Виртуальная машина

Telegram Open Network

Николай Дуров

23 марта 2020

Аннотация

Целью этого текста является предоставление описания виртуальной машины Telegram Open Network Virtual Machine (TON VM или TVM), используемой для выполнения смарт-контрактов в блокчейне TON.

Введение

Основной целью виртуальной машины Telegram Open Network (TON VM или TVM) является выполнение кода смарт-контракта в блокчейне TON. TVM должен поддерживать все операции, необходимые для синтаксического анализа входящих сообщений и постоянных данных, а также для создания новых сообщений и изменения постоянных данных. Кроме того, TVM должен соответствовать следующим требованиям:

- Он должен предусматривать возможные будущие расширения и улучшения, сохраняя при этом обратную совместимость и функциональную совместимость, потому что код смарт-контракта, однажды зафиксированный в блокчейне, должен продолжать работать предсказуемым образом независимо от любых будущих модификаций виртуальной машины.

- Он должен стремиться к достижению высокой плотности «(виртуального) машинного кода», чтобы код типичного смарт-

контракта занимал как можно меньше постоянного хранилища блокчейна.

- Она должна быть полностью детерминированной. Другими словами, каждый запуск одного и того же кода с одними и теми же входными данными должен давать один и тот же результат, независимо от конкретного используемого программного и аппаратного обеспечения.[1]

При проектировании TVM руководствуются этими требованиями. Хотя в этом документе описывается предварительная и экспериментальная версия TVM,[2] встроенные в систему механизмы обратной совместимости позволяют нам относительно не беспокоиться об эффективности кодирования операций, используемых для кода TVM в этой предварительной версии.

TVM не предназначен для реализации в аппаратном обеспечении (например, в специализированном микропроцессоре); скорее, он должен быть реализован в программном обеспечении, работающем на обычном оборудовании. Это соображение позволяет нам включить некоторые высокоуровневые концепции и операции в TVM, которые потребуют запутанного микрокода в аппаратной реализации, но не представляют существенных проблем для реализации программного обеспечения. Такие операции полезны для достижения высокой плотности кода и минимизации байтового (или запоминающей ячейки) профиля кода смарт-контракта при развертывании в блокчейне TON.

[1] Например, в TVM отсутствуют арифметические операции с плавающей запятой (которые могут быть эффективно реализованы с использованием аппаратно поддерживаемого *двойного* типа на большинстве современных процессоров), поскольку результат выполнения таких операций зависит от конкретной базовой аппаратной реализации и настроек режима округления. Вместо этого TVM поддерживает специальные целочисленные арифметические операции, которые при необходимости можно использовать для моделирования арифметики с фиксированной точкой.

[2] Производственная версия, вероятно, потребует некоторых настроек и модификаций до запуска, что станет очевидным только после использования экспериментальной версии в тестовой среде в течение некоторого времени.

Содержание

1 Обзор

В этой главе представлен обзор основных особенностей и принципов проектирования TVM. Более подробная информация по каждой теме приводится в последующих главах.

1.0 Нотация для битовых строк

Следующая нотация используется для битовых строк (или *битовых строк*), т. е. конечных строк, состоящих из двоичных цифр (битов), `0` и `1`, во всем этом документе.

1.0.1. Шестнадцатеричное обозначение для битовых строк. Когда длина битовой строки кратна четырем, мы подразделяем ее на группы по четыре бита и представляем каждую группу одной из шестнадцати шестнадцати шестнадцатеричных цифр `0`–`9`, `A`–`F` обычным способом: 0_{16} ↔ `0000`, 1_{16} ↔ `0001`, ..., F_{16} ↔ `1111` . Результирующая шестнадцатеричная строка является нашим эквивалентным представлением для исходной двоичной строки.

1.0.2. Строка битов длины, не делимые на четыре. Если длина двоичной строки не делится на четыре, мы увеличиваем ее на один `1` и несколько (возможно, ноль) 0 в конце, так что ее длина становится делимой на четыре, а затем преобразуем ее в строку шестнадцатеричных цифр, как описано выше. Чтобы указать, что такое преобразование имело место, в конец шестнадцатеричной строки добавляется специальный «тег завершения». Обратное преобразование (применяется, если присутствует тег завершения) состоит в том, чтобы сначала заменить каждую шестнадцатеричную цифру четырьмя соответствующими битами, а затем удалить все конечные нули (если таковые имеются) и последний `1` , непосредственно предшествующий им (если результирующая битовая строка на данный момент не пуста).

Обратите внимание, что существует несколько допустимых шестнадцатеричных представлений для одной и той же битовой строки. Среди них самый короткий – «канонический». Он может быть детерминированно получен с помощью вышеуказанной процедуры.

Например, `8A` соответствует двоичной строке `10001010`, в то время как `8A_` и `8A0_` соответствуют `100010`. Пустая битовая строка может быть представлена либо '', '`8_`', '`0_`', '`_`' или '`00_`'.

1.0.3. Подчеркивание того, что строка является шестнадцатеричным представлением битовой строки. Иногда нам нужно подчеркнуть, что строка шестнадцатеричных цифр (с или без `_` в конце) является шестнадцатеричным представлением битовой строки. В таких случаях мы либо предваряем `x` к результирующей строке (например, `x8A`), либо предваряем `x{` и добавляем `}` (например, `x{2D9_}`, что `00101101100`).
Это не следует путать с шестнадцатеричными числами, обычно предваряемыми `0x` (например, `0x2D9` или `0x2d9`, что является целым числом 729).

1.0.4. Сериализация битовой строки в последовательность октетов. Когда битовая строка должна быть представлена в виде последовательности 8-битных байтов (октетов), которые принимают значения в целых числах 0… 255, это достигается по существу тем же способом, что и выше: мы разбиваем битовую строку на группы по восемь бит и интерпретируем каждую группу как двоичное представление целого числа 0… 255. Если длина битовой строки не кратна восьми, битовая строка увеличивается двоичным `1` и до семи двоичных `0` перед разделением на группы. Тот факт, что такое завершение было применено, обычно отражается битом «тег завершения».

Например, `00101101100` соответствует последовательности двух октетов (*0x2d,0x90*) (шестнадцатеричный) или (45,144) (десятичный), вместе с битом тега завершения, равным `1` (что означает, что завершение было применено), который должен храниться отдельно.

В некоторых случаях удобнее предположить, что завершение включено по умолчанию, а не хранить дополнительный бит тега завершения отдельно. Согласно таким соглашениям, 8n-битные строки представлены *n* + 1 октетом, причем последний октет всегда равен `0x80` = 128.

1.1 TVM - это стековая машина

Прежде всего, *TVM - это стековая машина.* Это означает, что вместо того, чтобы хранить значения в некоторых «переменных» или «регистрах

общего назначения», они хранятся в стеке (LIFO), по крайней мере, с точки зрения «низкого уровня» (TVM).[3]

Большинство операций и пользовательских функций берут свои аргументы из верхней части стека и заменяют их результатом. Например, примитив сложения целых чисел (встроенная операция) `ADD` не принимает никаких аргументов, описывающих, какие регистры или немедленные значения должны быть сложены вместе и где должен храниться результат. Вместо этого два верхних значения берутся из стека, они складываются вместе, а их сумма выталкивается в стек на их место.

1.1.1. Значения TVM. Сущности, которые могут храниться в стеке TVM, будут называться *значениями TVM* или просто *значениями* для краткости. Они принадлежат к одному из нескольких предопределенных *типов значений*. Каждое значение принадлежит ровно одному типу значений. Значения всегда хранятся в стеке вместе с тегами, однозначно определяющими их типы, а все встроенные операции TVM (или *примитивы*) принимают только значения предопределенных типов.

Например, примитив сложения целых чисел `ADD` принимает только два целочисленных значения и в результате возвращает одно целое значение. Нельзя предоставить `ADD` две строки вместо двух целых чисел, ожидая, что она объединит эти строки или неявно преобразует строки в их десятичные целочисленные значения; любая попытка сделать это приведет к исключению проверки типов во время выполнения.

1.1.2. Статическая типизация, динамическая типизация и проверка типов во время выполнения. В некоторых отношениях TVM выполняет своего рода динамическую типизацию с использованием проверки типов во время выполнения. Однако это не делает код TVM «динамически типизированным языком», как PHP или Javascript, потому что все примитивы принимают значения и возвращают результаты предопределенных (значений) типов, каждое значение принадлежит строго одному типу, и значения никогда не преобразуются неявно из одного типа в другой. Если, с другой стороны, сравнить код TVM с обычным микропроцессорным машинным кодом, то можно увидеть, что

[3] Высокоуровневый язык смарт-контрактов может создать видимость переменных для простоты программирования; однако высокоуровневый исходный код, работающий с переменными, будет переведен в машинный код TVM, сохраняя все значения этих переменных в стеке TVM.

TVM-механизм пометки значений предотвращает, например, использование адреса строки в качестве числа или, что еще более губительно, использование числа в качестве адреса строки, тем самым устраняя возможность всевозможных ошибок и уязвимостей безопасности, связанных с недопустимыми доступами к памяти. обычно приводит к повреждению памяти и сбоям сегментации. Это свойство очень желательно для виртуальной машины, используемой для выполнения смарт-контрактов в блокчейне. В этом отношении настойчивость TVM в пометке всех значений соответствующими типами вместо того, чтобы переинтерпретировать битовую последовательность в регистре в зависимости от потребностей операции, в которой она используется, является просто дополнительным механизмом безопасности типа во время выполнения.

Альтернативой было бы каким-то образом проанализировать код смарт-контракта на корректность шрифта и безопасность шрифта, прежде чем разрешить его выполнение в виртуальной машине или даже до того, как разрешить его загрузку в блокчейн в качестве кода смарт-контракта. Такой статический анализ кода для полной машины Тьюринга кажется трудоемкой и нетривиальной проблемой (вероятно, эквивалентной проблеме остановки для машин Тьюринга), чего мы предпочли бы избежать в контексте смарт-контрактов блокчейна.

Следует иметь в виду, что всегда можно реализовать компиляторы из статически типизированных высокоуровневых языков смарт-контрактов в код TVM (и мы ожидаем, что большинство смарт-контрактов для TON будут написаны на таких языках), так же, как можно скомпилировать статически типизированные языки в обычный машинный код (например, архитектуру x86). Если компилятор работает правильно, результирующий машинный код никогда не будет генерировать исключения проверки типов во время выполнения. Все теги типов, прикрепленные к значениям, обрабатываемым TVM, всегда будут иметь ожидаемые значения и могут быть безопасно проигнорированы во время анализа результирующего кода TVM, за исключением того, что генерация во время выполнения и проверка этих тегов типа TVM немного замедлит выполнение кода TVM.

1.1.3. Предварительный перечень типов значений. Предварительный список типов значений, поддерживаемых TVM, выглядит следующим образом:

- *Integer* — Знаковые 257-битные целые числа, представляющие целые числа в диапазоне $-2^{256} \ldots 2^{256} - 1$, а также специальное значение "not-a-number" `NaN`.

- *Cell* – ячейка TVM состоит максимум из 1023 бит данных и максимум из четырех ссылок на другие клетки. Все постоянные данные (включая TVM код) в блокчейне TON представлена в виде коллекции ячеек TVM (см. [1, 2.5.14]).

- *Tuple* – упорядоченная коллекция до 255 компонентов, имеющих произвольный тип значения, возможно, различные. Может использоваться для представления непостоянных значений произвольных алгебраических типов данных.

- *Null* – тип с ровно одним значением $\perp$, используемым для представления пустых списков, пустых ветвей двоичных деревьев, отсутствия возвратного значения в некоторых ситуациях и так далее.

- *Slice* – Срез ячейки TVM, или срез для краткости, представляет собой смежную «подъячейку» существующей ячейки, содержащую некоторые из ее битов данных и некоторые из ее ссылок. По сути, фрагмент — это представление только для чтения для подячейки ячейки. Фрагменты используются для распаковки данных, ранее сохраненных (или сериализованных) в ячейке или дереве ячеек.

- *Builder* – конструктор ячеек TVM, или конструктор для краткости, представляет собой «неполную» ячейку, которая поддерживает быстрые операции добавления битовых строк и ссылок на ячейки в ее конце. Строители используются для упаковки (или сериализации) данных из верхней части стека в новые ячейки (например, перед передачей их в постоянное хранилище).
- *Continuation* — представляет собой «маркер выполнения» для TVM, который может быть вызван (выполнен) позже. Таким образом, он обобщает адреса функций (т. е. указатели функций и ссылки), адреса возврата подпрограмм, адреса указателей инструкций, адреса обработчиков исключений, закрытия, частичные приложения, анонимные функции и так далее.

Этот список типов значений является неполным и может быть расширен в будущих версиях TVM без нарушения старого кода TVM, в основном из-за того, что все первоначально определенные примитивы принимают только значения известных им типов и не будут выполняться (генерировать исключение для проверки типов) при вызове значений новых типов. Кроме того, существующие типы значений также могут быть расширены в будущем: например, 257-битное *целое число* может стать 513-битным *LongInteger*, при этом первоначально определенные арифметические примитивы терпят неудачу, если какой-либо из аргументов или результат не вписывается в исходный подтип *Integer*. Обратная совместимость в отношении введения новых типов значений и расширения существующих типов значений будет более подробно рассмотрена ниже (см. 5.1.4).

1.2 Категории инструкций TVM

TVM *Инструкции*, также *называемые примитивами*, а иногда и *(встроенными) операциями*, являются наименьшими операциями, атомарно выполняемыми TVM, которые могут присутствовать в коде TVM. Они делятся на несколько категорий, в зависимости от типов значений (см. 1.1.3), над которыми они работают. Наиболее важными из этих категорий являются:

- *Стековые (манипулятивные) примитивы* — переупорядочивают данные в стеке TVM, чтобы другие примитивы и определяемые пользователем функции могли быть впоследствии вызваны с правильными аргументами. В отличие от большинства других примитивов, они полиморфны, т. е. работают со значениями произвольных типов.

- *Кортежные (манипулятивные) примитивы* — создание, изменение и разложение *кортежей. Подобно примитивам стека, они полиморфны.*

- *Константные* или буквальные *примитивы* — вставьте в стек некоторые «постоянные» или «буквальные» значения, встроенные в сам код TVM, тем самым предоставляя аргументы другим примитивам. Они чем-то похожи на стековые примитивы, но менее универсальны, поскольку работают со значениями конкретных типов.

- *Арифметические примитивы* — Выполнение обычных целочисленных арифметических операций над значениями типа *Integer*.

- *Cell (manipulation) primitives (примитивы создания ячеек*) — создание новых ячеек и хранение в них данных (примитивы создания ячеек) или чтение данных из ранее созданных ячеек (примитивы синтаксического анализа ячеек). Поскольку вся память и

постоянное хранилище TVM состоит из клеток, эти примитивы клеточных манипуляций фактически соответствуют «инструкциям доступа к памяти» других архитектур. Примитивы создания ячеек обычно работают со значениями типа Builder, в то время как примитивы синтаксического анализа ячеек работают с Slices.

- *Продолжения* и *примитивы потока контроля* — создание и изменение *continuations, а также выполнение существующих Continuations различными способами, включая условное и повторное выполнение.*

- *Пользовательские* или *специфичные для приложения примитивы* — эффективно выполнять определенные высокоуровневые действия, требуемые приложением (в нашем случае TON Blockchain), такие как вычисление хэш-функций, выполнение криптографии эллиптической кривой, отправка новых сообщений блокчейна, создание новых смарт-контрактов и так далее. Эти примитивы соответствуют стандартным библиотечным функциям, а не микропроцессорным инструкциям.

1.3 Контрольные регистры

В то время как TVM является стековой машиной, некоторые редко изменяемые значения, необходимые почти во всех функциях, лучше передавать в определенные специальные регистры, а не в верхней части стека. В противном случае для управления всеми этими значениями потребуется непомерно большое количество операций переупорядочивания стека.

С этой целью модель TVM включает, помимо стека, до 16 специальных *регистров управления*, обозначаемых `c0` до `c15` или $c(0)$ до $c(15)$. Первоначальная версия TVM использует только некоторые из этих регистров; остальные могут быть поддержаны позже.

1.3.1. Значения, хранящиеся в контрольных регистрах. Значения, хранящиеся в регистрах управления, имеют те же типы, что и значения, хранящиеся в стеке. Однако некоторые управляющие регистры

принимают только значения определенных типов, и любая попытка загрузить значение другого типа приведет к исключению.

1.3.2. Перечень контрольных регистров. Исходная версия TVM определяет и использует следующие управляющие регистры:

- `c0` — содержит *следующее продолжение* или *возвращаемое продолжение* (аналогично возвратному адресу подпрограммы в обычных проектах). Это значение должно быть *продолжением*.
- `c1` — Содержит *альтернативное (возвращаемое) продолжение; это значение должно быть Continuation. Оно используется в некотором (экспериментальном) потоке управления, позволяющем TVM определять и вызывать «подпрограммы с двумя точками выхода».*
- `c2` — содержит *обработчик исключений. Это значение является значением Continuation, вызываемым всякий раз, когда инициируется исключение.*
- `c3` — Содержит *текущий словарь, по сути хэш-карту, содержащую код всех функций, используемых в программе. По причинам, объясненным далее в 4.6, это значение также является продолжением, а не ячейкой,* как можно было бы ожидать.
- `c4` — содержит корень *постоянных данных или просто данные.* Это значение является ячейкой. Когда вызывается код смарт-контракта, c4 указывает на корневую ячейку его постоянных данных, хранящихся в состоянии блокчейна. Если смарт-контракту необходимо изменить эти данные, он изменяет c4 перед возвратом.
- `c5` — содержит выходные *действия. Это также ячейка,* инициализированная ссылкой на пустую ячейку, но ее конечное значение считается одним из выходов смарт-контракта. Например, примитив SENDMSG, специфичный для блокчейна TON, просто вставляет сообщение в список, хранящийся в выходных действиях.

- `c7` — содержит корень *временных данных.* Это *Tuple*, инициализированный ссылкой на пустой tuple перед вызовом смарт-контракта и отбрасываемый после его окончания.[4]

В будущем может быть определено больше контрольных регистров для конкретных целей TON Blockchain или языка программирования высокого уровня, если это необходимо.

1.4 Общее состояние TVM (SCCCG)

Общее состояние TVM состоит из следующих компонентов:

- *Стек* (см. 1.1) — содержит ноль или более *значений* (см. 1.1.1), каждое из которых относится к одному из *типов значений* , перечисленных в 1.1.3.
- *Управляющие регистры c0–c15* — содержат некоторые специфические значения, описанные в 1.3.2. (В текущей версии используется только семь контрольных регистров.)
- *Текущее продолжение cc* — содержит текущее продолжение (т.е. код, который обычно выполняется после завершения текущего примитива). Этот компонент аналогичен регистру указателя инструкций (ip) в других архитектурах.
- *Текущая кодовая страница cp* — специальное знаковое 16-битное целое значение, которое выбирает способ декодирования следующего опкода TVM. Например, будущие версии TVM могут использовать другие кодовые страницы для добавления новых опкодов с сохранением обратной совместимости.
- *Газовые лимиты газа* — содержит четыре знаковых 64-битных целых числа: текущий газовый предел g_l, *максимальный газовый предел* g_m, *оставшийся газ* g_r *и газовый кредит* g_c. *Всегда* $0 \le g_l \le g_m$,

[4] В контексте TON Blockchain `c7` инициализируется одноэлементным *кортежем*, единственным компонентом которого является *кортеж,* содержащий специфические для блокчейна данные. Смарт-контракт может свободно модифицировать `c7` для хранения своих временных данных при условии, что первый компонент этого *кортежа* остается нетронутым.

