `POP r7`; `POP r6`; `POP r3`; `POP r2` right before the `RET` instruction, in order to restore the original values of these registers from the stack. These four additional `PUSH`/`POP` pairs would increase the operation count and code size in bytes by $4 \times 2 = 8$. A similar analysis can be done for other register machines as well, leading to Table **2**.

We see that under these assumptions the stack machines are the obvious winners in terms of code density, and are in the winning group with respect to extendability.

C.2.4. Case $m = 8$, $n = 16$: registers `r8...r15` must be preserved. The analysis of this case is similar to the previous one. The results are summarized in Table **3**.

Notice that the resulting table is very similar to Table **1**, apart from the "Opcode space" columns and the row for the one-address machine. Therefore, the conclusions of **C.2** still apply in this case, with some minor modifications. We must emphasize, however, that *these conclusions are valid only for leaf functions, i.e., functions that do not call other functions.* Any program aside from the very simplest will have many non-leaf functions, especially if we are minimizing resulting machine code size (which prevents inlining of functions in most cases).

C.2.5. A fairer comparison using a binary code instead of a byte code. The reader may have noticed that our preceding discussion of k-

		Operations			Code bytes			Opcode space		
Machine	r	data	arith	total	data	arith	**total**	data	**arith**	total
3-addr. (opt.)	*0*	0	11	12	0	22	**23**	32/256	**64/256**	97/256
3-addr. (real.)	*0*	0	11	12	0	30	**31**	32/256	**34/256**	67/256
2-addr.	*0*	4	11	16	8	22	**31**	33/256	**4/256**	38/256
1-addr.	*1*	13	11	25	19	11	**31**	49/256	**64/256**	114/256
stack (basic)	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
stack (comp.)	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 3: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address and stack machines, generated for a sample leaf function (cf. **C.1.1**), assuming that only the last 8 of the 16 registers must be preserved by called functions ($m = 8$, $n = 16$). This table is similar to Table **2**, but has smaller values of r.

address register machines and stack machines depended very much on our insistence that complete instructions be encoded by an integer number of bytes. If we had been allowed to use a "bit" or "binary code" instead of a byte code for encoding instructions, we could more evenly balance the opcode space used by different machines. For instance, the opcode of `SUB` for a three-address machine had to be either 4-bit (good for code density, bad for opcode space) or 12-bit (very bad for code density), because the complete instruction has to occupy a multiple of eight bits (e.g., 16 or 24 bits), and $3 \cdot 4 = 12$ of those bits have to be used for the three register names.

Therefore, let us get rid of this restriction.

Now that we can use any number of bits to encode an instruction, we can choose all opcodes of the same length for all the machines considered. For instance, all arithmetic instructions can have 8-bit opcodes, as the stack machine does, using 1/256 of the opcode space each; then the three-address register machine will use 20 bits to encode each complete arithmetic instruction. All `MOV`s, `XCHG`s, `PUSH`es, and `POP`s on register machines can be assumed to have 4-bit opcodes, because this is what we do for the most common stack manipulation primitives on a stack machine. The results of these changes are shown in Table **4**.

We can see that the performance of the various machines is much more balanced, with the stack machine still the winner in terms of the code density, but with the three-address machine enjoying the second place it really merits. If we were to consider the decoding speed and the possibility of parallel execution of instructions, we would have to choose the three-address machine, because it uses only 12 instructions instead of 21.

Machine	r	Operations data	arith	total	Code bytes data	arith	**total**	Opcode space data	**arith**	total
3-addr.	*0*	0	11	12	0	27.5	**28.5**	64/256	**4/256**	69/256
2-addr.	*0*	4	11	16	6	22	**29**	64/256	**4/256**	69/256
1-addr.	*1*	13	11	25	16	16.5	**32.5**	64/256	**4/256**	69/256
stack (basic)	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
stack (comp.)	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 4: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address and stack machines, generated for a sample leaf function (cf. **C.1.1**), assuming that only 8 of the 16 registers must be preserved by functions ($m = 8$, $n = 16$). This time we can use fractions of bytes to encode instructions, so as to match opcode space used by different machines. All arithmetic instructions have 8-bit opcodes, all data/stack manipulation instructions have 4-bit opcodes. In other respects this table is similar to Table **3**.

C.3 Sample non-leaf function

This section compares the machine code for different register machines for a sample non-leaf function. Again, we assume that either $m = 0$, $m = 8$, or $m = 16$ registers are preserved by called functions, with $m = 8$ representing the compromise made by most modern compilers and operating systems.