$g_c \geq 0$ и $g_r \leq g_l + g_c$; g_c обычно инициализируется нулем, g_r инициализируется $g_l + g_c$ и постепенно уменьшается по мере запуска TVM. Когда g_r становится отрицательным или если конечное значение g_r меньше g_c, возникает исключение из газа.

Обратите внимание, что не существует «стека возврата», содержащего обратные адреса всех ранее вызываемых, но незавершенных функций. Вместо этого используется только регистр управления `c0`. Причина этого будет объяснена позже в 4.1.9.

Также обратите внимание, что нет регистров общего назначения, потому что TVM является стековой машиной (см. 1.1). Таким образом, приведенный выше список, который можно обобщить как «стек, управление, продолжение, кодовая страница и газ» (SCCCG), аналогично классическому состоянию машины SECD («стек, среда, управление, дамп»), действительно является *полным* состоянием TVM.[5]

1.5Целочисленная арифметика

Все арифметические примитивы TVM оперируют несколькими аргументами типа *Integer*, взятыми из верхней части стека, и возвращают свои результаты, одного типа, в стек. Напомним, что *Integer* представляет все целочисленные значения в диапазоне $-2^{256} \leq x < 2^{256}$, а дополнительно содержит специальное значение `NaN` («nota-число»).

Если один из результатов не вписывается в поддерживаемый диапазон целых чисел или если один из аргументов является `NaN`, то этот результат или все результаты заменяются `NaN`, и (по умолчанию) создается исключение переполнения целых чисел. Однако специальные «тихие» версии арифметических операций будут просто производить `NaNs` и продолжать идти. Если эти `NaN` в конечном итоге используются в «не совсем» арифметической операции или в неарифметической операции, произойдет целочисленное исключение переполнения.

[5] Строго говоря, существует также текущий *контекст библиотеки*, который состоит из словаря с 256-битными ключами и значениями ячеек, используемых для загрузки ссылочных ячеек библиотеки 3.1.7.

1.5.1. Отсутствие автоматического преобразования целых чисел. Обратите внимание, что *целые числа TVM являются «математическими» целыми числами, а не 257-битными строками, интерпретируемыми по-разному в зависимости от используемого примитива, как это характерно для других конструкций машинного кода. Например, TVM имеет только один примитив умножения* `MUL`, а не два (`MUL` для беззнакового умножения и `IMUL` для знакового умножения), как это происходит, например, в популярной архитектуре x86.

1.5.2. Автоматические проверки переполнения. Обратите внимание, что все арифметические примитивы TVM выполняют проверку результатов переполнения. Если результат не вписывается в *тип Integer*, он заменяется `NaN`, и (обычно) возникает исключение. В частности, результат не является автоматически уменьшенным по модулю 2^{256} или 2^{257}, как это принято для большинства аппаратных архитектур машинного кода.

1.5.3. Пользовательские проверки переполнения. В дополнение к автоматическим проверкам переполнения,

TVM включает в себя пользовательские проверки переполнения, выполняемые примитивами FITS *n* и UFITS *n, где* $1 \le n \le 256$. Эти примитивы проверяют, является ли значение в стеке (верхней части) целым числом x в диапазоне $-2^{n-1} \le x < 2^{n-1}$ или $0 \le x < 2^n$ соответственно, и заменяют значение NaN и (необязательно) генерируют исключение переполнения целых чисел, если это не так. Это значительно упрощает реализацию произвольных *n*-битных целочисленных типов, знаковых или беззнаковых: программист или компилятор должны вставлять соответствующие примитивы FITS или UFITS либо после каждой арифметической операции (что более разумно, но требует больше проверок), либо перед сохранением вычисленных значений и возвращением их из функций. Это важно для смарт-контрактов, где неожиданные целочисленные переполнения являются одним из наиболее распространенных источников ошибок.

1.5.4. Редукция по модулю 2^n. TVM также имеет примитив `MODPOW2` n, который уменьшает целое число в верхней части стека по модулю 2^n, с результатом в диапазоне от 0 до $2^n - 1$.

1.5.5. *Целое число* равно 257-битному, а не 256-битному. Теперь можно понять, почему целое число TVM является (знаковым) 257-битным, а не 256-битным. Причина в том, что это наименьший целочисленный тип, содержащий как знаковые 256-битные целые числа, так и беззнаковые 256-битные целые числа, который не требует автоматического переинтерпретирования одной и той же 256-битной строки в зависимости от используемой операции (см. 1.5.1).

1.5.6. Деление и округление. Наиболее важными примитивами деления являются DIV, MOD и DIVMOD. Все они берут из стека два числа, *x* и *y* (y берется из верхней части стека, а *x* изначально находится под ним), вычисляют частное *q* и остаток *r* деления *x* на *y* (т. е. два целых числа, таких, что *x* = *yq* + *r* и |r| < |y|) и верните либо *q, r, либо оба из них. Если y* равно нулю, то все ожидаемые результаты заменяются NaNs, и (обычно) генерируется целочисленное исключение переполнения.

Реализация деления в TVM несколько отличается от большинства других реализаций в отношении округления. По умолчанию эти примитивы округляются до −∞, что означает, что *q* = bx/yc, а *r* имеет тот же знак, *что и y. (В большинстве традиционных реализаций деления вместо этого используется «округление до нуля», что означает, что r* имеет тот же знак, что и *x.) Помимо этого «округления пола», доступны два других режима округления, называемых «потолочным округлением» (с q* = dx/ye, и *r* и *y*, имеющие противоположные знаки) и «ближайшее округление» (с *q* = bx/y + 1/2c и |r| ≤ |y|/2). Эти режимы округления выбираются с помощью других примитивов деления, с буквами C и R, добавленными к их мнемонике. Например, DIVMODR вычисляет как частное, так и остаток, используя округление до ближайшего целого числа.

1.5.7. Комбинированные операции умножения-деления, умножения-сдвига и сдвига-деления. Для упрощения реализации арифметики с фиксированной точкой TVM поддерживает комбинированные операции умножения-деления, умножения-сдвига и сдвиг-деления с промежуточным произведением двойной длины (т.е. 514-бит). Например, `MULDIVMODR` принимает три целочисленных аргумента из стека, a, b и c, сначала вычисляет ab, используя 514-разрядный промежуточный результат, а затем делит ab на c с помощью округления до ближайшего целого числа. Если c равно нулю или если частное не помещается в *Integer*,

возвращаются либо два `NaNs`, либо генерируется целочисленное исключение переполнения, в зависимости от того, использовалась ли тихая версия операции. В противном случае и частное, и остаток выталкиваются в стек.

2 Стек

Эта глава содержит общее обсуждение и сравнение регистровых и стековых машин, расширенное далее в Приложении C, и описывает два основных класса примитивов манипулирования стеком, используемых TVM: *базовый* и *составной примитивы манипулирования стеком*. Также приводится неформальное объяснение их достаточности для всех переупорядочиваний стека, необходимых для правильного вызова других примитивов и определяемых пользователем функций. Наконец, проблема эффективной реализации примитивов манипулирования стеком TVM обсуждается в 2.3.

2.1 Соглашения о вызовах стека

Стековая машина, такая как TVM, использует стек — и особенно значения в верхней части стека — для передачи аргументов в вызываемые функции и примитивы (например, встроенные арифметические операции) и получения их результатов. В этом разделе обсуждаются соглашения о вызовах стека TVM, вводится некоторая нотация и сравниваются соглашения о вызовах стека TVM с соглашениями определенных регистровых машин.

2.1.1. Обозначение "стековых регистров". Напомним, что стековая машина, в отличие от более обычной регистровой машины, не имеет регистров общего назначения. Тем не менее, можно рассматривать значения в верхней части стека как своего рода «регистры стека».

Обозначаем `s0` или $\mathtt{s}(0)$ значение в верхней части стека, `s1` или $\mathtt{s}(1)$ значение непосредственно под ним и так далее. Общее количество значений в стеке называется его *глубиной*. Если глубина стека равна n, то $\mathtt{s}(0)$, $\mathtt{s}(1)$, ..., $\mathtt{s}(n-1)$ четко определены, в то время как $\mathtt{s}(n)$ и все последующие $\mathtt{s}(i)$ с $i \geq n$ — нет. Любая попытка использовать $\mathtt{s}(i)$ с i n должна привести к исключению недолива стека.

Компилятор или программист-человек в «коде TVM» будет использовать эти «регистры стека» для хранения всех объявленных переменных и промежуточных значений, аналогично тому, как регистры общего назначения используются на регистровой машине.

2.1.2. Толкающие и лопающие значения. Когда значение x *выталкивается* в стек глубины n, оно становится новым `s0`; в то же время старое `s0` становится новым `s1`, старое `s1` — новым `s2` и так далее. Глубина результирующего стека равна $n + 1$.

Аналогичным образом, когда значение x *выскакивает* из стека глубины $n \geq 1$, это старое значение `s0` (то есть старое значение в верхней части стека). После этого он удаляется из стека, и старый `s1` становится новым `s0` (новое значение в верхней части стека), старый `s2` становится новым `s1` и так далее. Глубина результирующего стека равна $n - 1$.

Если изначально $n = 0$, то стек *пуст*, и из него нельзя выскочить значение. Если примитив пытается вытащить значение из пустого стека, возникает исключение *переполнения стека.*

2.1.3. Обозначение для гипотетических регистров общего назначения. Чтобы сравнить стековые машины с достаточно общими регистровыми машинами, мы обозначим регистры общего назначения регистровой машины r0, `r1` и т. д. или $\texttt{r}(0)$, $\texttt{r}(1)$, ..., $\texttt{r}(n - 1)$, где n — общее число регистров. Когда нам нужно конкретное значение n, мы будем использовать $n = 16$, соответствующее очень популярной архитектуре x86-64.

2.1.4. Регистр верхней части стека `s0` против регистра аккумулятора `r0`. Некоторые архитектуры регистровых машин требуют, чтобы один из аргументов для большинства арифметических и логических операций находился в специальном регистре, называемом *аккумулятором*. В нашем сравнении мы предположим, что аккумулятор является регистром общего назначения `r0`; в противном случае мы могли бы просто перенумеровать регистры. В этом отношении аккумулятор чем-то похож на «регистр» s0 верхнего стека стековой машины, потому что практически все операции стековой машины используют `s0` в качестве одного из своих аргументов и возвращают свой результат как `s0`.

2.1.5. Регистрация соглашений о вызовах. При компиляции для регистровой машины высокоуровневые языковые функции обычно получают свои аргументы в определенных регистрах в предопределенном порядке. Если аргументов слишком много, эти функции забирают остаток из стека (да, регистровая машина обычно тоже имеет стек!). Однако

некоторые соглашения о вызове регистров вообще не передают аргументов в регистрах и используют только стек (например, исходные соглашения о вызове, используемые в реализациях Pascal и C, хотя современные реализации C также используют некоторые регистры).

Для простоты предположим, что аргументы функции до $m \leq n$ передаются в регистрах, и что эти регистры `r0`, `r1`, ..., `r`$(m - 1)$, в таком порядке (если используются какие-то другие регистры, мы можем просто перенумеровать их).[6]

2.1.6. Порядок аргументов функции. Если для функции или примитива требуются m аргументов x_1, ..., x_m, они выталкиваются вызывающим объектом в стек в том же порядке, начиная с x_1. Поэтому при вызове функции или примитива ее первый аргумент x_1 находится в `s`$(m - 1)$, второй аргумент x_2 — в `s`$(m - 2)$ и так далее. Последний аргумент x_m находится в `s0` (т.е. в верхней части стека). Именно вызываемая функция или примитив несет ответственность за удаление своих аргументов из стека.

В этом отношении соглашения о вызовах стека TVM, соблюдаемые, по крайней мере, примитивами TMV, совпадают с соглашениями Паскаля и Форта и противоположны соглашениям C (в которых аргументы помещаются в стек в обратном порядке и удаляются вызывающим абонентом после восстановления контроля, а не вызываемым).

Конечно, реализация языка высокого уровня для TVM может выбрать некоторые другие соглашения о вызовах для своих функций, отличные от соглашений по умолчанию. Это может быть полезно для определенных функций, например, если общее количество аргументов зависит от значения первого аргумента, как это происходит для «вариадных функций», таких как `scanf` и `printf`. В таких случаях первый один или

[6] Наше включение r0 здесь создает незначительный конфликт с нашим предположением, что накопительный регистр, если он присутствует, также равен `r0`; для простоты мы решим эту проблему, предположив, что первый аргумент функции передается в аккумуляторе.

несколько аргументов лучше передать в верхней части стека, а не где-то в каком-то неизвестном месте глубоко в стеке.

2.1.7. Аргументы арифметическим примитивам на регистровых машинах. На стековом компьютере встроенные арифметические примитивы (такие как `ADD` или `DIVMOD`) следуют тем же соглашениям о вызовах, что и определяемые пользователем функции. В этом отношении определяемые пользователем функции (например, функция, вычисляющая квадратный корень из числа) могут рассматриваться как «расширения» или «пользовательские обновления» стековой машины. Это одно из самых явных преимуществ стековых машин (и языков программирования стека, таких как Forth) по сравнению с регистровыми машинами.

Напротив, арифметические инструкции (встроенные операции) на регистровых машинах обычно получают свои параметры из регистров общего назначения, закодированных в полном опкоде. Таким образом, двоичная операция, такая как `SUB`, требует двух аргументов, $\mathtt{r}(i)$ и $\mathtt{r}(j)$,причем i и j задаются инструкцией. Также должен быть указан регистр $\mathtt{r}(k)$ для хранения результата. Арифметические операции могут принимать несколько возможных форм, в зависимости от того, разрешено ли i, j и k принимать произвольные значения:

- Трехадресная форма — позволяет программисту произвольно выбирать не только два исходных регистра r(i) и r(j), но и отдельный регистр назначения r(k). Эта форма характерна для большинства RISC-процессоров, а также для наборов инструкций XMM и AVX SIMD в архитектуре x86-64.
- Двухадресная форма — использует один из двух операндных регистров (обычно r(i)) для хранения результата операции, так что $k = i$ никогда не указывается явно. Внутри инструкции кодируются только i и j. Это наиболее распространенная форма арифметических операций на регистровых машинах, и довольно популярна на микропроцессорах (включая семейство x86).
- Одноадресная форма — всегда берет один из аргументов из аккумулятора r0, и сохраняет результат в r0; тогда $i = k = 0$, и только j должен быть указан инструкцией. Эта форма используется некоторыми более простыми микропроцессорами (такими как Intel 8080).

Обратите внимание, что эта гибкость доступна только для встроенных операций, но не для пользовательских функций. В этом отношении регистровые машины не так легко «модернизируются», как стековые машины.[7]

2.1.8. Возвращаемые значения функций. В стековых машинах, таких как TVM, когда функции или примитиву необходимо вернуть результирующее значение, он просто отправляет его в стек (из которого уже удалены все аргументы функции). Таким образом, вызывающий абонент сможет получить доступ к результирующему значению через верхнюю часть стека "register" `s0`.

Это полностью соответствует соглашениям вызова Forth, но немного отличается от соглашений вызова Pascal и C, где для `регистр аккумулятора r0`. возвращаемого значения обычно используется

2.1.9. Возврат нескольких значений. Некоторые функции могут захотеть вернуть несколько значений y_1, ..., y_k, причем k не обязательно равно единице. В этих случаях возвращаемые значения k помещаются в стек в их естественном порядке, начиная с y_1.

Например, примитив DIVMOD «разделить на остаток» должен возвращать два значения: частное q и остаток r. Поэтому `DIVMOD` выталкивает q и r в стек в указанном порядке, так что частное доступно после этого при `s1`, а оставшееся — при `s0`. Чистый эффект `DIVMOD` состоит в том, чтобы разделить исходное значение `s1` на исходное значение `s0` и вернуть частное в `s1` и остаток в `s0`. В этом конкретном случае глубина стека и значения всех остальных «регистров стека» остаются неизменными, поскольку `DIVMOD` принимает два аргумента и возвращает два результата. В общем, значения других «регистров стека», которые лежат в стеке ниже переданных аргументов и возвращаемых значений, смещаются в соответствии с изменением глубины стека.

[7] Например, если написать функцию для извлечения квадратных корней, эта функция всегда будет принимать свой аргумент и возвращать свой результат в одних и тех же регистрах, в отличие от гипотетической встроенной инструкции квадратного корня, которая может позволить программисту произвольно выбирать исходный и конечный регистры. Поэтому определяемая пользователем функция значительно менее гибкая, чем встроенная инструкция на регистровой машине.

В принципе, некоторые примитивы и определяемые пользователем функции могут возвращать переменное число результирующих значений. В этом отношении применимы приведенные выше замечания о вариадных функциях (см. 2.1.6): общее число результирующих значений и их типы должны определяться значениями в верхней части стека. (Например, можно отправить возвращаемые значения y_1, ..., y_k, а затем введите их общее число k как целое число. Затем вызывающая сторона определяет общее количество возвращаемых значений, проверяя `s0`.)

В этом отношении TVM, опять же, добросовестно соблюдает конвенции о вызове Форта.

2.1.10. Стековая нотация. Когда стек глубины n содержит значения z_1, ..., z_n, в таком порядке, с z_1 самым глубоким элементом и z_n в верхней части стека, содержимое стека часто представлено списком z_1 z_2 ... z_n, в указанном порядке. Когда примитив преобразует исходное состояние стека S' в новое состояние S'', это часто записывается как $S' - S''$; это так называемая *нотация стека*. Например, действие примитива деления `DIV` можно описать $S\ x\ y - S\ \lfloor x/y \rfloor$, молчаливо предполагая, что все остальные значения глубже в стеке остаются, где S — любой список значений. Обычно это сокращается как $x\ y - \lfloor x/y \rfloor$ intact.

В качестве альтернативы можно описать DIV как примитив, который работает на стеке S' глубины $n \geq 2$, делит s1 на s0 и возвращает закругленный коэффициент пола как s0 нового стека S'' глубины $n - 1$. Новое значение s(i) равно старому значению s(i + 1) для $1 \leq i < n - 1$. Эти описания эквивалентны, но говорят, что DIV преобразует $x\ y$ в $\lfloor x/y \rfloor$, или ... x y в ... $\lfloor x/y \rfloor$, более лаконично.

Нотация стека широко используется в Приложении A, где перечислены все определенные в настоящее время примитивы TVM.

2.1.11. Явное определение количества аргументов функции. Стековые машины обычно передают текущий стек в полном объеме вызывающему примитиву или функции. Этот примитив или функция обращается только к нескольким значениям в верхней части стека, которые представляют его аргументы, и передает возвращают значения на их место, по соглашению оставляя все более глубокие значения нетронутыми. Затем результирующий стек, опять же в полном объеме, возвращается вызывающему объекту.

Большинство примитивов TVM ведут себя таким образом, и мы ожидаем, что большинство пользовательских функций будут реализованы в соответствии с такими соглашениями. Однако TVM предоставляет механизмы, указывающие, сколько аргументов должно быть передано в вызываемую функцию (см. 4.1.10). При использовании этих механизмов указанное количество значений перемещается из стека вызывающей стороны в (обычно изначально пустой) стек вызываемой функции, в то время как более глубокие значения остаются в стеке вызывающей стороны и недоступны для вызываемого объекта. Вызывающая сторона также может указать, сколько возвращаемых значений она ожидает от вызываемой функции.

Такие механизмы проверки аргументов могут быть полезны, например, для библиотечной функции, которая вызывает предоставляемые пользователем функции, переданные ей в качестве аргументов.

2.2 Примитивы манипулирования стеком

Стековая машина, такая как TVM, использует множество примитивов манипулирования стеком для переупорядочивания аргументов в другие примитивы и определяемые пользователем функции, чтобы они располагались в верхней части стека в правильном порядке. В этом разделе обсуждается, какие примитивы манипулирования стеком необходимы и достаточны для достижения этой цели, и какие из них используются TVM. Некоторые примеры кода, использующего эти примитивы, можно найти в Приложении C.

2.2.1. Базовые примитивы манипулирования стеком. Наиболее важными примитивами манипулирования стеком, используемыми TVM, являются следующие:

- *Top-of-stack exchange operation: XCHG s0,s(i) или XCHG s(i) — обменные значения s0 и s(i). Когда i* = 1, операция XCHG s1 традиционно обозначается SWAP. Когда *i* = 0, это NOP (операция, которая ничего не делает, по крайней мере, если стек не пуст).
- *Arbitrary exchange operation: XCHG s(i),s(j) — обменные значения s(i) и s(j). Обратите внимание, что эта операция не является строго*

необходимой, поскольку она может быть смоделирована тремя биржами верхнего стека: XCHG s(i); XCHG s(j); XCHG s(i). Однако полезно иметь произвольные обмены в качестве примитивов, потому что они требуются довольно часто.

- *Операция push: PUSH s(i) — помещает копию (старого) значения s(i) в стек. Традиционно PUSH s0 также обозначается DUP (он дублирует значение в верхней части стека), а PUSH s1 — OVER.*
- *Операция Pop: POP s(i) — удаляет верхнее значение стека и помещает его в (новый) s*(*i* − 1) или старый s(i). Традиционно POP s0 также обозначается DROP (он просто роняет верхнее значение стека), а POP s1 — NIP.

Некоторые другие «несистематические» операции манипулирования стеком также могут быть определены (например, `ROT`, с обозначением *стека a b c – b c a*). Хотя такие операции определены в языках стека, таких как Forth (где также присутствуют `DUP`, `DROP`, `OVER`, `NIP` и `SWAP`), они не являются строго необходимыми, поскольку *перечисленных выше базовых примитивов управления стеком* достаточно для переупорядочивания регистров стека, чтобы обеспечить правильный вызов любых арифметических примитивов и пользовательских функций.

2.2.2. Достаточно базовых примитивов манипулирования стеком. Компилятор или программист TVM-кода может использовать базовые примитивы стека следующим образом.

Предположим, что вызываемая функция или примитив должны быть переданы, скажем, трем аргументам x, y и z, которые в настоящее время находятся в регистрах `стека` s(i),s(j) и s(k). В этом случае компилятор (или программист) может выдать операцию `PUSH` s(i) (если после вызова этого примитива требуется копия x) или `XCHG` s(i) (если она не понадобится впоследствии), чтобы поместить первый аргумент x в верхнюю часть стека. Затем компилятор (или программист) может использовать `PUSH` s(j') или `XCHG` s(j'),где $j' = j$ или $j + 1$, чтобы поместить y в новую верхнюю часть стека.[8]

[8] Конечно, если использовать второй вариант, это разрушит первоначальное расположение x в верхней части стека. В этом случае следует либо выдать `SWAP` перед

Действуя таким образом, мы видим, что мы можем поместить исходные значения x, y и z — или их копии, если это необходимо — в места `s2`, `s1` и `s0`, используя последовательность push-операций и обменных операций (см. 2.2.4 и 2.2.5 для более подробного объяснения). Чтобы сгенерировать эту последовательность, компилятору необходимо знать только три значения i, j и k, описывающие старые расположения рассматриваемых переменных или временных значений, и некоторые флаги, описывающие, понадобится ли каждое значение впоследствии или необходимо только для этого примитива или вызова функции. Расположение других переменных и временных значений будет затронуто в процессе, но компилятор (или программист-человек) может легко отслеживать их новые местоположения.

Аналогичным образом, если результаты, возвращаемые функцией, необходимо отбросить или переместить в другие регистры стека, подходящая последовательность операций обмена и pop выполнит эту работу. В типичном случае одного возвращаемого значения в `s0` это достигается либо операцией `XCHG s(`i`)`, либо `операцией POP s(`i`)` (в большинстве случаев `DROP`).[9]

Перестановка результирующего значения или значений перед возвратом из функции по существу является той же проблемой, что и упорядочение аргументов для вызова функции, и достигается аналогичным образом.

2.2.3. Примитивы манипуляции составным стеком. Чтобы повысить плотность кода TVM и упростить разработку компиляторов, могут быть определены составные примитивы манипулирования стеком, каждый из которых объединяет до четырех базовых примитивов exchange и push или exchange и pop. Такие составные стековые операции могут включать, например:

`XCHG s(`$j0$`)`,либо заменить предыдущую операцию `XCHG s(`i`)` на `XCHG s1, s(`i`)`, чтобы x обменивался на `s1` с самого начала.

[9] Обратите внимание, что наиболее распространенная `операция XCHG s(`i`)` здесь не требуется, если мы не настаиваем на сохранении одного и того же временного значения или переменной всегда в одном и том же месте стека, а скорее отслеживаем его последующие местоположения. Мы переместим его в другое место, подготавливая аргументы к следующему вызову примитива или функции.

- `XCHG2 s(i),s(j)` — эквивалент `XCHG s1,s(i); XCHG s(j)`.
- `PUSH2 s(i),s(j)` — эквивалент `PUSH s(i); PUSH s(j + 1)`.
- `XCPU s(i),s(j)` — эквивалент `XCHG s(i); PUSH s(j)`.
- `PUXC s(i),s(j)` — эквивалентно `PUSH s(i); SWAP; XCHG s(j + 1)`. Когда $j \neq i$ и $j \neq 0$, это также эквивалентно `XCHG s(j); PUSH s(i); SWAP`.
- `XCHG3 s(i),s(j),s(k)`— эквивалент `XCHG s2,s(i); XCHG s1,s(j); XCHG s(k)`.
- `PUSH3 s(i),s(j),s(k)`— эквивалент PUSH `s(i); PUSH s(j +1); PUSH s(k + 2)`.

Конечно, такие операции имеют смысл только в том случае, если они допускают более компактное кодирование, чем эквивалентная последовательность базовых операций. Например, если все обмены верхнего стека, обмены `XCHG s1,s(i)` и push- и pop-операции допускают однобайтовые кодировки, единственными операциями составного стека, предложенными выше, которые могут заслуживать включения в набор примитивов манипулирования стеком, являются `PUXC`, `XCHG3` и `PUSH3`.

Эти составные стековые операции существенно дополняют другие примитивы (инструкции) в коде «истинными» местоположениями их операндов, что примерно аналогично тому, что происходит с двухадресным или трехадресным регистровым машинным кодом. Однако вместо того, чтобы кодировать эти места внутри опкода арифметики или другой инструкции, как это принято для регистровых машин, мы указываем эти местоположения в предыдущей операции манипулирования составным стеком. Как уже описано в 2.1.7, преимущество такого подхода заключается в том, что определяемые пользователем функции (или редко используемые конкретные примитивы, добавленные в будущую версию TVM), также могут извлечь из него выгоду (см. C.3 для более подробного обсуждения с примерами).

2.2.4. Мнемоника операций с компаундным штабелем. Мнемоника операций с составным стеком, некоторые примеры которых приведены в подразделе 2.2.3, создается следующим образом.

В $\gamma \geq 2$ формальных аргумента s(i_1), ..., s(i_γ) к такой операции *O* представляют значения в исходном стеке, которые в конечном итоге окажутся в s($\gamma - 1$), ..., s0 представляет собой последовательность после выполнения этой составной операции, по крайней мере, если все i_v, $1 \leq v \leq \gamma$, различны и, по меньшей мере, γ. Сама мнемоника операции *O* представляет собой последовательность γ двухбуквенных строк PU и XC, причем PU означает, что соответствующий аргумент должен быть PUshed (т. е. должна быть создана копия), а XC означает, что значение должно быть eXChanged (т. Е. Никакая другая копия исходного значения не создается). Последовательности нескольких строк PU или XC могут быть сокращены до одного PU или XC с последующим количеством копий. (Например, мы пишем `PUXC2PU` вместо `PUXCXCPU`.)

В качестве исключения, если мнемоника будет состоять только из `PU` или только `строк XC`, так что составная операция эквивалентна последовательности *m* `PUSHes` или `eXCHanGes`, `PUSHm` или `XCHGm` вместо `PUm` или `XCm` используется нотация.

2.2.5. Семантика операций составного стека. Каждая γ-ая составная операция *O* $s(i_1),\ldots,s(i_\gamma)$ переводится в эквивалентную последовательность основных операций стека путем индукции следующим образом:

- В качестве основы индукции, если $\gamma = 0$, единственная нулевая операция стека соединения соответствует пустой последовательности основных операций стека.
- Эквивалентно, мы могли бы начать индукцию с $\gamma = 1$. Тогда `PU` s(i) соответствует последовательности, состоящей из одной базовой операции `PUSH` s(i), а `XC` s(i) соответствует одноэлементной последовательности, состоящей из XCHG s(i).
- Для $\gamma \geq 1$ (или для $\gamma \geq 2$, если мы используем $\gamma = 1$ в качестве основания индукции) есть два подкафайла:

1. Os(i_1),...,s(i_γ), с O = `XC`*O'*, где O' — составная операция арности $\gamma-1$ (т.е. мнемоника O' состоит из $\gamma-1$ струн `XC` и `PU`). Пусть α будет общее количество `PUshes` в O, а β будет количество `eXChanges`, так что $\alpha+\beta = \gamma$. Затем исходная операция переводится в `XCHG` s($\beta -1$),s(i_1), за которой следует трансляция *O's(i_2),...,s(i_γ),* определяемая

индукционной гипотезой.

2. $Os(i_1),\ldots,\mathtt{s}(i_\gamma)$, с O = PUO', где O' — составная операция арити γ−1. Затем исходная операция переводится в PUSH s(i_1); XCHG s(β), за которым следует трансляция $O's(i_2+1),\ldots,\mathtt{s}(i_\gamma+1)$, определяемая индукционной гипотезой.[10]

2.2.6. Инструкции по манипулированию стеком полиморфны. Обратите внимание, что инструкции по манипулированию стеком являются чуть ли не единственными «полиморфными» примитивами в TVM, то есть они работают со значениями произвольных типов (включая типы значений, которые появятся только в будущих версиях TVM). Например, SWAP всегда обменивается двумя верхними значениями стека, даже если одно из них является целым числом, а другое — ячейкой. Почти все другие инструкции, особенно инструкции по обработке данных (включая арифметические инструкции), требуют, чтобы каждый из их аргументов имел какой-то фиксированный тип (возможно, разный для разных аргументов).

2.3 Эффективность примитивов манипулирования стеком

Примитивы манипулирования стеком, используемые стековой машиной, такой как TVM, должны быть реализованы очень эффективно, поскольку они составляют более половины всех инструкций, используемых в типичной программе. Фактически, TVM выполняет все эти инструкции за (маленькое) постоянное время, независимо от используемых значений (даже если они представляют собой очень большие целые числа или очень большие деревья ячеек).

2.3.1. Реализация примитивов манипулирования стеком: использование ссылок на операции вместо объектов. Эффективность реализации TVM примитивов манипулирования стеком обусловлена тем, что типичная реализация TVM сохраняет в стеке не сами объекты значений, а только ссылки (указатели) на такие объекты. Поэтому инструкция SWAP должна

[10] Альтернативный, возможно, лучший перевод PUO's(i_1),…,s(i_γ) состоит из перевода $O's(i_2),\ldots,\mathtt{s}(i_\gamma)$,за которым следует PUSH s(i_1 + α - 1); XCHG s(γ-1).

обмениваться только ссылками в `s0` и `s1`, а не фактическими объектами, на которые они ссылаются.

2.3.2. Эффективная реализация инструкций `DUP` и `PUSH` с помощью copy-on-write. Кроме того, инструкция `DUP` (или, в более общем смысле, `PUSH` `s`(i)), которая, по-видимому, делает копию потенциально большого объекта, также работает в небольшое постоянное время, потому что она использует технику замедленного копирования копирования при записи: она копирует только ссылку, а не сам объект, но увеличивает «счетчик ссылок» внутри объекта, тем самым разделяя объект между двумя ссылками. При обнаружении попытки изменить объект со счетчиком ссылок, превышающим единицу, сначала делается отдельная копия рассматриваемого объекта (что влечет за собой определенный «штраф за неуникальность» или «штраф за копирование» за инструкцию по манипулированию данными, которая вызвала создание новой копии).

2.3.3. Сбор мусора и подсчет ссылок. Когда счетчик ссылок объекта TVM становится нулевым (например, потому что последняя ссылка на такой объект была поглощена операцией `DROP` или арифметической инструкцией), он немедленно освобождается. Поскольку циклические ссылки невозможны в структурах данных TVM, этот метод подсчета ссылок обеспечивает быстрый и удобный способ освобождения неиспользуемых объектов, заменяя медленные и непредсказуемые сборщики мусора.

2.3.4. Прозрачность реализации: Значения стека являются «значениями», а не «ссылками». Независимо от только что рассмотренных деталей реализации, все значения стека на самом деле являются «значениями», а не «ссылками», с точки зрения программиста TVM, аналогично значениям всех типов в функциональных языках программирования. Любая попытка изменить существующий объект, на который ссылаются любые другие объекты или расположения стека, приведет к прозрачной замене этого объекта его идеальной копией до фактического выполнения изменения.

Другими словами, программист всегда должен действовать так, как если бы самими объектами непосредственно манипулировали стековые, арифметические и другие примитивы преобразования данных, и рассматривать предыдущее обсуждение только как объяснение высокой эффективности примитивов манипулирования стеком.

2.3.5. Отсутствие круговых ссылок. Можно попытаться создать круговую ссылку между двумя ячейками, A и B, следующим образом: сначала создайте A и запишите в него некоторые данные; затем создайте B и запишите в него некоторые данные вместе со ссылкой на ранее построенную ячейку A; наконец, добавьте ссылку на B в A. Хотя может показаться, что после этой последовательности операций мы получаем ячейку A, которая относится к B, которая, в свою очередь, относится к A, это не так. Фактически, мы получаем новую ячейку A', которая содержит копию данных, первоначально хранящихся в ячейке A , вместе со ссылкой на ячейку B, которая содержит ссылку на (оригинальную) ячейку A.

Таким образом, прозрачный механизм копирования при записи и парадигма «все является ценностью» позволяют нам создавать новые ячейки, используя только ранее построенные ячейки, тем самым запрещая появление круговых ссылок. Это свойство также применимо ко всем другим структурам данных: например, отсутствие циклических ссылок позволяет TVM использовать подсчет ссылок для немедленного освобождения неиспользуемой памяти вместо того, чтобы полагаться на сборщики мусора. Точно так же это свойство имеет решающее значение для хранения данных в блокчейне TON.

3 Ячейки, память и постоянное хранилище

В этой главе кратко описываются ячейки TVM, используемые для представления всех структур данных внутри памяти TVM и ее постоянного хранилища, а также основные операции, используемые для создания ячеек, записи (или сериализации) данных в них и чтения (или десериализации) данных из них.

3.1 Общие положения о клетках

В этом разделе представлена классификация и общие описания типов клеток.

3.1.1. Память TVM и постоянное хранилище состоят из ячеек. Напомним, что память TVM и постоянное хранилище состоят из *ячеек (TVM*). Каждая ячейка содержит до 1023 бит данных и до четырех ссылок на другие ячейки. [11] Циркулярные ссылки запрещены и не могут быть созданы с помощью TVM (см. 2.3.5). Таким образом, все клетки, хранящиеся в памяти TVM и постоянном хранилище, представляют собой направленный ациклический граф (DAG).

3.1.2. Обычные и экзотические клетки. Помимо данных и ссылок, ячейка имеет *тип ячейки*, закодированный целым числом $-1 \ldots 255$. Клетка типа -1 называется *обычной*, такие клетки не требуют какой-либо специальной обработки. Клетки других типов называются *экзотическими* и могут быть *загружены* — автоматически заменены другими клетками при попытке их десериализации (т. е. преобразовать их в *срез* с помощью `инструкции CTOS`). Они также могут демонстрировать нетривиальное поведение при вычислении их хэшей.

Наиболее распространенным использованием экзотических клеток является представление некоторых других клеток, например, клеток,

[11] С точки зрения низкоуровневых операций с ячейками, эти биты данных и ссылки на ячейки не смешиваются. Другими словами, (обычная) ячейка по существу представляет собой пару, состоящую из списка до 1023 бит и списка до четырех ссылок на ячейки, без предписания порядка, в котором ссылки и биты данных должны быть десериализованы, хотя схемы TL-B, по-видимому, предполагают такой порядок.

присутствующих во внешней библиотеке или обрезанных из исходного дерева клеток при создании доказательства Меркла.

Тип экзотической ячейки хранится в виде первых восьми бит ее данных. Если экзотическая ячейка содержит менее восьми бит данных, она недопустима.

3.1.3. Уровень ячейки. Каждая ячейка c имеет другой атрибут $\mathrm{Lvl}(c)$, называемый ее *(de Brujn) уровнем*, который в настоящее время принимает целочисленные значения в диапазоне 0...3.
Уровень обычной клетки всегда равен максимуму уровней всех ее детей c_i:

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

для обычной ячейки c , содержащей r ссылки на ячейки c_1, ..., c_r. Если $r = 0$, $\mathrm{Lvl}(c) = 0$. Экзотические клетки могут иметь разные правила для установки своего уровня.

Уровень клетки влияет на количество *более высоких хэшей,* которые она имеет. Точнее, ячейка уровня l имеет l более высокие хэши $\mathrm{Hash}_1(c)$, ..., $\mathrm{Hash}_l(c)$ в дополнение к своему представлению хэша $\mathrm{Hash}(c) = \mathrm{Hash}_\infty(c)$. Ячейки ненулевого уровня появляются внутри *доказательств Меркла* и *обновлений Меркла*, после обрезки некоторых ветвей дерева ячеек, представляющих значение абстрактного типа данных.

3.1.4. Стандартное представление ячеек. Когда ячейка должна быть передана по сетевому протоколу или сохранена в дисковом файле, она должна быть *сериализована*. Стандартное представление $\mathrm{CellRepr}(c) = \mathrm{CellRepr}_\infty(c)$ ячейки c в виде последовательности октета (байта) строится следующим образом:

1. Сначала сериализуются два дескрипторных байта d_1 и d_2. Байт d_1 равен $r+8s+32l$, *где* $0 \le r \le 4$ — количество клеточных ссылок, содержащихся в ячейке, $0 \le l \le 3$ — уровень ячейки, а $0 \le s \le 1$ — 1 для экзотических клеток и 0 для обычных клеток. Байт d_2 равен $\lfloor b/8 \rfloor + \lceil b/8 \rceil$, где $0 \le b \le 1023$ — количество битов данных в c.

2. Затем биты данных сериализуются как $\lceil b/8 \rceil$ 8-битные октеты (байты). Если b не кратно восьми, к битам данных добавляется

двоичное число 1 и до шести двоичных 0. После этого данные разбиваются на восьмибитные группы ⌈b/8⌉, и каждая группа интерпретируется как беззнаковое целое число биг-эндиана 0...255 и сохраняется в октете.