C.3.1. Sample source code for a non-leaf function. A sample source file may be obtained by replacing the built-in integer type with a custom *Rational* type, represented by a pointer to an object in memory, in our function for solving systems of two linear equations (cf. **C.1.1**):

```
struct Rational;
typedef struct Rational *num;
extern num r_add(num, num);
extern num r_sub(num, num);
extern num r_mul(num, num);
extern num r_div(num, num);

(num, num) r_f(num a, num b, num c, num d, num e, num f) {
  num D = r_sub(r_mul(a, d), r_mul(b, c));   // a*d-b*c
  num Dx = r_sub(r_mul(e, d), r_mul(b, f));  // e*d-b*f
  num Dy = r_sub(r_mul(a, f), r_mul(e, c));  // a*f-e*c
  return (r_div(Dx, D), r_div(Dy, D));      // Dx/D, Dy/D
}
```

We will ignore all questions related to allocating new objects of type *Rational* in memory (e.g., in heap), and to preventing memory leaks. We may assume that the called subroutines `r_sub`, `r_mul`, and so on allocate new objects simply by advancing some pointer in a pre-allocated buffer, and that unused objects are later freed by a garbage collector, external to the code being analysed.

Rational numbers will now be represented by pointers, addresses, or references, which will fit into registers of our hypothetical register machines or into the stack of our stack machines. If we want to use TVM as an instance of these stack machines, we should use values of type *Cell* to represent such references to objects of type *Rational* in memory.

We assume that subroutines (or functions) are called by a special `CALL` instruction, which is encoded by three bytes, including the specification of the function to be called (e.g., the index in a "global function table").

C.3.2. Three-address and two-address register machines, $m = 0$ preserved registers. Because our sample function does not use built-in arithmetic instructions at all, compilers for our hypothetical three-address and two-address register machines will produce the same machine code. Apart from the previously introduced `PUSH r`(i) and `POP r`(i) one-byte instructions, we assume that our two- and three-address machines support the following two-byte instructions: `MOV r`(i)`,s`(j), `MOV s`(j)`,r`(i), and `XCHG r`(i)`,s`(j), for $0 \le i, j \le 15$. Such instructions occupy only $3/256$ of the opcode space, so their addition seems quite natural.

We first assume that $m = 0$ (i.e., that all subroutines are free to destroy the values of all registers). In this case, our machine code for `r_f` does not have to preserve any registers, but has to save all registers containing useful values into the stack before calling any subroutines. A size-optimizing compiler might produce the following code:

```
PUSH r4     // STACK: e
PUSH r1     // STACK: e b
PUSH r0     //     .. e b a
PUSH r6     //     .. e b a f
PUSH r2     //     .. e b a f c
PUSH r3     //     .. e b a f c d
MOV r0,r1   // b
MOV r1,r2   // c
CALL r_mul  // bc
```

```
PUSH r0     //     .. e b a f c d bc
MOV r0,s4   // a
MOV r1,s1   // d
CALL r_mul  // ad
POP r1      // bc; .. e b a f c d
CALL r_sub  // D:=ad-bc
XCHG r0,s4  // b ; .. e D a f c d
MOV r1,s2   // f
CALL r_mul  // bf
POP r1      // d ; .. e D a f c
PUSH r0     //     .. e D a f c bf
MOV r0,s5   // e
CALL r_mul  // ed
POP r1      // bf; .. e D a f c
CALL r_sub  // Dx:=ed-bf
XCHG r0,s4  // e ; .. Dx D a f c
POP r1      // c ; .. Dx D a f
CALL r_mul  // ec
XCHG r0,s1  // a ; .. Dx D ec f
POP r1      // f ; .. Dx D ec
CALL r_mul  // af
POP r1      // ec; .. Dx D
CALL r_sub  // Dy:=af-ec
XCHG r0,s1  // Dx; .. Dy D
MOV r1,s0   // D
CALL r_div  // x:=Dx/D
XCHG r0,s1  // Dy; .. x D
POP r1      // D ; .. x
CALL r_div  // y:=Dy/D
MOV r1,r0   // y
POP r0      // x ; ..
RET
```

We have used 41 instructions: 17 one-byte (eight `PUSH`/`POP` pairs and one `RET`), 13 two-byte (`MOV` and `XCHG`; out of them 11 "new" ones, involving the stack), and 11 three-byte (`CALL`), for a total of $17 \cdot 1 + 13 \cdot 2 + 11 \cdot 3 = 76$ bytes.[31]