A. . Наконец, каждая из ссылок на ячейки r представлена 32 байтами, содержащими 256-битное *представление хэша* Hash(c_i), описанного ниже в 3.1.5, упомянутой *ячейки* c_i .

Таким образом, получается 2 + ⌈b/8⌉ + 32r байт CellRepr(c).

3.1.5. Хэш представления ячейки. 256-битный *хэш представления* или просто *хэш* Hash(c) ячейки c рекурсивно определяется как sha256 стандартного представления ячейки c:

$$\text{Hash}(c) := \text{sha256}(\text{CellRepr}(c)) \quad (2)$$

Обратите внимание, что циклические ссылки на ячейки не допускаются и не могут быть созданы с помощью TVM (см. 2.3.5), поэтому эта рекурсия всегда заканчивается, и хэш представления любой ячейки четко определен.

3.1.6. Чем выше хэши ячейки. Напомним, что клетка c уровня l имеет l более высокие хэши $\text{Hash}_i(c)$, $1 \le i \le l$, а также. *Экзотические клетки имеют свои собственные правила вычисления своих высших хэшей.* Высшие хэши Hash_i(c) обычной ячейки с вычисляются аналогично хэшу представления, но с использованием более высоких хэшей $\text{Hash}_i(c_j)$ ее дочерних c_j вместо их репрезентативных хэшей $\text{Hash}(c_j)$. По соглашению мы устанавливаем Hash_∞(c) := Hash(c) и Hash_i(c) := Hash_∞(c) = Hash(c) для всех $i > l$.[12]

[12] С теоретической точки зрения мы могли бы сказать, что клетка c имеет бесконечную последовательность хэшей $(\text{Hash}_i(c))_{i\ge1}$, которая в конечном итоге стабилизируется: $\text{Hash}_i(c) \to \text{Hash}_\infty(c)$. Тогда уровень l является просто наибольшим *индексом* i, таким образом, что $\text{Hash}_i(c) \ne \text{Hash}_\infty(c)$.

3.1.7. Виды экзотических клеток. В настоящее время TVM поддерживает следующие типы ячеек:

- Тип −1: *Обычная ячейка* — содержит до 1023 бит данных и до четырех ссылок на ячейки.
- Тип 1: *Обрезанная ветвь ячейки c* — может иметь любой уровень $1 \le l \le 3$. Он содержит ровно 8 + 256l бит данных: сначала 8-битное целое число, равное 1 (представляющее тип ячейки), затем его l более высокие хэши $\mathrm{Hash}_1(c)$, ..., $\mathrm{Hash}_l(c)$. Уровень l обрезанной ячейки ветви можно назвать ее индексом де Брюйна, поскольку он определяет внешнее доказательство Меркла или обновление Меркла, при построении которого ветвь была обрезана. Попытка загрузить обрезанную ячейку ветви обычно приводит к исключению.
- Тип 2: *ссылочная ячейка библиотеки* — всегда имеет уровень 0 и содержит 8+256 бит данных, включая целое число 2 8-разрядного типа и хэш представления Hash(c') ячейки библиотеки, на которую ссылаются. При загрузке ссылочная ячейка библиотеки может быть прозрачно заменена ячейкой, на которую она ссылается, если она найдена в текущем *контексте библиотеки.*
- Тип 3: *Ячейка доказательства Меркла c* — имеет ровно одну ссылку c_1 и уровень $0 \le l \le 3$, что должно быть на единицу меньше уровня ее единственного дочернего c_1:

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

  8 + 256 бит данных ячейки доказательства Меркла содержат ее 8-битное целое число типа 3, за которым следует $\mathrm{Hash}_1(c_1)$ (предполагается, что оно равно $\mathrm{Hash}(c_1)$, если $\mathrm{Lvl}(c_1) = 0$). Высшие хэши $\mathrm{Hash}_i(c)$ из c вычисляются аналогично более высоким хэшам обычной ячейки, но с $\mathrm{Hash}_{i+1}(c_1)$, используемым вместо $\mathrm{Hash}_i(c_1)$. При загрузке ячейка защиты Меркла заменяется ячейкой c_1.
- Тип 4: *Ячейка обновления Меркла c* — имеет двух детей c_1 и c_2. *Его уровень* $0 \le l \le 3$ задается

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

  Обновление Меркла ведет себя как доказательство Меркла как для c_1, так и для c_2 и содержит 8+256+256 бит данных с $\mathrm{Hash}_1(c_1)$ и $\mathrm{Hash}_1(c_2)$. Однако дополнительным требованием является то, что все обрезанные ячейки ветви *c', которые являются потомками* c_2 *и*

связаны с, также должны быть потомками c_1.[13] При загрузке ячейки обновления Merkle она заменяется ячейкой c_2.

3.1.8. Все значения алгебраических типов данных являются деревьями ячеек. Произвольные значения произвольных алгебраических типов данных (например, всех типов, используемых в функциональных языках программирования) могут быть сериализованы в деревья ячеек (уровня 0), и такие представления используются для представления таких значений в TVM. Механизм копирования при записи (см. 2.3.2) позволяет TVM идентифицировать ячейки, содержащие те же данные и ссылки, и хранить только одну копию таких ячеек. Это фактически превращает дерево ячеек в направленный ациклический граф (с дополнительным свойством, что все его вершины должны быть доступны из отмеченной вершины, называемой «корнем»). Однако это оптимизация хранилища, а не существенное свойство TVM. С точки зрения программиста кода TVM, следует думать о структурах данных TVM как о деревьях ячеек.

3.1.9. TVM-код представляет собой дерево ячеек. Сам tvM-код также представлен деревом ячеек. Действительно, код TVM — это просто значение некоторого сложного алгебраического типа данных, и поэтому он может быть сериализован в дерево ячеек.

Точный способ, которым код TVM (например, код сборки TVM) преобразуется в дерево ячеек, объясняется позже (см. 4.1.4 и 5.2), в разделах, посвященных инструкциям потока управления, продолжениям и кодированию инструкций TVM.

[13] Обрезанная ветвь ячейки c_0 уровня l связана ячейкой Меркла (доказательство или обновление) *c, если на пути от c* есть ячейки Меркла к ее потомку *c'*, включая c, включая c.

3.1.10. Парадигма «Все – мешок клеток». Как описано в [1, 2.5.14], все данные, используемые блокчейном TON, включая сами блоки и состояние блокчейна, могут быть представлены — и представлены — в виде коллекций или «мешков» ячеек. Мы видим, что структура данных TVM (ср. 3.1.8) и кода (ср. 3.1.9) хорошо вписывается в эту парадигму «все — мешок клеток». Таким образом, TVM, естественно, может использоваться для выполнения смарт-контрактов в блокчейне TON, а блокчейн TON может использоваться для хранения кода и постоянных данных этих смарт-контрактов между вызовами TVM. (Конечно, и TVM, и блокчейн TON были разработаны таким образом, чтобы это стало возможным.)

3.2 Инструкции и ячейки для обработки данных

Следующая большая группа инструкций TVM состоит из инструкций по *манипулированию данными*, также известных как *инструкции по манипулированию клетками* или просто *клеточные инструкции*. Они соответствуют инструкциям доступа к памяти других архитектур.

3.2.1. Классы инструкций по клеточной манипуляции. Инструкции ячейки TVM естественным образом подразделяются на два основных класса:

- *Инструкции по созданию ячеекструктурируют новые ячейки из значений, ранее хранившихся в стеке, и ранее или инструкций сериализации*, используемых для построения ячеек.
- *Инструкции по синтаксическому анализу ячеекданные, ранее хранившиеся в ячейках с помощью инструкций по созданию ячеек или инструкций по десериализации*, используемых для извлечения

Кроме того, существуют экзотические клеточные *инструкции,* используемые для создания и проверки экзотических клеток (см. 3.1.2), которые, в частности, используются для представления обрезанных ветвей доказательств Меркла и самих доказательств Меркла.

3.2.2. *Значения Builder* и *Slice*. Инструкции по созданию ячеек обычно работают со *значениями Builder*, которые можно хранить только в стеке (см. 1.1.3). Такие значения представляют собой частично построенные ячейки, для которых могут быть определены быстрые операции по добавлению битовых строк, целых чисел, других ячеек и ссылок на другие ячейки. Аналогичным образом, инструкции синтаксического анализа ячеек

интенсивно используют *значения Slice*, которые представляют собой либо остаток частично проанализированной ячейки, либо значение (подячейку), находящееся внутри такой ячейки и извлеченное из нее инструкцией синтаксического анализа.

3.2.3. Значения *Builder* и *Slice* существуют только как значения стека. Обратите внимание, что *объекты Builder* и *Slice* отображаются только как значения в стеке TVM. Они не могут храниться в «памяти» (то есть деревьях клеток) или «постоянном хранении» (которое также является мешком клеток). В этом смысле объектов Cell гораздо больше, чем объектов *Builder* или *Slice* в среде TVM, но, как это ни парадоксально, программа TVM видит объекты *Builder* и *Slice* в своем стеке чаще, чем *Cells. На самом деле, программа TVM не имеет большого использования для* значений Cell, потому что они неизменяемы и непрозрачны; все примитивы манипулирования клетками требуют, чтобы *значение Cell* было преобразовано в *Builder* или *Slice* сначала, прежде чем его можно будет изменить или проверить.

3.2.4. TVM не имеет отдельного *типа значения Bitstring*. Обратите внимание, что TVM не предлагает отдельного типа значения bitstring. Вместо этого битовые строки представлены *фрагментами, которые вообще не имеют ссылок, но все еще могут содержать до 1023 бит данных.*

3.2.5. Ячейки и клеточные примитивы ориентированы на биты, а не байты. Важным моментом является то, что *TVM рассматривает данные, хранящиеся в ячейках, как последовательности (строки, потоки) из (до 1023) бит, а не из байтов*. Другими словами, TVM — это *бит-ориентированная машина*, а не байт-ориентированная машина. При необходимости приложение может свободно использовать, скажем, 21-битные целочисленные поля внутри записей, сериализованных в ячейки TVM, тем самым используя меньше байтов постоянного хранилища для представления одних и тех же данных.

3.2.6. Таксономия примитивов создания (сериализации) клеток. Примитивы создания ячеек обычно принимают аргумент *Builder* и аргумент, представляющий значение, подлежащее сериализации. Дополнительные аргументы, контролирующие некоторые аспекты

процесса сериализации (например, сколько бит следует использовать для сериализации), также могут быть предоставлены либо в стеке, либо в качестве непосредственного значения внутри инструкции. Результатом примитивного создания ячейки обычно является другой *Builder*, представляющий собой конкатенацию исходного конструктора и сериализацию предоставленного значения.

Поэтому можно предложить классификацию примитивов сериализации клеток по ответам на следующие вопросы:

- Какой тип сериализуемых значений?
- Сколько бит используется для сериализации? Если это переменное число, происходит ли оно из стека или из самой инструкции?
- Что произойдет, если значение не укладывается в заданное количество бит? Генерируется ли исключение или флаг успеха равен нулю, автоматически возвращаемый в верхней части стека?
- Что произойдет, если будет сгенерировано исключение insuexception или будет возвращен нулевой флаг успеха вместе с пробелом, оставшимся в *конструкторе*? Является ли неизмененный оригинальный *строитель*?

Мнемоника примитивов сериализации клеток обычно начинается с `ST`. Последующие буквы описывают следующие атрибуты:

- Тип сериализуемых значений и формат сериализации (например, для знаковых целых чисел, `U` для целых чисел без знака). `Я`
- Источник ширины поля в используемых битах (например, инструкции сериализации означают, что битовая ширина n предоставляется inX для целого числа стека; в противном случае она должна быть встроена в инструкцию в качестве немедленного значения).
- Действие, которое необходимо выполнить, если операция не может быть завершена (по умолчанию генерируется исключение; «тихие» версии инструкций сериализации помечены буквой `Q` в их мнемонике).

Эта схема классификации используется для создания более полной таксономии примитивов сериализации клеток, которую можно найти в A.7.1.

3.2.7. Целочисленные примитивы сериализации. Целочисленные примитивы сериализации также могут быть классифицированы в соответствии с приведенной выше таксономией. Например:

- Существуют знаковые и беззнаковые (big-endian) целочисленные примитивы сериализации.
- Размер *n* используемого битового поля ($1 \le n \le 257$ для знаковых целых чисел, $0 \le n \le 256$ для целых чисел без знака) может быть либо взят из верхней части стека, либо встроен в саму инструкцию.
- Если целое число *x* , подлежащее сериализации, не находится в диапазоне $-2^{n-1} \le x < 2^{n-1}$ (для знаковой целочисленной сериализации) или $0 \le x < 2^n$ (для целочисленной сериализации без знака), обычно создается исключение проверки диапазона, и если *n* бит не могут быть сохранены в предоставленном *Builder, создается исключение переполнения ячеек.*
- Тихие версии инструкций сериализации не создают исключений; вместо этого они нажимают -1 поверх результирующего *Builder* при успешном завершении или возвращают исходный *Builder* с 0 поверх него, чтобы указать на сбой.

Инструкции целочисленной сериализации имеют мнемонику, такую как `STU 20` («хранить 20-битное целое значение без знака») или `STIXQ` («тихо хранить целое значение переменной длины, предоставляемое в стеке»). Полный список этих инструкций, включая их мнемонику, описания и опкоды, приведен в A.7.1.

3.2.8. Целые числа в ячейках по умолчанию являются биг-эндианами. Обратите внимание, что порядок битов по умолчанию в целых *числах, сериализованных в ячейки, является биг-порядковым*, а не литтлендианским.[14] В этом отношении *TVM является машиной с большим порядком байтов*. Однако это влияет только на сериализацию целых чисел внутри ячеек. Внутреннее представление типа значения

[14] Отрицательные числа представлены с помощью дополнения двух. Например, целое число 17 сериализуется инструкцией `STI 8` в bitstring `xEF`.

Integer зависит от реализации и не имеет отношения к работе TVM. Кроме того, существуют некоторые специальные примитивы, такие как STULE для (де)сериализации целых чисел с младшим порядком байтов, которые должны храниться в интегральном числе байтов (в противном случае «малый порядковый номер» не имеет смысла, если только вы также не готовы вернуть порядок битов внутри октетов). Такие примитивы полезны для взаимодействия с миром с младшим порядком байтов, например, для синтаксического анализа сообщений пользовательского формата, поступающих в смарт-контракт TON Blockchain из внешнего мира.

3.2.9. Другие примитивы сериализации. Другие примитивы создания ячеек сериализуют битовые строки (т. е. срезы ячеек без ссылок), либо взятые из стека, либо предоставленные в виде литеральных аргументов; срезы ячеек (которые соединяются с конструктором ячеек очевидным образом); другие *строители (которые также объединены); и ссылки на ячейки (*STREF).

3.2.10. Другие примитивы создания клеток. В дополнение к примитивам сериализации ячеек для определенных встроенных типов значений, описанных выше, существуют простые примитивы, которые создают новый пустой *Builder* и отправляют его в стек (NEWC) или преобразуют *Builder* в ячейку (ENDC), тем самым завершая процесс создания ячейки. ENDC можно объединить с STREF в единую инструкцию ENDCST, которая завершает создание ячейки и сразу же сохраняет ссылку на нее в «внешнем» *конструкторе*. Существуют также примитивы, которые получают количество битов данных или ссылок, уже хранящихся в *построителе*, и проверяют, сколько битов данных или ссылок может быть сохранено.

3.2.11. Таксономия примитивов десериализации клеток. Примитивы разбора ячеек, или десериализации, можно классифицировать, как описано в разделе 3.2.6, со следующими изменениями:

- Они работают с *Slices (представляющими оставшуюся часть анализируемой ячейки) вместо Builders.*
- Они возвращают десериализованные значения вместо того, чтобы принимать их в качестве аргументов.
- Они могут выпускаться в двух вариантах, в зависимости от того, удаляют ли они десериализованную порцию из *поставляемого Slice*

(«операции извлечения») или оставляют ее неизмененной («операции предварительной выборки»).

- Их мнемоника обычно начинается с LD (или PLD для операций предварительной выборки) вместо ST.

Например, 20-битное целое число с большим порядком байтов, ранее сериализованное в ячейку инструкцией STU 20, скорее всего, будет десериализовано позже соответствующей инструкцией LDU 20.

Опять же, более подробная информация об этих инструкциях приводится в пункте A.7.2.

3.2.12. Другие примитивы среза клеток. В дополнение к примитивам десериализации клеток, описанным выше, TVM предоставляет некоторые очевидные примитивы для инициализации и завершения процесса десериализации клеток. Например, можно преобразовать *Cell* в *Slice* (`CTOS`), чтобы можно было начать ее десериализацию; или проверить, пуст ли *Slice*, и сгенерировать исключение, если это не так (`ENDS`); или десериализовать ссылку на ячейку и немедленно преобразовать ее в *Slice* (`LDREFTOS`, эквивалентная двум инструкциям `LDREF` и `CTOS`).

3.2.13. Изменение сериализованного значения в ячейке. Читатель может задаться вопросом, как могут быть изменены значения, сериализованные внутри ячейки. Предположим, что ячейка содержит три сериализованных 29-битных целых числа (x,y,z),представляющих координаты точки в пространстве, и мы хотим заменить y на $y' = y + 1$, оставив остальные координаты нетронутыми. Как мы этого добьемся?

TVM не предлагает никаких способов изменения существующих значений (см. 2.3.4 и 2.3.5), поэтому наш пример может быть выполнен только с помощью серии операций следующим образом:

1. Десериализовать исходную ячейку на три *целых числа* x, y, z в стеке (например, с помощью `CTOS; LDI 29; LDI 29; LDI 29; ENDS`).
2. Увеличение y на единицу (например, с помощью `SWAP; INC; SWAP`).
3. Наконец, сериализуйте *полученные целые числа в новую ячейку (например, с помощью* `XCHG s2; NEWC; STI 29; STI 29; STI 29; ENDC`).

3.2.14. Изменение постоянного хранения смарт-контракта. Если TVM-код хочет изменить свое постоянное хранилище, представленное деревом ячеек, укорененных в $c4$, ему просто нужно переписать регистр управления $c4$ корнем дерева ячеек, содержащих новое значение его постоянного хранилища. (Если необходимо изменить только часть постоянного хранилища, см. 3.2.13.)

3.3 Хэш-карты или словари

Хэш-карты, или *словари*, представляют собой определенную структуру данных, представленную деревом ячеек. По сути, хэш-карта представляет собой карту из *ключей*, которые представляют собой битовые строки фиксированной или переменной длины, в *значения* произвольного типа X, таким образом, чтобы можно было быстро искать и модифицировать. В то время как любая такая структура может быть проверена или изменена с помощью универсальных примитивов сериализации и десериализации клеток, TVM вводит специальные примитивы для облегчения работы с этими хэш-картами.

3.3.1. Основные типы хэш-карт. Двумя основными типами хэш-карт, предопределенными в TVM, являются *HashmapE* n X или *HashmapE*(n,X),которые представляют частично определенную карту *из n*-битных строк (называемых *ключами*) для некоторых фиксированных 0 ≤ n ≤ 1023 в *значения* некоторого типа *X, и Hashmap(n,X),* которая похожа на HashmapE(n,X), но не может быть пустой (т.е. он должен содержать хотя бы одну пару ключ-значение).

Также доступны другие типы хэш-карт, например, с ключами произвольной длины до некоторой предопределенной привязки (до 1023 бит).

3.3.2. Хэшкарты как деревья Патрисии. Абстрактное представление хэш-карты в TVM представляет собой *дерево Патрисии* или компактную *двоичную трию*. Это двоичное дерево с ребрами, помеченными битовыми строками, так что объединение всех меток ребер на пути от корня к листу равно ключу хэш-карты. Соответствующее значение сохраняется в этом листе (для хэш-карт с ключами фиксированной длины) или опционально и в промежуточных вершинах (для хэш-карт с ключами переменной длины). Кроме того, любая промежуточная вершина должна

иметь два дочерних элемента, и метка левого дочернего элемента должна начинаться с двоичного нуля, в то время как метка правого дочернего элемента должна начинаться с двоичного. Это позволяет нам не хранить первый бит меток края явно.

Нетрудно заметить, что любая коллекция пар ключ-значение (с отдельными ключами) представлена уникальным деревом Патрисии.

3.3.3. Сериализация хэш-карт. Сериализация хэш-карты в дерево ячеек (или, в более общем плане, в *срез*) определяется следующей схемой TL-B:[15]

```
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 {m:#} {n:#} len:(Unary ~n)
            s:(n * Bit) = HmLabel ~n m;
hml_long$10 {m:#} n:(#<= m) s:(n * Bit) = HmLabel ~n
m;
hml_same$11 {m:#} 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;
```

[15] Описание более старой версии TL можно найти на `https://core.telegram.org/mtproto/TL`.

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

3.3.4. Краткое объяснение схем TL-B. Схема TL-B, как и приведенная выше, включает в себя следующие компоненты.

Правая часть каждого «уравнения» представляет собой *тип*, либо простой (например, `Bit` или `True`), либо параметризованный (например, `Hashmap` n X). Параметры типа должны быть либо натуральными числами (т.е. неотрицательными целыми числами, которые на практике должны укладываться в 32 бита), такими как n в `Hashmap` n X, либо другими типами, такими как X в `Hashmap` n X.

Левая часть каждого уравнения описывает способ определения или даже сериализации значения типа, указанного в правой части. Такое описание начинается с имени *конструктора*, такого как `hm_edge` или `hml_long`, за которым сразу же следует необязательный *тег конструктора*, такой как `#_` или `$10`, который описывает битовую строку, используемую для кодирования (сериализации) рассматриваемого конструктора. Такие теги могут быть даны как в двоичной (после знака доллара), так и в шестнадцатеричной нотации (после хэш-знака), используя соглашения, описанные в 1.0. Если тег явно не указан, TL-B вычисляет 32-разрядный тег конструктора по умолчанию, хэшируя текст «уравнения», определяющего этот конструктор определенным образом. Поэтому пустые теги должны быть явно предоставлены `#_` или `$_`. Все имена конструкторов должны быть разными, а теги конструктора для одного и того же типа должны представлять собой код префикса (в противном случае десериализация не была бы уникальной).

За конструктором и его необязательным тегом следуют *определения полей*. Каждое определение поля имеет форму *ident* : *type-expr*, где *ident* — идентификатор с именем поля [16] (заменяется подчеркиванием для анонимных полей), а *type-expr* — тип поля. Тип, представленный здесь, является выражением *типа*, которое может включать простые типы или параметризованные типы с подходящими параметрами. *Переменные*, т.е. (идентификаторы) ранее определенных полей типов # (натуральные

[16] Имя поля полезно для представления значений определяемого типа в человекочитаемой форме, но оно не влияет на двоичную сериализацию.

числа) или *Type* (тип типов), могут использоваться в качестве параметров для параметризованных типов. Процесс сериализации рекурсивно сериализует каждое поле в соответствии с его типом, а сериализация значения в конечном итоге состоит из объединения битовых строк, представляющих конструктор (т. е. тег конструктора) и значений полей.

Некоторые поля могут быть *неявными*. Их определения окружены фигурными скобками, которые указывают на то, что поле фактически не присутствует в сериализации, но что его значение должно быть выведено из других данных (обычно параметров сериализуемого типа).

Некоторые вхождения «переменных» (т.е. уже определенных полей) префиксируются тильдой. Это указывает на то, что вхождение переменной используется противоположным образом поведения по умолчанию: в левой части уравнения это означает, что переменная будет выведена (вычислена) на основе этого вхождения, а не подставлена ее ранее вычисленное значение; в правой части, наоборот, это означает, что переменная не будет выведена из сериализуемого типа, а скорее будет вычислена в процессе десериализации. Другими словами, тильда преобразует «входной аргумент» в «выходной аргумент», и наоборот.[17]

Наконец, некоторые уравнения также могут быть включены в фигурные скобки. Это некие «уравнения», которые должны удовлетворяться включенными в них «переменными». Если одна из переменных имеет префикс тильды, ее значение будет однозначно определяться значениями всех других переменных, участвующих в уравнении (которые должны быть известны в этой точке), когда определение обрабатывается слева направо.

Карет (ˆ), предшествующий типу X , означает, что вместо сериализации значения типа X в виде битовой строки внутри текущей ячейки мы помещаем это значение в отдельную ячейку и добавляем ссылку на него в текущую ячейку. Поэтому $\hat{}\mathbb{X}$ означает «тип ссылок на ячейки, содержащие значения типа X».

Параметризованный тип $\#<= p$ с $p : \#$ (это обозначение означает «p типа $\#$», т. е. натуральное число) обозначает подтип натуральных чисел типа $\#$, состоящий из целых чисел $0\ldots p$; он сериализуется в $\lceil \log_2(p + 1) \rceil$ биты как целое число больших порядковых чисел без знака. Тип $\#$ сам по

[17] Это операция «линейного отрицания» $(-)^{\perp}$ линейной логики, отсюда и наша нотация $\tilde{\cdot}$

себе сериализуется как 32-разрядное целое число без знака. Параметризованный тип `##` b с b : `#<=31` эквивалентен `#<=` $2^b - 1$ (т.е. это целое число без знака b-бит).

3.3.5. Применение к сериализации хэш-карт. Поясним чистый результат применения общих правил, описанных в разделе 3.3.4, к схеме TL-B, представленной в разделе 3.3.3.

Предположим, мы хотим сериализовать значение типа *HashmapE n X* для некоторого целого числа $0 \le n \le 1023$ и некоторого типа *X* (т.е. словаря с *n-битными ключами и значениями типа X, допускающего абстрактное представление в виде дерева Патрисии (ср. 3.3.2))*.

Прежде всего, если наш словарь пуст, он сериализуется в один двоичный файл `0`, который является тегом нулевого конструктора `hme_empty`. В противном случае его сериализация состоит из двоичного файла `1` (тег `hme_root`) вместе со ссылкой на ячейку, содержащую сериализацию значения типа *Hashmap* n X (т.е. обязательно непустого словаря).

Единственный способ сериализации значения типа *Hashmap* n X задается конструктором `hm_edge`, который инструктирует нас сначала сериализовать метку `метки` ребра, ведущую к корню рассматриваемого поддерева (т.е. общий префикс всех ключей в нашем (под)словаре). Эта метка относится к типу `HmLabel` $l^{\perp}$ n, что означает, что это битовая полоса длиной не более n, сериализованная таким образом, что истинная длина l метки, $0 \le l \le n$, становится известной из сериализации метки. (Этот специальный метод сериализации описан отдельно в 3.3.6.)

За меткой должна следовать сериализация `узла` типа *HashmapNode m X*, где $m = n - l$. Он соответствует вершине дерева Патрисии, представляющей собой непустой субдиктарий оригинального словаря с .m-битными ключами, полученными путем удаления из всех ключей исходного субдиционария их общего префикса длины l

Если $m = 0$, значение типа `HashmapNode` 0 X задается конструктором `hmn_leaf`, который описывает лист дерева Патрисии или, эквивалентно, субдистанцию с 0-битными ключами. Лист просто состоит из соответствующего `значения` типа X и сериализуется соответствующим образом.

С другой стороны, если $m > 0$, то значение типа `HashmapNode` $m\ X$ соответствует форку (т.е. промежуточному узлу) в дереве Patricia и задается конструктором hmn_fork. Его сериализация состоит из левой и `правой` ссылок на ячейки, содержащие значения типа `Hashmap` $m{-}1\ X$, которые соответствуют левому и правому дочерним элементам рассматриваемого промежуточного узла или, эквивалентно, двум поддистанциям оригинального словаря, состоящим из пар ключ-значение с ключами, начинающимися с двоичного `0` или двоичного `1`. соответственно. Поскольку первый бит всех ключей в каждом из этих подразделов известен и фиксирован, он удаляется, а результирующие (обязательно непустые) субдикции рекурсивно сериализуются как значения типа `Hashmap` $m{-}1\ X$.

3.3.6. Сериализация этикеток. Существует несколько способов сериализации метки длиной не более n, если ее точная длина равна $l \leq n$ (напомним, что точная длина должна быть выведена из сериализации самой метки, в то время как верхняя граница n известна до того, как метка будет сериализована или десериализована). Эти способы описываются тремя конструкторами `hml_short`, `hml_long` и `hml_same` типа `HmLabel` $l^{\perp}\ n$:

- hml_short — описывает способ сериализации «коротких» меток небольшой длины *l ≤ n. Такая сериализация состоит из двоичного 0 (тег конструктора hml_short), за которым следуют l* двоичные 1s и один двоичный 0 (унарное представление длины *l), за которым следуют l* биты, составляющие саму метку.
- hml_long — описывает способ сериализации «длинных» меток произвольной длины *l ≤ n.* Такая сериализация состоит из двоичного файла 10 (тег конструктора hml_long), за которым следует двоичное представление большого порядкового числа байтов длины $0 \leq l \leq n$ в битах $\lceil \log_2(n + 1) \rceil$, за которыми следуют *l* биты, составляющие саму метку.
- hml_same — описывает способ сериализации «длинных» меток, состоящий из *l* повторений одного бита *v. Такая сериализация состоит из 11 (тег конструктора hml_same), за которым следует бит v, за которым следует длина l,* хранящаяся в битах $\lceil \log_2(n + 1) \rceil$, как и раньше.

Каждая метка всегда может быть сериализована по крайней мере двумя различными способами, используя `hml_short` или `hml_long` конструкторы. Обычно самая короткая сериализация (а в случае галстука — лексикографически наименьшая среди самых коротких) является предпочтительной и генерируется примитивами хешкарты TVM, в то время как другие варианты по-прежнему считаются действительными.

Эта схема кодирования меток была разработана таким образом, чтобы быть эффективной для словарей со «случайными» ключами (например, хэши некоторых данных), а также для словарей с «обычными» ключами (например, представления целых чисел в некотором диапазоне с большим порядком байтов).

3.3.7. Пример сериализации словаря. Рассмотрим словарь с тремя 16-битными ключами 13, 17 и 239 (считаются целыми числами с большим порядком байтов) и соответствующими 16-битными значениями 169, 289 и 57121.

В двоичной форме:

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

Соответствующее дерево Патриции состоит из корня A, двух промежуточных узлов B и C и трех листовых узлов D, E и F, соответствующих 13, 17 и 239 соответственно. Корень A имеет только один дочерний элемент, B; метка на краю AB равна $00000000 = 0^8$. Узел B имеет два дочерних элемента: его левый дочерний узел является промежуточным узлом C с краем BC, помеченным $(0)00$, в то время как его правый дочерний элемент является листом F с BF, помеченным $(1)1101111$. Наконец, C имеет два листовых дочерних D и E, причем CD помечен $(0)1101$ и CE — $(1)0001$.

Соответствующее значение типа `HashmapE 16 (## 16)` может быть записано в удобочитаемой форме как:

```
(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)))))
```

Сериализация этой структуры данных в дерево ячеек состоит из шести ячеек со следующими двоичными данными, содержащимися в них:

```
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
```

Здесь A — корневая клетка, $A.0$ — клетка в первой ссылке A, $A.1$ — клетка во второй ссылке A и так далее. Это дерево ячеек может быть представлено более компактно с помощью шестнадцатеричной нотации, описанной в 1.0, с использованием отступа для отражения структуры дерева клеток:

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

В общей сложности 93 бита данных и 5 ссылок в 6 ячейках были использованы для сериализации этого словаря. Обратите внимание, что простое представление трех 16-битных ключей и соответствующих им 16-битных значений уже потребовало бы 96 бит (хотя и без каких-либо ссылок), поэтому эта конкретная сериализация оказывается довольно эффективной.

3.3.8. Способы описания сериализации типа X. Обратите внимание, что встроенные примитивы TVM для работы со словарем должны знать что-то о сериализации типа X; в противном случае они не смогли бы корректно работать с *Hashmap n X*, потому что значения типа X сразу же содержатся в ячейках листьев дерева Patricia. Существует несколько вариантов описания сериализации типа *X*:

- Простейшим случаем является случай, когда *X* = ^Y для какого-либо другого типа *Y*. При этом сама сериализация *X* всегда состоит из одной ссылки на ячейку, которая на самом деле должна содержать значение типа *Y*, что не имеет отношения к примитивам словарных манипуляций.
- Другой простой случай, когда сериализация любого значения типа *X* всегда состоит из $0 \le b \le 1023$ бит данных и $0 \le r \le 4$ ссылок. Целые числа *b* и *r* затем могут быть переданы в примитив манипуляции словаря в качестве простого описания *X. (Обратите внимание, что предыдущий случай соответствует b* = 0, *r* = 1.)
- Более сложный случай может быть описан четырьмя целыми числами $1 \le b_0,b_1 \le 1023$, $0 \le r_0,r_1 \le 4$, причем b_i и r_i используются, когда первый бит сериализации равен *i. Когда* $b_0 = b_1$ и $r_0 = r_1$*, этот случай сводится к предыдущему.*
- Наконец, наиболее общее описание сериализации типа *X* дается функцией *расщепления* $split_X$ для *X,* которая принимает один параметр Slice *s и возвращает два Slices, s'* и *s'', где s'* — единственный префикс *s*, который является сериализацией значения типа *X, а s''* — остаток *s.* Если такого префикса не существует, ожидается, что функция разделения вызовет исключение. Обратите внимание, что компилятор для языка высокого уровня, который поддерживает некоторые или все алгебраические типы TL-B, скорее всего, автоматически генерирует функции расщепления для всех типов, определенных в программе.

3.3.9. Упрощающее предположение о сериализации X. Можно заметить, что значения типа X всегда занимают оставшуюся часть ячейки `hm_edge`/`hme_leaf` внутри сериализации *HashmapE n X*. Поэтому, если мы не будем настаивать на строгой проверке всех доступных словарей, мы можем предположить, что все, что осталось необранным в `ячейке` `hm_edge`/`hme_leaf` после десериализации ее `label`, является

значением типа X. Это значительно упрощает создание примитивов словарных манипуляций, поскольку в большинстве случаев они вообще не нуждаются в какой-либо информации о X .

3.3.10. Основные словарные операции. Представим классификацию основных операций со словарями (т.е. значения *D* типа *HashmapE n X*):

- Get(D,k) — Заданный *D* : *HashmapE(n,X) и ключ k* : *n·bit, возвращает соответствующее значение D[k] : X*[?] хранится в *D.*
- Set(D,k,x) — Заданный *D* : *HashmapE(n,X), ключ k* : *n* · bit и значение *x* : *X,* устанавливает *D'[k] в x* в копии *D'* из *D* и возвращает полученный словарь *D'* (ср. 2.3.4).
- Add(D,k,x) — аналогично Set, но добавляет пару ключ-значение (k,x) в *D* только в том случае, если ключ *k* отсутствует в *D.*
- Replace(D,k,x) — аналогично Set, но изменяет *D'[k] на x* только в том случае, если ключ *k* уже присутствует в *D.*
- GetSet, GetAdd, GetReplace — аналогично Set, Add и Replace соответственно, но также возвращает старое значение *D[k].*
- Delete(D,k) — удаляет ключ *k* из словаря *D* и возвращает полученный словарь *D'.*
- GetMin(D), GetMax(D) — получает минимальный или максимальный ключ *k* из словаря *D вместе с соответствующим значением x* : *X.*
- RemoveMin(D), RemoveMax(D) — аналогично GetMin и GetMax, но также удаляет соответствующий ключ из словаря *D* и возвращает измененный словарь *D'.* Может использоваться для итерации по всем элементам *D,* эффективно используя (копию) самого *D* в качестве итератора.
- GetNext(D,k) — вычисляет минимальный ключ *k' > k* (или *k' ≥ k* в варианте) и возвращает его вместе с соответствующим *значением x'* : *X.* Может использоваться для итерации по всем элементам *D.*
- GetPrev(D,k) — вычисляет максимальный ключ *k'< k* (или *k' ≤ k* в варианте) и возвращает его вместе с соответствующим значением *x'* : *X.*
- Empty(n) — Создает пустой словарь *D* : *HashmapE(n,X).*
- IsEmpty(D) — проверяет, пуст ли словарь.
- Create(n,{(k_i,x_i)}) — Заданный *n,* создает словарь из списка *(k_i,x_i)* пар ключ-значение, переданных в стеке.

- GetSubdict(D,l,k_0) — Заданное *D* : *HashmapE(n,X)* и некоторая *l-битная* строка k_0 : *l* ·bit для 0 ≤ *l* ≤ *n,* возвращает субдиционарный *D'*= D/k_0 из *D,* состоящий из ключей, начинающихся с k_0. Результат *D'* может быть либо типа *HashmapE(n,X),* либо типа *HashmapE(n − l,X).*
- ReplaceSubdict(*D,l,*k_0*,D'*) — Заданный *D* : *HashmapE(n,X), 0 ≤ l ≤ n,* k_0 : *l* · bit, и *D'* : *HashmapE(n − l,X),* заменяет на *D'* субдиционарный D/k_0 D , состоящий из ключей, начинающихся с k_0, и возвращает полученный словарь *D''* : *HashmapE(n;X).* Некоторые варианты *ReplaceSubdict* могут также возвращать старое значение рассматриваемого субдиционарного D/k_0 .
- DeleteSubdict(*D,l,*k_0) — эквивалент ReplaceSubdict с *D'*, являющимся пустым словарем.
- Split(D) — Заданный *D* : *HashmapE(n,X),* возвращает D_0 := *D/0 u* D_1 := *D/1 : HashmapE(n* − 1,*X*), два поддистанции *D*, состоящие из всех ключей, начинающихся с 0 и 1 соответственно.
- Merge(D_0,D_1) — Заданные D_0 и D_1 : *HashmapE(n−1,X),* вычисляет *D* : *HashmapE(n,X),* так что *D/0* = D_0 и *D/1* = D_1.
- Foreach(D,f) — выполняет функцию *f* с двумя аргументами *k* и *x*, при этом *(k,x)* проходит по всем парам ключ-значение словаря *D* в лексикографическом порядке.[18]
- ForeachRev(D,f) — аналогично Foreach, но обрабатывает все пары ключевых значений в обратном порядке.
- TreeReduce(*D,o,f,g*) — Заданный *D* : *HashmapE(n,X),* значение *o* : *X,* и две функции *f* : *X* →?*Y* и *g* : *Y* × *Y* →?*Y* , выполняет «древовидное восстановление» *D* , сначала применяя *f* ко всем листьям, а затем используя *g*, чтобы вычислить значение, соответствующее форку, начиная со значений, присвоенных его дочерним элементам.[19]

[18] Фактически, f может получить m дополнительных аргументов и вернуть m модифицированных значений, которые передаются при следующем вызове f. Это может быть использовано для реализации операций «map» и «reduce» со словарями.

[19] Могут быть введены версии этой операции, где f и g получают дополнительный аргумент bitstring, равный ключу (для листьев) или общему префиксу всех ключей (для форков) в соответствующем поддереве.