[31] Code produced for this function by an optimizing compiler for x86-64 architecture

C.3.3. Three-address and two-address register machines, $m = 8$ preserved registers. Now we have eight registers, `r8` to `r15`, that are preserved by subroutine calls. We might keep some intermediate values there instead of pushing them into the stack. However, the penalty for doing so consists in a `PUSH`/`POP` pair for every such register that we choose to use, because our function is also required to preserve its original value. It seems that using these registers under such a penalty does not improve the density of the code, so the optimal code for three- and two-address machines for $m = 8$ preserved registers is the same as that provided in **C.3.2**, with a total of 42 instructions and 74 code bytes.

C.3.4. Three-address and two-address register machines, $m = 16$ preserved registers. This time *all* registers must be preserved by the subroutines, excluding those used for returning the results. This means that our code must preserve the original values of `r2` to `r5`, as well as any other registers it uses for temporary values. A straightforward way of writing the code of our subroutine would be to push registers `r2` up to, say, `r8` into the stack, then perform all the operations required, using `r6`–`r8` for intermediate values, and finally restore registers from the stack. However, this would not optimize code size. We choose another approach:

```
PUSH r0     // STACK: a
PUSH r1     // STACK: a b
MOV r0,r1   // b
MOV r1,r2   // c
CALL r_mul  // bc
PUSH r0     //     .. a b bc
MOV r0,s2   // a
MOV r1,r3   // d
CALL r_mul  // ad
POP r1      // bc; .. a b
CALL r_sub  // D:=ad-bc
XCHG r0,s0  // b;  .. a D
MOV r1,r5   // f
CALL r_mul  // bf
PUSH r0     //     .. a D bf
```

with size-optimization enabled actually occupied 150 bytes, due mostly to the fact that actual instruction encodings are about twice as long as we had optimistically assumed.

```
MOV r0,r4   // e
MOV r1,r3   // d
CALL r_mul  // ed
POP r1      // bf; .. a D
CALL r_sub  // Dx:=ed-bf
XCHG r0,s1  // a ; .. Dx D
MOV r1,r5   // f
CALL r_mul  // af
PUSH r0     //     .. Dx D af
MOV r0,r4   // e
MOV r1,r2   // c
CALL r_mul  // ec
MOV r1,r0   // ec
POP r0      // af; .. Dx D
CALL r_sub  // Dy:=af-ec
XCHG r0,s1  // Dx; .. Dy D
MOV r1,s0   // D
CALL r_div  // x:=Dx/D
XCHG r0,s1  // Dy; .. x D
POP r1      // D ; .. x
CALL r_div  // y:=Dy/D
MOV r1,r0   // y
POP r0      // x
RET
```

We have used 39 instructions: 11 one-byte, 17 two-byte (among them 5 "new" instructions), and 11 three-byte, for a total of $11 \cdot 1 + 17 \cdot 2 + 11 \cdot 3 = 78$ bytes. Somewhat paradoxically, the code size in bytes is slightly longer than in the previous case (cf. **C.3.2**), contrary to what one might have expected. This is partially due to the fact that we have assumed two-byte encodings for "new" `MOV` and `XCHG` instructions involving the stack, similarly to the "old" instructions. Most existing architectures (such as x86-64) use longer encodings (maybe even twice as long) for their counterparts of our "new" move and exchange instructions compared to the "usual" register-register ones. Taking this into account, we see that we would have obtained here 83 bytes (versus 87 for the code in **C.3.2**) assuming three-byte encodings of new operations, and 88 bytes (versus 98) assuming four-byte encodings. This shows that, for two-address architectures without optimized encodings for register-stack

move and exchange operations, $m = 16$ preserved registers might result in slightly shorter code for some non-leaf functions, at the expense of leaf functions (cf. **C.2.3** and **C.2.4**), which would become considerably longer.

C.3.5. One-address register machine, $m = 0$ preserved registers. For our one-address register machine, we assume that new register-stack instructions work through the accumulator only. Therefore, we have three new instructions, `LD s(`j`)` (equivalent to `MOV r0,s(`j`)` of two-address machines), `ST s(`j`)` (equivalent to `MOV s(`j`),r0`), and `XCHG s(`j`)` (equivalent to `XCHG r0,s(`j`)`). To make the comparison with two-address machines more interesting, we assume one-byte encodings for these new instructions, even though this would consume $48/256 = 3/16$ of the opcode space.