3.3.11. Таксономия словарных примитивов. Словарные примитивы, подробно описанные в A.10, можно классифицировать по следующим категориям:

- Какую словарную операцию (см. 3.3.10) они выполняют?
- Являются ли они специализированными для случая *X* = ^Y? Если да, то представляют ли они значения типа *Y* по *Cells или по Slices?* (Универсальные версии всегда представляют значения типа X как фрагменты.)
- Передаются ли и возвращаются ли сами словари в виде *Cells* или *Slices?* (Большинство примитивов представляют словари в виде *Slices*.)
- Является ли длина ключа *n* фиксированной внутри примитива или она передается в стеке?
- Представлены ли ключи *Slices* или *целыми числами* со знаком или без знака?

Кроме того, TVM включает в себя специальные примитивы сериализации/десериализации, такие как `STDICT`, `LDDICT` и `PLDDICT`. Они могут быть использованы для извлечения словаря из сериализации охватывающего объекта или для вставки словаря в такую сериализацию.

3.4 Хэш-карты с ключами переменной длины

TVM обеспечивает некоторую поддержку словарей или хэш-карт с ключами variablelength, в дополнение к поддержке словарей с ключами фиксированной длины (как описано в 3.3 выше).

3.4.1. Сериализация словарей с помощью ключей переменной длины. Сериализация *VarHashmap в* дерево ячеек (или, в более общем плане, в *Slice*) определяется схемой TL-B, аналогичной описанной в 3.3.3:

```
vhm_edge#_ {n:#} {X:Type} {l:#} {m:#} label:(HmLabel
~l n)
       {n = (~m) + l} узел:(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. Сериализация префиксных кодов. Одним из частных случаев словаря с ключами переменной длины является *префиксный код*, где ключи не могут быть префиксами друг друга. Значения в таких словарях могут встречаться только в листьях дерева патриции.

Сериализация префиксного кода определяется следующей схемой TL-B:

```
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 Поток управления, продолжения и исключения

В этой главе описываются *продолжения*, которые могут представлять маркеры выполнения и обработчики исключений в TVM. Продолжения глубоко вовлечены в поток управления программой TVM; в частности, вызовы подпрограмм и условное и итерированное выполнение реализуются в TVM с использованием специальных примитивов, которые

принимают одно или несколько продолжений в качестве своих аргументов.

Завершаем эту главу обсуждением проблемы рекурсии и семейств взаимно рекурсивных функций, усугубляемой тем, что циклические ссылки не допускаются в структурах данных TVM (включая код TVM).

4.1 Продолжения и подпрограммы

Вспомним (cf.1.1.3), что *значения Continuation* представляют собой «маркеры выполнения», которые могут быть выполнены позже, например, примитивами `EXECUTE=CALLX` («execute» или «call indirect») или `JMPX` («косвенный прыжок»). Таким образом, продолжения отвечают за выполнение программы и активно используются примитивами потока управления, позволяющими вызовы подпрограмм, условные выражения, циклы и т. д.

4.1.1. Обычные продолжения. Наиболее распространенным видом продолжений являются *обычные продолжения*, содержащие следующие данные:

- Код фрагмента (см. 1.1.3 и 3.2.2), содержащий (оставшуюся часть) исполняемый код TVM.
- Стек стека (возможно, пустой), содержащий исходное содержимое стека для выполняемого кода.
- Список (возможно, пустой) сохранения пар (c*(i)*,v_i) (также называемый "savelist"), содержащий значения контрольных регистров, которые должны быть восстановлены перед выполнением кода.
- 16-разрядное целое значение cp, выбирающее кодовую страницу TVM, используемую для интерпретации кода TVM из **code**.
- Необязательное неотрицательное целое число **nargs**, указывающее на количество аргументов, ожидаемых продолжением.

4.1.2. Простые обычные продолжения. В большинстве случаев обычные продолжения являются простейшими, имеющими пустой **`stack`** и **`save`**. Они состоят по существу из ссылочного **`code`** (оставшейся части) выполняемого кода и кодовой страницы **`cp`**, которая будет использоваться при декодировании инструкций из этого кода.

4.1.3. Текущее продолжение `cc`. «Текущее продолжение» `cc` является важной частью общего состояния TVM, представляя код, выполняемый прямо сейчас (см. 1.1). В частности, то, что мы называем «текущим стеком» (или просто «стеком») при обсуждении всех других примитивов, на самом деле является стеком текущего продолжения. Все остальные компоненты общего состояния TVM также могут рассматриваться как части текущего продолжения `cc`; однако они могут быть извлечены из текущего продолжения и сохранены отдельно как часть общего состояния по соображениям производительности. Вот почему мы описываем стек, управляющие регистры и кодовую страницу как отдельные части состояния TVM в 1.4.

4.1.4. Нормальная работа TBM, или основного контура. TVM обычно выполняет следующие операции:

Если текущее продолжение `cc` является обычным, оно декодирует первую инструкцию из *Slice* **code**, подобно тому, как другие ячейки десериализуются примитивами TVM `LD*` (ср. 3.2 и 3.2.11): сначала декодирует опкод, а затем параметры инструкции (например, 4-битные поля, указывающие на «регистры стека», используемые для примитивов манипулирования стеком, или постоянные значения для примитивов "push constant" или "literal"). Остальная часть *Slice* затем помещается в `код` нового **cc**, а декодированная операция выполняется в текущем стеке. Весь этот процесс повторяется до тех пор, пока в **cc.code** не останется никаких операций.

Если **code** пуст (т.е. не содержит битов данных и ссылок), или если встречается (редко необходимая) явная инструкция возврата подпрограммы (`RET`), текущее продолжение отбрасывается, а «возвращаемое продолжение» из контрольного регистра **c0** загружается в **cc** (этот процесс обсуждается более подробно, начиная с 4.1.6). [20] Затем выполнение продолжается путем синтаксического анализа операций из нового текущего продолжения.

4.1.5. Внеочередные продолжения. В дополнение к обычным продолжениям, рассмотренным до сих пор (см. 4.1.1), TVM включает в

[20] Если в коде не осталось битов данных, но все еще есть ровно одна ссылка, выполняется неявный `JMP` к ячейке в этой ссылке вместо неявного `RET`.

себя некоторые *экстраординарные продолжения*, представляющие некоторые менее распространенные состояния. Примеры экстраординарных продолжений включают:

- Продолжение ec_quit с нулевым параметром, что представляет собой окончание работы TVM. Это продолжение является исходным значением c0, когда TVM начинает выполнять код смарт-контракта.
- Продолжение ec_until, которое содержит ссылки на два других продолжения (обычное или нет), представляющих тело выполняемого цикла и код, который будет выполняться после цикла.

Выполнение экстраординарного продолжения TBM зависит от его конкретного класса, и отличается от операций для обычных продолжений, описанных в 4.1.4.[21]

4.1.6. Переход на другое продолжение: `JMP` и `RET`. Процесс переключения на другое продолжение c может выполняться такими инструкциями, как `JMPX` (который берет c из стека) или `RET` (который использует `c0` как c). Этот процесс немного сложнее, чем просто установка значения `cc` в c: перед этим либо все значения, либо верхние n значений в текущем стеке перемещаются в стек продолжения c, и только тогда остальная часть текущего стека отбрасывается.

Если все значения необходимо переместить (наиболее распространенный случай), и если продолжение c имеет пустой стек (также наиболее распространенный случай; обратите внимание, что экстраординарные продолжения предполагают пустой стек), то новый стек c равен стеку текущего продолжения, поэтому мы можем просто перенести текущий стек в полном объеме в c . (Если мы сохраним текущий стек как отдельную часть общего состояния TVM, мы вообще ничего не должны делать.)

4.1.7. Определение количества n аргументов, передаваемых следующему продолжению c. По умолчанию n равно глубине текущего стека. Однако, если c имеет явное значение **`nargs`** (число аргументов, которые должны

[21] Технически TVM может просто вызвать виртуальный метод `run()` продолжения, находящегося в настоящее время в `cc`.

быть предоставлены), то n вычисляется как n', *равное c.nargs* минус текущая глубина стека *c.*

Кроме того, существуют специальные формы JMPX и RET, которые предоставляют явное значение *n'',* количество параметров из текущего стека, передаваемых продолжению *c.* Если указано значение *n''*, оно должно быть меньше или равно глубине текущего стека, иначе возникает исключение недопотока стека. Если предоставлены и *n'*, и *n''*, мы должны иметь *n' ≤ n'', и в этом случае используется n = n'*. Если *n''* предоставляется, а *n'* нет, то используется *n = n''*.

Можно также предположить, что значение по умолчанию n'' равно глубине исходного стека, и что значения n'' всегда удаляются из верхней части исходного стека, даже если только n' из них фактически перемещаются в стек следующего продолжения *c*. Несмотря на то, что остальная часть текущего стека впоследствии отбрасывается, это описание станет полезным позже.

4.1.8. Восстановление контрольных регистров из нового продолжения *c*. После вычисления нового стека значения управляющих регистров, присутствующих в *c.***save**, восстанавливаются соответствующим образом, а текущая кодовая страница **cp** также устанавливается в *c.***cp**. Только после этого TVM устанавливает **cc** равным новому *c* и начинает его выполнение.[22]

4.1.9. Вызовы подпрограмм: `CALLX` или `примитивы` EXECUTE. Выполнение продолжений в качестве подпрограмм несколько сложнее, чем переключение на продолжения.

Рассмотрим примитив `CALLX` или `EXECUTE` , который берет продолжение *c* из (текущего) стека и выполняет его как подпрограмму.

Помимо выполнения манипуляций со стеком, описанных в разделах 4.1.6 и 4.1.7, и установки новых регистров управления и кодовой страницы, как описано в 4.1.8, эти примитивы выполняют несколько дополнительных шагов:

[22] Уже использованный список сохранения `cc.save` нового `cc` очищается перед началом выполнения.

1. После удаления верхних *значений n''* из текущего стека (ср. 4.1.7) остаток (обычно пустой) не отбрасывается, а вместо этого сохраняется в (старом) текущем продолжении `cc`.

2. Старое значение специального регистра `c0` сохраняется в (ранее пустом) списке сохранения `cc.save`.

3. Модифицированное таким образом продолжение `cc` не отбрасывается, а вместо этого устанавливается как новое `c0`, которое выполняет роль «следующего продолжения» или «обратного продолжения» для вызываемой подпрограммы.

4. После этого переход на *c* продолжается, как и раньше. В частности, некоторые управляющие регистры восстанавливаются из *c.save*, потенциально перезаписывая значение `c0`, заданное на предыдущем шаге. (Поэтому хорошей оптимизацией было бы проверить, что *c0* присутствует в *c.save* с самого начала, и пропустите три предыдущих шага как бесполезные в этом случае.)

Таким образом, вызываемая подпрограмма может вернуть управление вызывающему объекту, переключив текущее продолжение на возвращаемое продолжение, сохраненное в `c0`. Вложенные вызовы подпрограмм работают правильно, поскольку предыдущее значение `c0` в конечном итоге сохраняется в новом `c0` реестре элементов управления `savelist c0.save`, из которого оно восстанавливается позже.

4.1.10. Определение количества аргументов, передаваемых и/или возвращаемых значений, принятых из подпрограммы. Подобно `JMPX` и `RET`, `CALLX` также имеет специальные (редко используемые) формы, которые позволяют явно указать количество *n''* аргументов, передаваемых из текущего стека в вызываемую подпрограмму (по умолчанию *n''* равен глубине текущего стека, т. е. передается целиком). Кроме того, может быть указано второе число *n'''*, используемое для установки **`nargs`** модифицированного `продолжения cc` перед сохранением его в новом `c0`; новые **`nargs`** равны глубине старого стека минус *n''* плюс *n'''*. Это означает, что вызывающий абонент готов передать в *ровно n''* аргументов и готов принять вместо них ровно вызываемую подпрограмму *n'''* результатов.

Такие формы CALLX и RET в основном предназначены для библиотечных функций, которые принимают функциональные аргументы и хотят их безопасно вызывать. Другое приложение связано с «поддержкой виртуализации» TVM, которая позволяет коду TVM запускать другой код TVM внутри «виртуальной машины TVM». Такие методы виртуализации могут быть полезны для реализации сложных платежных каналов в блокчейне TON (см. [1, 5]).

4.1.11. CALLCC: вызов с текущим продолжением. Обратите внимание, что TVM поддерживает форму примитива "вызов с текущим продолжением". А именно, примитив CALLCC похож на CALLX или JMPX в том, что он берет продолжение c из стека и переключается на него; однако CALLCC не отбрасывает предыдущее текущее продолжение c' (как это делает JMPX) и не записывает c' в c0 (как это делает CALLX), а скорее толкает c' в стек (новый) в качестве дополнительного аргумента к c. Примитив JMPXDATA делает то же самое, но отправляет только (остаток) кода предыдущего текущего продолжения в виде *Slice*.

4.2 Примитивы потока управления: условное и итерированное выполнение

4.2.1. Условное исполнение: IF, IFNOT, IFELSE. Важная модификация EXECUTE (или CALLX) заключается в его условных формах. Например, IF принимает целое число x и продолжение c и выполняет c (так же как это сделал бы EXECUTE) только в том случае, если x не равен нулю; в противном случае оба значения просто отбрасываются из стека. Аналогично, IFNOT принимает x и c, но выполняет c только в том случае, если $x = 0$. Наконец, IFELSE принимает x, c и c', удаляет эти значения из стека и выполняет c, если $x \neq 0$ или c', если $x = 0$.

4.2.2. Итерированное выполнение и циклы. Более сложные модификации EXECUTE включают в себя:

- `REPEAT` – Принимает целое число *n* и продолжение *c* и выполняет *c n* раз.[23]
- WHILE — принимает *c'* и *c'', выполняет c', а затем принимает верхнее значение x* из стека. Если *x* не равен нулю, он выполняет *c''* , а затем начинает новый цикл, выполняя *c'* снова; если *x* равен нулю, он останавливается.
- UNTIL — принимает *c, выполняет его, а затем берет верхнее целое число x* из стека. Если *x* равно нулю, начинается новая итерация; если *x* не равен нулю, ранее выполненный код возобновляется.

4.2.3. Постоянные, или буквальные, продолжения. Мы видим, что можем создавать сколь угодно сложные условные выражения и циклы в коде TVM при условии, что у нас есть средства для проталкивания константных продолжений в стек. Фактически, TVM включает в себя специальные версии примитивов «литерала» или «константы», которые вырезают *следующие n* байтов или битов из оставшейся части текущего кода `cc.code` в срез ячейки, а затем помещают его в стек не как *Slice* (как это делает `PUSHSLICE`), а как простое *обычное продолжение* (в котором есть только `code` и `cp`).

Простейшим из этих примитивов является `PUSHCONT`, который имеет немедленный аргумент *n* , описывающий количество последующих байтов (в байт-ориентированной версии TVM) или битов, подлежащих преобразованию в простое продолжение. Другим примитивом является `PUSHREFCONT`, который удаляет первую ссылку на ячейку из текущего продолжения `cc.code`, преобразует ячейку, на которую ссылается, в фрагмент ячейки и, наконец, преобразует фрагмент ячейки в простое продолжение.

4.2.4. Константные продолжения в сочетании с условными или итерированными примитивами выполнения. Поскольку постоянные продолжения очень часто используются в качестве аргументов для условных или итерированных примитивов выполнения, объединенных версии этих примитивов (например, `IFCONT` или `UNTILREFCONT`) могут быть определены в будущей редакции TVM, которые объединяют

[23] Реализация `REPEAT` включает в себя экстраординарное продолжение, которое запоминает оставшееся количество итераций, тело цикла *c* и возвращаемое продолжение *c'*. (Последний термин представляет собой остальную часть тела функции, которая вызывала `REPEAT`, который обычно хранится в `c0` нового `cc`.)

PUSHCONT или PUSHREFCONT с другим примитивом. Если проверить полученный код, IFCONT очень похож на более привычную инструкцию «условно-ветвь-вперед».

4.3 Операции с продолжением

4.3.1. Продолжения непрозрачны. Обратите внимание, что все продолжения *непрозрачны*, по крайней мере, в текущей версии TVM, что означает, что нет никакого способа изменить продолжение или проверить его внутренние данные. Почти единственное использование продолжения состоит в том, чтобы подать его примитиву потока управления.

Хотя есть некоторые аргументы в пользу включения поддержки неопакетных продолжений в TVM (наряду с непрозрачными продолжениями, которые необходимы для виртуализации), текущая редакция не предлагает такой поддержки.

4.3.2. Разрешенные операции с продолжениями. Тем не менее, некоторые операции с непрозрачными продолжениями все еще возможны, главным образом потому, что они эквивалентны операциям типа «создать новое продолжение, которое сделает что-то особенное, а затем вызовет исходное продолжение». Разрешенные операции с продолжениями включают:

- Поместите одно или несколько значений в стек продолжения *c* (таким образом, создав частичное применение функции или закрытие).
- Установка сохраненного значения контрольного регистра c(i) внутри списка сохранения *c.save продолжения c.* Если значение для рассматриваемого контрольного регистра уже имеется, эта операция в автоматическом режиме ничего не выполняет.

4.3.3. Пример: операции с контрольными регистрами. TVM имеет некоторые примитивы для установки и проверки значений контрольных регистров. Наиболее важными из них являются PUSH $c(i)$ (выталкивает текущее значение $c(i)$ в стек) и POP $c(i)$ (задает значение $c(i)$ из стека, если предоставленное значение имеет правильный тип). Однако существует также модифицированная версия последней инструкции, называемая POPSAVE $c(i)$, которая сохраняет старое значение $c(i)$ (для $i > 0$) в продолжение при $c0$, как описано в 4.3.2 перед установкой нового значения.

4.3.4. Пример: установка количества аргументов для функции в ее коде. Примитив `LEAVEARGS` n демонстрирует другое применение продолжений в операции: он оставляет только верхние n значений текущего стека, а остаток перемещает в стек продолжения в `c0`.
Этот примитив позволяет вызываемой функции «возвращать» ненужные аргументы в стек вызывающего объекта, что полезно в некоторых ситуациях (например, связанных с обработкой исключений).

4.3.5. Логические схемы. Продолжение c можно рассматривать как фрагмент кода с двумя необязательными точками выхода, хранящимися в списке сохранения c: основной точкой выхода, заданной $c.c0$:= $c.save$(`c0`), и вспомогательной точкой выхода, заданной $c.c1$:= $c.save$(`c1`) . В случае выполнения продолжение выполняет любое действие, для которого оно было создано, а затем (обычно) передает управление основной точке выхода или, в некоторых случаях, вспомогательной точке выхода. Мы иногда говорим, что продолжение c с определенными точками выхода $c.c0$ и $c.c1$ является *продолжением с двумя выходами* или *булевой цепью*, особенно если выбор точки выхода зависит от какого-то внутренне проверяемого условия.
4.3.6. Композиция продолжений. Можно *составить* два продолжения c и c', просто установив $c.c0$ или $c.c1$ в $c0$. Это создает новое продолжение, обозначаемое $c \circ_0 c'$ или $c \circ_1 c'$, которое отличается от c в своем списке сохранений. (Напомним, что если список сохранений c уже имеет запись, соответствующую рассматриваемому контрольному регистру, такая операция незаметно ничего не делает, как описано в разделе 4.3.2).

Составляя продолжения, можно строить цепочки или другие графики, возможно, с циклами, представляющими поток управления. Фактически, результирующий график напоминает блок-схему, с булевыми схемами, соответствующими «узлам условий» (содержащим код, который будет передавать управление либо `c0`, либо `c1` в зависимости от некоторого условия), и продолжениями с одним выходом, соответствующими «узлам действия».