By adapting the code provided in **C.3.2** to the one-address machine, we obtain the following:

```
PUSH r4     // STACK: e
PUSH r1     // STACK: e b
PUSH r0     //     .. e b a
PUSH r6     //     .. e b a f
PUSH r2     //     .. e b a f c
PUSH r3     //     .. e b a f c d
LD s1       // r0:=c
XCHG r1     // r0:=b, r1:=c
CALL r_mul  // bc
PUSH r0     //     .. e b a f c d bc
LD s1       // d
XCHG r1     // r1:=d
LD s4       // a
CALL r_mul  // ad
POP r1      // bc; .. e b a f c d
CALL r_sub  // D:=ad-bc
XCHG s4     // b ; .. e D a f c d
XCHG r1
LD s2       // f
XCHG r1     // r0:=b, r1:=f
CALL r_mul  // bf
POP r1      // d ; .. e D a f c
PUSH r0     //     .. e D a f c bf
LD s5       // e
```

```
CALL r_mul  // ed
POP r1      // bf; .. e D a f c
CALL r_sub  // Dx:=ed-bf
XCHG s4     // e ; .. Dx D a f c
POP r1      // c ; .. Dx D a f
CALL r_mul  // ec
XCHG s1     // a ; .. Dx D ec f
POP r1      // f ; .. Dx D ec
CALL r_mul  // af
POP r1      // ec; .. Dx D
CALL r_sub  // Dy:=af-ec
XCHG s1     // Dx; .. Dy D
POP r1      // D ; .. Dy
PUSH r1     //     .. Dy D
CALL r_div  // x:=Dx/D
XCHG s1     // Dy; .. x D
POP r1      // D ; .. x
CALL r_div  // y:=Dy/D
XCHG r1     // r1:=y
POP r0      // r0:=x ; ..
RET
```

We have used 45 instructions: 34 one-byte and 11 three-byte, for a total of 67 bytes. Compared to the 76 bytes used by two- and three-address machines in **C.3.2**, we see that, again, the one-address register machine code may be denser than that of two-register machines, at the expense of utilizing more opcode space (just as shown in **C.2**). However, this time the extra 3/16 of the opcode space was used for data manipulation instructions, which do not depend on specific arithmetic operations or user functions invoked.

C.3.6. One-address register machine, $m = 8$ preserved registers. As explained in **C.3.3**, the preservation of `r8`–`r15` between subroutine calls does not improve the size of our previously written code, so the one-address machine will use for $m = 8$ the same code provided in **C.3.5**.

C.3.7. One-address register machine, $m = 16$ preserved registers. We simply adapt the code provided in **C.3.4** to the one-address register machine:

```
PUSH r0     // STACK: a
```

```
PUSH r1     // STACK: a b
MOV r0,r1   // b
MOV r1,r2   // c
CALL r_mul  // bc
PUSH r0     //     .. a b bc
LD s2       // a
MOV r1,r3   // d
CALL r_mul  // ad
POP r1      // bc; .. a b
CALL r_sub  // D:=ad-bc
XCHG s0     // b;  .. a D
MOV r1,r5   // f
CALL r_mul  // bf
PUSH r0     //     .. a D bf
MOV r0,r4   // e
MOV r1,r3   // d
CALL r_mul  // ed
POP r1      // bf; .. a D
CALL r_sub  // Dx:=ed-bf
XCHG s1     // a ; .. Dx D
MOV r1,r5   // f
CALL r_mul  // af
PUSH r0     //     .. Dx D af
MOV r0,r4   // e
MOV r1,r2   // c
CALL r_mul  // ec
MOV r1,r0   // ec
POP r0      // af; .. Dx D
CALL r_sub  // Dy:=af-ec
XCHG s1     // Dx; .. Dy D
POP r1      // D ; .. Dy
PUSH r1     //     .. Dy D
CALL r_div  // x:=Dx/D
XCHG s1     // Dy; .. x D
POP r1      // D ; .. x
CALL r_div  // y:=Dy/D
MOV r1,r0   // y
POP r0      // x
```

```
RET
```

We have used 40 instructions: 18 one-byte, 11 two-byte, and 11 three-byte, for a total of $18 \cdot 1 + 11 \cdot 2 + 11 \cdot 3 = 73$ bytes.