4.3.7. Основное продолжение композиции примитивов. Двумя основными примитивами для составления продолжений являются `COMPOS` (также известный как `SETCONT c0` и `BOOLAND`) и `COMPOSALT` (также известный как `SETCONT c1` и `BOOLOR`), которые берут c и c' из стека, устанавливают $c.c0$ или $c.c1$ в c' и возвращают полученный

результирующее продолжение $c'' = c \circ_0 c'$ или $c \circ_1 c'$. Все остальные операции по композиции продолжения могут быть выражены в терминах этих двух примитивов.

4.3.8. Продвинутое продолжение композиции примитивов. Однако TVM может составлять продолжения не только взятые из стека, но и взятые из `c0` или `c1`, или из текущего продолжения `cc`; аналогичным образом, результат может быть помещен в стек, сохранен в `c0` или `c1` или использован в качестве нового текущего продолжения (т. Е. Управление может быть передано ему). Кроме того, TBM может определять примитивы условной композиции, выполняя некоторые из вышеуказанных действий только в том случае, если целое значение, взятое из стека, не равно нулю.

Например, `EXECUTE` можно описать как cc ←*c*$\circ_0$*cc, с продолжением c,* взятым из исходного стека. Аналогично, `JMPX` — это cc ← *c, а* `RET` *(также известный как* `RETTRUE` *в контексте логической схемы) — это cc* ← c0. Другие интересные примитивы включают `THENRET` (c′ ← *c* $\circ_0$ c0) и `ATEXIT` (c0 ← *c* $\circ_0$ c0).

Наконец, некоторые «экспериментальные» примитивы также включают c1 и $\circ_1$. Например:

- RETALT или RETFALSE делает cc ← c1.
- Условные версии RET и RETALT также могут быть полезны: RETBOOL берет целое число *x* из стека и выполняет RETTRUE, если $x \neq 0$, RETFALSE в противном случае.
- INVERT делает c0 ↔ c1; если два продолжения в c0 и c1 представляют две ветви, которые мы должны выбрать в зависимости от некоторого логического выражения, INVERT отрицает это выражение на внешнем уровне.
- INVERTCONT выполняет *c.c0 ↔ c.c1 в продолжение c*, взятое из стека.
- Варианты ATEXIT включают ATEXITALT (c1 ← *c*$\circ_1$*c1) и SETEXITALT (c1* ← (c $\circ_0$ c0) $\circ_1$ c1).
- BOOLEVAL берет продолжение *c* из стека и делает cc ← (c $\circ_0$ (PUSH −1))$\circ_1$ (PUSH0) $\circ_0$ cc. Если *c* представляет собой логическую схему, чистый эффект заключается в том, чтобы оценить ее и протолкнуть либо −1, либо 0 в стек, прежде чем продолжить.

4.4 Продолжения как объекты

4.4.1. Представление объектов с помощью продолжений. Объектно-ориентированное программирование в стиле Smalltalk (или Objective C)

может быть реализовано с помощью продолжений. Для этого объект представлен специальным продолжением *o*. Если у него есть какие-либо поля данных, их можно хранить в стеке *o*, делая *o* частичным приложением (т.е. продолжением с непустым стеком).

Когда кто-то хочет вызвать метод m из o с аргументами x_1, x_2, ..., x_n, он выталкивает аргументы в стек, затем толкает магическое число, соответствующее методу m, а затем выполняет o, передавая $n+1$ *аргументы* (ср. 4.1.10). Затем *o* использует целое число m в верхней части стека для выбора ветви с требуемым методом и выполняет его. Если *o* нужно изменить свое состояние, он просто вычисляет новое продолжение *o'* того же рода (возможно, с тем же кодом, *что и o*, но с другим начальным стеком). Новое продолжение *o'* возвращается вызывающему объекту вместе с любыми другими возвращаемыми значениями, которые необходимо вернуть.

4.4.2. Сериализуемые объекты. Другой способ представления объектов в стиле Smalltalk в виде продолжений или даже деревьев ячеек заключается в использовании примитива JMPREFDATA (вариант JMPXDATA, ср. 4.1.11), который берет первую ссылку на ячейку из кода текущего продолжения, преобразует ячейку, на которую ссылается, в простое обычное продолжение и передает ей управление, сначала передавая оставшуюся часть текущего продолжения в виде *Slice.* в стек. Таким образом, объект может быть представлен ячейкой *o˜*, которая содержит JMPREFDATA в начале своих данных, и фактическим кодом объекта в первой ссылке (можно сказать, что первая ссылка ячейки *o˜* является *классом* объекта *o˜*). Остальные данные и ссылки этой ячейки будут использоваться для хранения полей объекта.

Такие объекты имеют преимущество в том, что они являются деревьями клеток, а не просто продолжениями, что означает, что они могут храниться в постоянном хранилище смарт-контракта TON.

4.4.3. Уникальные продолжения и возможности. Возможно, имеет смысл (в будущем пересмотре TVM) пометить некоторые продолжения как уникальные, что означает, что они не могут быть скопированы, даже с задержкой, путем увеличения их счетчика ссылок до значения, превышающего единицу. Если непрозрачное продолжение уникально, оно по сути становится *возможностью*, которая может быть либо использована его владельцем ровно один раз, либо передана кому-то другому.

Например, представьте себе продолжение, представляющее выходной поток на принтер (это пример продолжения, используемого в качестве объекта, см. 4.4.1). При вызове с одним целочисленным аргументом n это продолжение выводит символ с кодом n на принтер и возвращает новое продолжение того же рода, отражающее новое состояние потока. Очевидно, что копирование такого продолжения и параллельное использование двух копий приведет к некоторым непреднамеренным побочным эффектам; пометка его как уникального запретила бы такое неблагоприятное использование.

4.5 Обработка исключений

Обработка исключений TVM довольно проста и заключается в передаче контроля продолжению, хранящемуся в контрольном реестре `c2`.

4.5.1. Два аргумента обработчика исключений: параметр исключения и номер исключения. Каждое исключение характеризуется двумя аргументами: *номером исключения* (*Integer*) и параметром *exception* (любое значение, чаще всего нулевое *целое число*). Номера исключений 0–31 зарезервированы для TVM, в то время как все остальные номера исключений доступны для пользовательских исключений.

4.5.2. Примитивы для создания исключения. Существует несколько специальных примитивов, используемых для создания исключения. Самый общий из них, `THROWANY`, берет из стека два аргумента, *v* и $0 \le n < 2^{16}$, и выдает исключение с числом *n* и *значением v.* Существуют варианты этого примитива, которые предполагают, что *v* является нулевым целым числом, хранит *n как* литеральное значение и/или обусловлены целочисленным значением, взятым из стека. Определяемые пользователем исключения могут использовать произвольные значения в виде *v* (например, деревья ячеек), если это необходимо.

4.5.3. Исключения, генерируемые TVM. Конечно, некоторые исключения порождаются нормальными примитивами. Например, арифметическое исключение переполнения генерируется всякий раз, когда результат арифметической операции не вписывается в знаковое 257-битное целое число. В таких случаях аргументы исключения, *v* и *n*, определяются самим TVM.

4.5.4. Обработка исключений. Сама обработка исключений заключается в передаче управления обработчику исключений, т. е. продолжению, указанному в управляющем регистре `c2`, с v и n, предоставленными в качестве двух аргументов для этого продолжения, как если бы `JMP` в `c2` был запрошен с $n'' = 2$ аргументами (см. 4.1.7 и 4.1.6). Как следствие, v и n оказываются в верхней части стека обработчика исключений. Остальная часть старого стека отбрасывается.

Обратите внимание, что если продолжение в `c2` имеет значение c2 в своем списке сохранений, оно будет использоваться для настройки нового значения `c2` перед выполнением обработчика исключений. В частности, если обработчик исключений вызывает `МЕТОД THROWANY`, он повторно присваивает исходное исключение с восстановленным значением `c2`. Этот трюк позволяет обработчику исключений обрабатывать только некоторые исключения, а остальное передавать внешнему обработчику исключений.

4.5.5. Обработчик исключений по умолчанию. При создании экземпляра TVM `c2` содержит ссылку на «продолжение обработчика исключений по умолчанию», которое является `ec_fatal` экстраординарным продолжением (см. 4.1.5). Его выполнение приводит к завершению выполнения TVM, при этом аргументы v и n исключения возвращаются внешнему вызывающему абоненту. В контексте блокчейна TON n будет храниться как часть результата транзакции.

4.5.6. Примитив `TRY`. Примитив `TRY` можно использовать для реализации C++-подобной обработки исключений. Этот примитив принимает два продолжения, c и c'. Он сохраняет старое значение `c2` в списке сохранения c', устанавливает `c2` в c' и выполняет c так же, как `EXECUTE`, но дополнительно сохраняет старое значение `c2` в списке сохранения нового `c0`. Обычно используется версия примитива `TRY` с явным числом аргументов n'', переданных продолжению c.

Чистый результат примерно эквивалентен оператору C++ `try { `c` } catch(...) { `c'`}` .

4.5.7. Список предопределенных исключений. Предопределенные исключения TVM соответствуют номерам исключений n в диапазоне 0–31. К ним относятся:

- *Нормальное окончание* (n = 0) — никогда не должно генерироваться,

но это полезно для некоторых трюков.

- *Альтернативное окончание* (n = 1) — Опять же, никогда не должно генерироваться.
- *Недолив стека* (n = 2) — недостаточно аргументов в стеке для примитива.
- *Переполнение стека* (n = 3) — в стеке хранится больше значений, чем разрешено этой версией TVM.
- *Переполнение целочисленного числа* (n = 4) — Целое число не укладывается в $-2^{256} \leq x < 2^{256}$ или произошло деление на ноль.
- *Ошибка проверки диапазона* (n = 5) — целое число вне ожидаемого диапазона.
- *Недопустимый опкод* (n = 6) — инструкция или ее непосредственные аргументы не могут быть декодированы.
- *Ошибка проверки типа* (n = 7) — аргумент примитива имеет неправильный тип значения.
- *Переполнение ячеек* (n = 8) — ошибка в одном из примитивов сериализации.
- *Недолив ячейки* (n = 9) — ошибка десериализации.
- *Ошибка словаря* (n = 10) — ошибка при десериализации объекта словаря.
- *Неизвестная ошибка* (n = 11) — Неизвестная ошибка, может быть выдана пользовательскими программами.
- *Фатальная ошибка* (n = 12) — выдается TVM в ситуациях, которые считаются невозможными.
- *Из газа* (n = 13) — Выбрасывается TVM, когда оставшийся газ (g_r) становится отрицательным. Это исключение обычно не может быть поймано и приводит к немедленному прекращению TVM.

Большинство из этих исключений не имеют параметра (т.е. вместо этого используют нулевое целое число). Порядок проверки этих исключений описан ниже в разделе 4.5.8.

4.5.8. Порядок недолива стека, проверка типа и исключения проверки диапазона. Все примитивы TVM сначала проверяют, содержит ли стек необходимое количество аргументов, создавая исключение переполнения стека, если это не так. Только тогда проверяются теги типов аргументов и их диапазоны (например, если примитив ожидает, что аргумент не только

будет целым числом, но и будет находиться в диапазоне от 0 до 256), начиная со значения в верхней части стека (последний аргумент) и углубляясь в стек. Если тип аргумента неправильный, создается исключение для проверки типов; если тип правильный, но значение не попадает в ожидаемый диапазон, создается исключение проверки диапазона.

Некоторые примитивы принимают переменное количество аргументов в зависимости от значений некоторого небольшого фиксированного подмножества аргументов, расположенных в верхней части стека. В этом случае приведенная выше процедура сначала запускается для всех аргументов из этого небольшого подмножества. Затем он повторяется для остальных аргументов, как только их количество и типы будут определены из уже обработанных аргументов.

4.6 Функции, рекурсия и словари

4.6.1. Проблема рекурсии. Условные и итерированные примитивы выполнения, описанные в 4.2, наряду с безусловными примитивами ветвления, вызова и возврата, описанными в 4.1, позволяют реализовать более или менее произвольный код с вложенными циклами и условными выражениями, за одним заметным исключением: можно создавать новые постоянные продолжения только из частей текущего продолжения. (В частности, нельзя таким образом вызвать подпрограмму из самой себя.) Поэтому выполняемый код, т.е. текущее продолжение, постепенно становится все меньше и меньше.[24]

4.6.2. *Y*-комбинаторное решение: передать продолжение в качестве аргумента самому себе. Одним из способов решения проблемы рекурсии является передача копии продолжения, представляющего тело рекурсивной функции, в качестве дополнительного аргумента самой себе. Рассмотрим, например, следующий код для факториальной функции:

```
71     PUSHINT 1
```

[24] Важным моментом здесь является то, что дерево ячеек, представляющих программу TVM, не может иметь циклических ссылок, поэтому использование `CALLREF` вместе со ссылкой на ячейку выше по дереву не сработает.

```
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
```

Это примерно соответствует определению тела вспомогательной функции с тремя аргументами *n*, *x* и *f*, так что *body*(*n,x,f*) равно *x,* если $n < 2$ и *f*(*n – 1, nx, f*) в противном случае, затем вызывая *body*(*n,1,body*) для вычисления факториала *n*. Затем рекурсия реализуется с помощью конструкции `DUP`; `EXECUTE`, или `DUP`; `JMPX` в случае хвостовой рекурсии. Этот трюк эквивалентен применению *Y*-комбинатора к телу функции.

4.6.3. Вариант решения *Y*-комбинатора. Другой способ рекурсивного вычисления факториала, более точно следующий классическому рекурсивному определению

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

выглядит следующим образом:

```
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
```

Это определение факториальной функции на два байта короче предыдущего, но оно использует общую рекурсию вместо хвостовой рекурсии, поэтому ее нельзя легко преобразовать в цикл.

4.6.4. Сравнение: нерекурсивное определение факториальной функции. Кстати, нерекурсивное определение факториала с помощью цикла `REPEAT` также возможно, и оно намного короче обоих рекурсивных определений:

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

4.6.5. Несколько взаимно рекурсивных функций. Если у человека есть коллекция f_1, ..., f_n взаимно рекурсивных функций, можно использовать тот же трюк, передав всю коллекцию continuations $\{f_i\}$ в стек по мере роста

экстрана, это становится n аргументов для каждой из этих функций. Однако по мере *роста n* это становится все более и более громоздким, так как приходится переупорядочивать эти дополнительные аргументы в стеке для работы с аргументами «true», а затем отправлять их копии в верхнюю часть стека перед любым рекурсивным вызовом.

4.6.6. Объединение нескольких функций в один кортеж. Можно также объединить коллекцию продолжений, представляющих функции f_1, ..., f_n, в "кортеж" $\mathbf{f} := (f_1,\ldots,f_n)$ и передать этот кортеж как один элемент стека $\mathbf{f}$. Например, когда $n \leq 4$ каждая функция может быть представлена ячейкой f_i (вместе с деревом клеток, укорененных в этой клетке), а кортеж может быть представлен ячейкой f, которая имеет ссылки на составляющие ее ячейки f_i. Однако это привело бы к необходимости «распаковки» необходимого компонента из этого кортежа перед каждым рекурсивным вызовом.

4.6.7. Объединение нескольких функций в функцию селектора. Другой подход заключается в объединении нескольких функций $f_1, \ldots, f_n$ в одну «селекторную функцию» *f, которая принимает дополнительный аргумент i, 1 ≤ i ≤ n, из верхней части стека, и вызывает соответствующую функцию* f_i. *Стековые машины, такие как TVM, хорошо подходят для этого подхода, поскольку они не требуют, чтобы функции* f_i имели одинаковое количество и типы аргументов. Используя этот подход, нужно было бы передать только один дополнительный аргумент, *f, каждой из этих функций и поместить в стек дополнительный аргумент i* перед каждым рекурсивным вызовом *f,* чтобы выбрать правильную функцию для вызова.

4.6.8. Использование выделенного регистра для сохранения функции селектора. Однако, даже если мы используем один из двух предыдущих подходов для объединения всех функций в один дополнительный аргумент, передача этого аргумента всем взаимно рекурсивным функциям все равно довольно громоздка и требует множества дополнительных операций манипулирования стеком. Поскольку этот аргумент изменяется очень редко, можно использовать выделенный регистр, чтобы сохранить его и прозрачно передать его всем вызываемым функциям. Это подход, используемый TVM по умолчанию.

4.6.9. Специальный регистр `c3` для функции селектора. Фактически, TVM использует выделенный регистр `c3` для сохранения продолжения,

представляющего текущую или глобальную «селекторную функцию», которая может быть использована для вызова любой из семейства взаимно рекурсивных функций. Специальные примитивы CALL *nn* или CALLDICT *nn* (ср. A.8.7) эквивалентны PUSHINT *nn*; PUSH c3; EXECUTE, и аналогично JMP *nn* или JMPDICT *nn* эквивалентны PUSHINT *nn*; PUSH c3; JMPX. Таким образом, программа TVM, которая в конечном итоге представляет собой большую коллекцию взаимно рекурсивных функций, может инициализировать c3 с правильной функцией селектора, представляющей семейство всех функций в программе, а затем использовать CALL *nn* для вызова любой из этих функций по ее индексу (иногда также называемому *селектором* функции).

4.6.10. Инициализация c3. Программа TVM может инициализировать c3 с помощью инструкции POP c3 . Однако, поскольку это обычно самое первое действие, выполняемое программой (например, смарт-контрактом), TVM делает некоторые положения для автоматической инициализации c3. А именно, c3 инициализируется кодом (начальным значением cc) самой программы, а дополнительный ноль (или, в некоторых случаях, какое-то другое предопределенное число *s*) вставляется в стек перед выполнением программы. Это примерно эквивалентно вызову JMPDICT 0 (или JMPDICT *s*) в самом начале программы, т. е. функция с нулевым индексом фактически является функцией main() для программы.

4.6.11. Создание селекторных функций и операторов switch. TVM делает специальные положения для простой и краткой реализации функций селектора (которые обычно составляют верхний уровень программы TVM) или, в более общем плане, произвольных переключателей или операторов случая (которые также полезны в программах TVM). Наиболее важными примитивами, включенными для этой цели, являются IFBITJMP, IFNBITJMP, IFBITJMPREF и IFNBITJMPREF (см. A.8.2). Они эффективно позволяют объединять подпрограммы, хранящиеся либо в отдельных ячейках, либо в качестве подсрезов определенных ячеек, в двоичное дерево решений с решениями, принятыми в соответствии с указанными битами целого числа, переданными в верхней части стека.

Другой инструкцией, полезной для реализации типов сумм-продуктов, является PLDUZ (см. A.7.2). Эта инструкция предварительно загружает первые несколько битов *Slice* в целое *число*, которое впоследствии может быть проверено IFBITJMP и другими подобными инструкциями.

4.6.12. Альтернатива: использование хэш-карты для выбора правильной функции. Еще одной альтернативой является использование *Hashmap* (ср. 3.3) для хранения «коллекции» или «словаря» кода всех функций в программе и использование примитивов поиска хэш-карты (ср. A.10) для выбора кода требуемой функции, которая затем может быть BLESSed в продолжение (ср. A.8.5) и выполнена. Также доступны специальные комбинированные примитивы «поиск, благословение и выполнение», такие как DICTIGETJMP и DICTIGETEXEC (см. A.10.11). Этот подход может быть более эффективным для более крупных программ и операторов switch.