C.3.8. Stack machine with basic stack primitives. We reuse the code provided in **C.1.5**, simply replacing arithmetic primitives (VM instructions) with subroutine calls. The only substantive modification is the insertion of the previously optional `XCHG s1` before the third multiplication, because even an optimizing compiler cannot now know whether `CALL r_mul` is a commutative operation. We have also used the "tail recursion optimization" by replacing the final `CALL r_div` followed by `RET` with `JMP r_div`.

```
PUSH s5     // a b c d e f a
PUSH s3     // a b c d e f a d
CALL r_mul  // a b c d e f ad
PUSH s5     // a b c d e f ad b
PUSH s5     // a b c d e f ad b c
CALL r_mul  // a b c d e f ad bc
CALL r_sub  // a b c d e f ad-bc
XCHG s3     // a b c ad-bc e f d
PUSH s2     // a b c ad-bc e f d e
XCHG s1     // a b c ad-bc e f e d
CALL r_mul  // a b c ad-bc e f ed
XCHG s5     // a ed c ad-bc e f b
PUSH s1     // a ed c ad-bc e f b f
CALL r_mul  // a ed c ad-bc e f bf
XCHG s1,s5  // a f c ad-bc e ed bf
CALL r_sub  // a f c ad-bc e ed-bf
XCHG s3     // a f ed-bf ad-bc e c
CALL r_mul  // a f ed-bf ad-bc ec
XCHG s3     // a ec ed-bf ad-bc f
XCHG s1,s4  // ad-bc ec ed-bf a f
CALL r_mul  // D ec Dx af
XCHG s1     // D ec af Dx
XCHG s2     // D Dx af ec
CALL r_sub  // D Dx Dy
XCHG s1     // D Dy Dx
PUSH s2     // D Dy Dx D
```

```
CALL r_div  // D Dy x
XCHG s2     // x Dy D
JMP r_div   // x y
```

We have used 29 instructions; assuming one-byte encodings for all stack operations, and three-byte encodings for `CALL` and `JMP` instructions, we end up with 51 bytes.

C.3.9. Stack machine with compound stack primitives. We again reuse the code provided in **C.1.7**, replacing arithmetic primitives with subroutine calls and making the tail recursion optimization:

```
PUSH2 s5,s2    // a b c d e f a d
CALL r_mul     // a b c d e f ad
PUSH2 s5,s4    // a b c d e f ad b c
CALL r_mul     // a b c d e f ad bc
CALL r_sub     // a b c d e f ad-bc
PUXC s2,s3     // a b c ad-bc e f e d
CALL r_mul     // a b c D e f ed
XCHG3 s6,s0,s5 //  (same as XCHG s2,s6; XCHG s1,s0; XCHG s0,s5)
               // e f c D a ed b
PUSH s5        // e f c D a ed b f
CALL r_mul     // e f c D a ed bf
CALL r_sub     // e f c D a ed-bf
XCHG s4        // e Dx c D a f
CALL r_mul     // e Dx c D af
XCHG2 s4,s2    // D Dx af e c
CALL r_mul     // D Dx af ec
CALL r_sub     // D Dx Dy
XCPU s1,s2     // D Dy Dx D
CALL r_div     // D Dy x
XCHG s2        // x Dy D
JMP r_div      // x y
```

This code uses only 20 instructions, 9 stack-related and 11 control flow-related (`CALL` and `JMP`), for a total of 48 bytes.

Machine	m	Operations data	cont.	total	Code bytes data	cont.	**total**	Opcode space data	**arith**	total
3-addr.	*0,8*	29	12	41	42	34	**76**	35/256	**34/256**	72/256
	16	27	12	39	44	34	**78**			
2-addr.	*0,8*	29	12	41	42	34	**76**	37/256	**4/256**	44/256
	16	27	12	39	44	34	**78**			
1-addr.	*0,8*	33	12	45	33	34	**67**	97/256	**64/256**	164/256
	16	28	12	40	39	34	**73**			
stack (basic)	–	18	11	29	18	33	**51**	64/256	**4/256**	71/256
stack (comp.)	–	9	11	20	15	33	**48**	84/256	**4/256**	91/256

Table 5: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address, and stack machines, generated for a sample non-leaf function (cf. **C.3.1**), assuming m of the 16 registers must be preserved by called subroutines.

C.4 Comparison of machine code for sample non-leaf function

Table **5** summarizes the properties of machine code corresponding to the same source file provided in **C.3.1**. We consider only the "realistically" encoded three-address machines. Three-address and two-address machines have the same code density properties, but differ in the utilization of opcode space. The one-address machine, somewhat surprisingly, managed to produced shorter code than the two-address and three-address machines, at the expense of using up more than half of all opcode space. The stack machine is the obvious winner in this code density contest, without compromizing its excellent extendability (measured in opcode space used for arithmetic and other data transformation instructions).

C.4.1. Combining with results for leaf functions. It is instructive to compare this table with the results in **C.2** for a sample leaf function, summarized in Table **1** (for $m = 0$ preserved registers) and the very similar Table **3** (for $m = 8$ preserved registers), and, if one is still interested in case $m = 16$ (which turned out to be worse than $m = 8$ in almost all situations), also to Table **2**.

We see that the stack machine beats all register machines on non-leaf functions. As for the leaf functions, only the three-address machine with the "optimistic" encoding of arithmetic instructions was able to beat the stack machine, winning by 15%, by compromising its extendability. However, the same three-address machine produces 25% longer code for non-leaf functions.

Machine	m	Operations data	cont.	total	Code bytes data	cont.	**total**	Opcode space data	**arith**	total
3-addr.	*0,8*	29	12	41	35.5	34	**69.5**	110/256	**4/256**	117/256
	16	27	12	39	35.5	34	**69.5**			
2-addr.	*0,8*	29	12	41	35.5	34	**69.5**	110/256	**4/256**	117/256
	16	27	12	39	35.5	34	**69.5**			
1-addr.	*0,8*	33	12	45	33	34	**67**	112/256	**4/256**	119/256
	16	28	12	40	33.5	34	**67.5**			
stack (basic)	–	18	11	29	18	33	**51**	64/256	**4/256**	71/256
stack (comp.)	–	9	11	20	15	33	**48**	84/256	**4/256**	91/256

Table 6: A summary of machine code properties for hypothetical 3-address, 2-address, 1-address, and stack machines, generated for a sample non-leaf function (cf. **C.3.1**), assuming m of the 16 registers must be preserved by called subroutines. This time we use fractions of bytes to encode instructions, enabling a fairer comparison. Otherwise, this table is similar to Table **5**.

If a typical program consists of a mixture of leaf and non-leaf functions in approximately equal proportion, then the stack machine will still win.

C.4.2. A fairer comparison using a binary code instead of a byte code. Similarly to **C.2.5**, we may offer a fairer comparison of different register machines and the stack machine by using arbitrary binary codes instead of byte codes to encode instructions, and matching the opcode space used for data manipulation and arithmetic instructions by different machines. The results of this modified comparison are summarized in Table **6**. We see that the stack machines still win by a large margin, while using less opcode space for stack/data manipulation.

C.4.3. Comparison with real machines. Note that our hypothetical register machines have been considerably optimized to produce shorter code than actually existing register machines; the latter are subject to other design considerations apart from code density and extendability, such as backward compatibility, faster instruction decoding, parallel execution of neighboring instructions, ease of automatically producing optimized code by compilers, and so on.

For example, the very popular two-address register architecture x86-64 produces code that is approximately twice as long as our "ideal" results for the two-address machines. On the other hand, our results for the stack machines are directly applicable to TVM, which has been explicitly designed with the considerations presented in this appendix in mind. Furthermore, the

actual TVM code is even *shorter* (in bytes) than shown in Table **5** because of the presence of the two-byte `CALL` instruction, allowing TVM to call up to 256 user-defined functions from the dictionary at `c3`. This means that one should subtract 10 bytes from the results for stack machines in Table **5** if one wants to specifically consider TVM, rather than an abstract stack machine; this produces a code size of approximately 40 bytes (or shorter), almost half that of an abstract two-address or three-address machine.

C.4.4. Automatic generation of optimized code. An interesting point is that the stack machine code in our samples might have been generated automatically by a very simple optimizing compiler, which rearranges values near the top of the stack appropriately before invoking each primitive or calling a function as explained in **2.2.2** and **2.2.5**. The only exception is the unimportant "manual" `XCHG3` optimization described in **C.1.7**, which enabled us to shorten the code by one more byte.

By contrast, the heavily optimized (with respect to size) code for register machines shown in **C.3.2** and **C.3.3** is unlikely to be produced automatically by an optimizing compiler. Therefore, if we had compared compiler-generated code instead of manually-generated code, the advantages of stack machines with respect to code density would have been even more striking.