5 Кодовые страницы и кодировка инструкций

В этой главе описывается механизм кодовой страницы, который позволяет TVM быть гибким и расширяемым при сохранении обратной совместимости по отношению к ранее сгенерированному коду.

Мы также обсудим некоторые общие соображения о кодировках инструкций (применимых к произвольному машинному коду, а не только к TVM), а также последствия этих соображений для TVM и выбор, сделанный при разработке нулевой кодовой страницы TVM (экспериментальной). Сами кодировки инструкций представлены далее в Приложении A.

5.1 Кодовые страницы и взаимодействие различных версий TVM

Кодовые *страницы* являются важным механизмом обратной совместимости и будущих расширений TVM. Они обеспечивают прозрачное выполнение кода, написанного для различных редакций TVM, с прозрачным взаимодействием между экземплярами такого кода. Механизм кодовых страниц, однако, является общим и достаточно мощным, чтобы включить некоторые другие первоначально непреднамеренные приложения.

5.1.1. Кодовые страницы в продолжениях. Каждое обычное продолжение содержит 16-битное *кодовой страницы* поле `cp` (ср. 4.1.1), которое определяет кодовую страницу, которая будет использоваться для выполнения ее кода. Если продолжение создается `PUSHCONT` (ср. 4.2.3) или аналогичным примитивом, оно обычно наследует текущую кодовую страницу (т.е. кодовую страницу `cc`).[25]

5.1.2. Текущая кодовая страница. Текущая кодовая страница `cp` (ср. 1.4) является кодовой страницей текущего продолжения `cc`. Он определяет способ декодирования следующей инструкции из `cc.code`, оставшейся части кода текущего продолжения. После того, как инструкция была

[25] Это не совсем так. Более точным утверждением является то, что обычно кодовая страница вновь созданного продолжения является известной функцией текущей кодовой страницы.

декодирована и выполнена, она определяет следующее значение текущей кодовой страницы. В большинстве случаев текущую кодовую страницу оставляют без изменений.

С другой стороны, все примитивы, которые переключают текущее продолжение, загружают новое значение `cp` из нового текущего продолжения. Таким образом, весь код в продолжениях всегда интерпретируется именно так, как он должен был быть.

5.1.3. Разные версии TVM могут использовать разные кодовые страницы. Различные версии TVM могут использовать разные кодовые страницы для своего кода. Например, исходная версия TVM может использовать нулевую кодовую страницу. Более новая версия может использовать кодовую страницу, которая содержит все ранее определенные опкоды, а также некоторые вновь определенные, используя часть ранее неиспользуемого пространства опкодов. Последующая версия может использовать еще одну кодовую страницу и так далее.

Однако более новая версия TVM будет выполнять старый код для нулевой кодовой страницы точно так же, как и раньше. Если старый код содержал опкод, используемый для некоторых новых операций, которые не были определены в исходной версии TVM, он все равно будет генерировать недопустимое исключение опкода, поскольку новые операции отсутствуют в нулевой кодовой странице.

5.1.4. Изменение поведения старых операций. Новые кодовые страницы также могут изменять эффекты некоторых операций, присутствующих в старых кодовых страницах, сохраняя при этом их опкоды и мнемонику.

Например, представьте себе будущее 513-битное обновление TVM (заменяющее текущую 257-битную конструкцию). Он может использовать 513-битный *тип Integer* в тех же арифметических примитивах, что и раньше. Однако, хотя опкоды и инструкции в новой кодовой странице будут выглядеть точно так же, как старые, они будут работать по-другому, принимая 513-битные целочисленные аргументы и результаты. С другой стороны, во время выполнения одного и того же кода в нулевой кодовой странице новая машина будет генерировать исключения всякий раз, когда целые числа, используемые в арифметике и других примитивах, не

вписываются в 257 бит.[26] Таким образом, обновление не изменит поведение старого кода.

5.1.5. Улучшение кодировки инструкций. Другим применением кодовых страниц является изменение кодировок инструкций, отражающих улучшенные знания фактических частот таких инструкций в кодовой базе. В этом случае новая кодовая страница будет иметь точно такие же инструкции, как и старая, но с другими кодировками, потенциально разной длины. Например, можно создать экспериментальную версию первой версии TVM, используя (префикс) биткод вместо исходного байт-кода, с целью достижения более высокой плотности кода.

5.1.6. Создание контекстно-зависимых кодировок инструкций. Другим способом использования кодовых страниц для повышения плотности кода является использование нескольких кодовых страниц с различными подмножествами всего набора инструкций, определенных в каждом из них, или с определенным целым набором инструкций, но с различными кодировками длины для одних и тех же инструкций на разных кодовых страницах.

Представьте себе, например, кодовую страницу «манипулирования стеком», где примитивы манипулирования стеком имеют короткие кодировки за счет всех других операций, и кодовую страницу «обработки данных», где все остальные операции короче за счет операций манипулирования стеком. Если операции манипулирования стеком имеют тенденцию происходить одна за другой, мы можем автоматически переключиться на кодовую страницу «манипулирования стеком» после выполнения любой такой инструкции. Когда происходит инструкция по обработке данных, мы переключаемся обратно на кодовую страницу «обработка данных». Если условные вероятности класса следующей

[26] Это еще один важный механизм обратной совместимости. Все значения вновь добавленных типов, а также значения, принадлежащие расширенным исходным типам, которые не принадлежат исходным типам (например, 513-битные целые числа, которые не вписываются в 257 бит в приведенном выше примере), рассматриваются всеми инструкциями (за исключением инструкций по манипулированию стеком, которые естественно полиморфны, ср. 2.2.6) в старых кодовых страницах как «значения неправильного типа», и соответствующим образом создавать исключения для проверки типов.

инструкции в зависимости от класса предыдущей инструкции значительно отличаются от соответствующих безусловных вероятностей, то этот метод — автоматическое переключение в режим манипулирования стеком для перестановки стека с более короткими инструкциями, а затем переключение назад — может значительно улучшить плотность кода.

5.1.7. Использование кодовых страниц для флагов статуса и управления. Другое потенциальное применение нескольких кодовых страниц внутри одной и той же ревизии TVM заключается в переключении между несколькими кодовыми страницами в зависимости от результата выполнения некоторых инструкций.

Например, представьте себе версию TVM, которая использует две новые кодовые страницы, 2 и 3. Большинство операций не изменяют текущую кодовую страницу. Однако целочисленные операции сравнения переключаются на кодовую страницу 2, если условие ложно, и на кодовую страницу 3, если оно истинно. Кроме того, новая операция `?EXECUTE`, подобно `EXECUTE`, действительно будет эквивалентен `EXECUTE` в кодовой странице 3, но вместо этого будет `DROP` в кодовой странице 2. Такой трюк эффективно использует бит 0 текущей кодовой страницы в качестве флага состояния.

В качестве альтернативы можно создать пару кодовых страниц — скажем, 4 и 5 — которые отличаются только примитивами десериализации клеток. Например, в кодовой странице 4 они могут работать как раньше, в то время как в кодовой странице 5 они могут десериализовать данные не из начала *Slice*, а из его конца. Две новые инструкции — скажем, `CLD` и `STD` — могут быть использованы для переключения на кодовую страницу 4

или кодовая страница 5. Очевидно, что теперь мы описали флаг состояния, влияющий на выполнение некоторых инструкций определенным новым образом.

5.1.8. Установка кодовой страницы в самом коде. Для удобства мы резервируем некоторый опкод во всех кодовых страницах — скажем, `FF` n — для инструкции `SETCP` n, с n от 0 до 255 (ср. A.13). Затем, вставив такую инструкцию в самое начало (основную функцию) программы (например, смарт-контракта TON Blockchain) или библиотечной функции, мы можем гарантировать, что код всегда будет выполняться в предполагаемой кодовой странице.

5.2 Кодировка инструкций

В этом разделе обсуждаются общие принципы кодирования инструкций, действительные для всех кодовых страниц и всех версий TVM. Позже в 5.3 обсуждается выбор, сделанный для экспериментальной «нулевой кодовой страницы».

5.2.1. Инструкции кодируются двоичным префиксным кодом. Все полные инструкции (т.е. инструкции вместе со всеми их параметрами, такими как имена регистров стека $s(i)$ или других встроенных констант) кодовой страницы TVM кодируются *двоичным префиксным кодом*. Это означает, что (конечная) двоичная строка (т.е. битовая строка) соответствует каждой полной инструкции, таким образом, что двоичные строки, соответствующие различным полным инструкциям, не совпадают, и ни одна двоичная строка среди выбранного подмножества не является префиксом другой двоичной строки из этого подмножества.

5.2.2. Определение первой инструкции из потока кода. Как следствие этого метода кодирования, любая двоичная строка допускает не более одного префикса, который является кодировкой некоторой полной инструкции. В частности, код `cc.code` текущего продолжения (который является *Slice*, и, следовательно, bitstring вместе с некоторыми ссылками на ячейки) допускает не более одного такого префикса, который соответствует (однозначно определенной) инструкции, которую TVM выполнит первой. После выполнения этот префикс удаляется из кода

текущего продолжения, а следующая инструкция может быть декодирована.

5.2.3. Недопустимый опкод. Если префикс `cc.code` не кодирует действительную инструкцию в текущей кодовой странице, *создается недопустимое исключение опкода* (см. 4.5.7). Однако случай пустого `cc.code` рассматривается отдельно, как описано в 4.1.4 (точное поведение может зависеть от текущей кодовой страницы).

5.2.4. Особый случай: заполнение в конце кода. В качестве исключения из приведенного выше правила некоторые кодовые страницы могут принимать некоторые значения `cc.code`, которые слишком коротки, чтобы быть допустимыми кодировками инструкций, в качестве дополнительных вариантов `NOP`, таким образом эффективно используя для них ту же процедуру, что и для пустого `cc.code`. Такие битстрочки могут быть использованы для заполнения кода ближе к его концу.

Например, если двоичная строка `00000000` (т.е. `x00`, ср. 1.0.3) используется в кодовой странице для кодирования `NOP`, ее собственные префиксы не могут кодировать какие-либо инструкции. Таким образом, эта кодовая страница может принимать `0`, `00`, `000`, ..., `0000000` в качестве вариантов `NOP`, если это все, что осталось в `cc.code`, вместо того, чтобы генерировать недопустимое исключение opcode.

Такое заполнение может быть полезным, например, если примитив `PUSHCONT` (ср. 4.2.3) создает только продолжения с кодом, состоящим из интегрального числа байтов, но не все инструкции кодируются интегральным числом байтов.

5.2.5. TVM-код является биткодом, а не байт-кодом. Напомним, что TVM является бит-ориентированной машиной в том смысле, что ее *Cells (и Slices)* естественным образом рассматриваются как последовательности битов, а не только октетов (байтов), ср. 3.2.5. Поскольку код TVM также хранится в ячейках (см. 3.1.9 и 4.1.4), нет смысла использовать только битовые строки длины, кратной на восемь, в качестве кодировок полных инструкций. Другими словами, вообще говоря, код *TVM является биткодом, а не байт-кодом*.

Тем не менее, некоторые кодовые страницы (например, наша экспериментальная кодовая страница ноль) могут использовать байт-код

(то есть использовать только кодировки, состоящие из интегрального числа байтов) — либо для простоты, либо для простоты отладки и изучения дампов памяти (т. е. ячейка).[27]

5.2.6. Пространство Опкода используется полной инструкцией. Вспомним из теории кодирования, что длины битовых строк l_i, используемые в двоичном префиксном коде, удовлетворяют неравенству Крафта-Макмиллана $\sum_i 2^{-l_i} \leq 1.$. Это применимо, в частности, к (полной) кодировке инструкций, используемой кодовой страницей TVM. Мы говорим, что *конкретная полная инструкция* (или, точнее, *кодировка полной инструкции) использует часть* 2^{-l} *пространства опкода, если она закодирована строкой l-бита. Можно видеть, что все полные инструкции вместе используют не более 1 (т.е. «максимум все пространство опкода»).*

5.2.7. Пространство опкода, используемое инструкцией или классом инструкций. Приведенная выше терминология распространяется на инструкции (рассматриваемые со всеми допустимыми значениями их параметров) или даже классы инструкций (например, все арифметические инструкции). Мы говорим, что (неполная) инструкция, или класс инструкций, занимает часть α пространства опкода, если α — сумма частей пространства опкода, занятых всеми полными инструкциями, принадлежащими этому классу.

5.2.8. Пространство опкодов для байт-кодов. Полезное приближение приведенных выше определений выглядит следующим образом: Рассмотрим все 256 возможных значений для первого байта кодировки инструкции. Предположим, что k из этих значений соответствуют конкретной инструкции или классу инструкций, которые мы рассматриваем. Тогда эта инструкция или класс инструкций занимает приблизительно часть $k/256$ пространства опкода.

Это приближение показывает, почему все инструкции не могут занимать вместе больше, чем часть $256/256 = 1$ пространства опкода, по крайней мере, без ущерба для уникальности декодирования инструкций.

[27] Если дампы ячеек шестнадцатеричны, кодировки, состоящие из интегрального числа шестнадцатеричных цифр (т. е. имеющие длину, делимую на четыре бита), могут быть одинаково удобными.

5.2.9. Практически оптимальные кодировки. Теория кодирования говорит нам, что в оптимально плотном кодировании часть пространства опкода, используемая полной инструкцией (2^{-l}, если полная инструкция закодирована в l битах), должна быть примерно равна вероятности или частоте ее возникновения в реальных программах.[28] То же самое должно относиться как к (неполным) инструкциям или примитивам (т.е. общим инструкциям без заданных значений параметров), так и к классам инструкций.

5.2.10. Пример: примитивы манипулирования стеком. Например, если инструкции по манипулированию стеком составляют примерно половину всех инструкций в типичной программе TVM, следует выделить примерно половину пространства опкода для кодирования инструкций по манипулированию стеком. Для таких инструкций можно зарезервировать первые байты («опкоды») `0x00`–`0x7f`. Если четверть этих инструкций являются `XCHG`, имеет смысл зарезервировать 0x00-0x1f для `XCHG`. Точно так же, если половина всех `XCHG` включает верхнюю часть стека `s0`, имеет смысл использовать 0x00-0x0f для кодирования `XCHG s0,s`(i).

5.2.11. Простые кодировки инструкций. В большинстве случаев используются простые кодировки полных инструкций. Простые кодировки начинаются с фиксированного bitstring называется *опкодом* инструкции, за которым следуют, скажем, 4-битные поля, содержащие индексы i стековых регистров `s`(i), указанных в инструкции, за которыми следуют все остальные постоянные (литеральные, немедленные) параметры, включенные в полную инструкцию. Хотя простые кодировки могут быть не совсем оптимальными, они допускают краткие описания, и их декодирование и кодирование могут быть легко реализованы.

Если (универсальная) инструкция использует простую кодировку с l-битным опкодом, то инструкция будет использовать 2^{-l} часть пространства опкода. Это замечание может быть полезным для соображений, описанных в разделах 5.2.9 и 5.2.10.

[28] Обратите внимание, что важна вероятность возникновения в коде, а не вероятность выполнения. Инструкция, встречающаяся в теле цикла, выполненного миллион раз, по-прежнему засчитывается только один раз.

5.2.12. Дальнейшая оптимизация плотности кода: коды Хаффмана. Можно построить оптимально плотный двоичный код для набора всех полных инструкций при условии, что их вероятности или частоты в реальном коде известны. Это хорошо известный код Хаффмана (для заданного распределения вероятностей). Однако такой код был бы крайне бессистемным и трудным для декодирования.

5.2.13. Практическое обучение кодировкам. На практике кодировки инструкций, используемые в TVM и других виртуальных машинах, предлагают компромисс между плотностью кода и простотой кодирования и декодирования. Такой компромисс может быть достигнут путем выбора простых кодировок (ср. 5.2.11) для всех инструкций (возможно, с отдельными простыми кодировками для некоторых часто используемых вариантов, таких как `XCHG s0,s(`i`)` среди всех `XCHG s(`i`),s(`j`)`), и выделения пространства опкода для таких простых кодировок с использованием эвристики, описанной в 5.2.9 и 5.2.10; именно этот подход в настоящее время используется в TVM.

5.3Кодировка инструкций в нулевой кодовой странице

В этом разделе приведены подробные сведения об экспериментальной кодировке инструкций для нулевой кодовой страницы, описанной в другом месте этого документа (см. Приложение A) и используемой в предварительной тестовой версии TVM.

5.3.1. Возможность модернизации. Прежде всего, даже если эта предварительная версия каким-то образом попадет в производственную версию TON Blockchain, механизм кодовой страницы (ср. 5.1) позволяет нам вводить лучшие версии позже без ущерба для обратной совместимости. [29] Так что в то же время мы можем свободно экспериментировать.

[29] Обратите внимание, что любые изменения после запуска не могут быть сделаны в одностороннем порядке; скорее, они потребуют поддержки по меньшей мере двух третей валидаторов.

5.3.2. Выбор инструкций. Мы решили включить много «экспериментальных» и не строго необходимых инструкций в нулевую кодовую страницу только для того, чтобы увидеть, как они могут быть использованы в реальном коде. Например, у нас есть как базовые (ср. 2.2.1), так и составные (ср. 2.2.3) примитивы манипулирования стеком, а также некоторые «несистематические», такие как `ROT` (в основном заимствованные из Forth). Если такие примитивы используются редко, их включение просто растрачивает некоторую часть пространства опкода и делает кодировки других инструкций немного менее эффективными, что мы можем себе позволить на данном этапе развития TVM.

5.3.3. Использование экспериментальных инструкций. Некоторым из этих экспериментальных инструкций были присвоены довольно длинные опкоды, просто чтобы вместить больше из них в пространство опкодов. Не следует бояться использовать их только потому, что они длинные; если эти инструкции окажутся полезными, они получат более короткие опкоды в будущих редакциях. Ноль кодовой страницы не предназначен для тонкой настройки в этом отношении.

5.3.4. Выбор байт-кода. Мы решили использовать байт-код (т.е. использовать кодировки полных инструкций длин, делимых на восемь). Хотя это может не привести к оптимальной плотности кода, поскольку такое ограничение длины затрудняет сопоставление частей пространства опкода, используемых для кодирования инструкций с расчетными частотами этих инструкций в коде TVM (ср. 5.2.11 и 5.2.9), такой подход имеет свои преимущества: он допускает более простой декодер инструкций и упрощает отладку (см. 5.2.5).

В конце концов, у нас сейчас недостаточно данных об относительных частотах различных инструкций, поэтому наши оптимизации плотности кода, вероятно, будут очень приблизительными на данном этапе. Простота отладки и экспериментирования, а также простота реализации более важны на этом этапе.

5.3.5. Простые кодировки для всех инструкций. По тем же причинам мы решили использовать простые кодировки для всех инструкций (см. 5.2.11 и 5.2.13) с отдельными простыми кодировками для некоторых очень часто

используемых подключений, как описано в 5.2.13. Тем не менее, мы попытались распределить пространство опкода, используя эвристику, описанную в 5.2.9 и 5.2.10.

5.3.6. Отсутствие контекстно-зависимых кодировок. Эта версия TVM также не использует контекстно-зависимые кодировки (см. 5.1.6). Они могут быть добавлены на более позднем этапе, если это будет сочтено полезным.

5.3.7. Перечень всех инструкций. Список всех инструкций, доступных в нулевой кодовой странице, вместе с их кодировками и (в некоторых случаях) краткими описаниями можно найти в Приложении A.

-

-

Ссылки

[1] Н. Дуров, *Telegram Open Network*, 2017.

A Инструкции и опкоды

В этом приложении перечислены все инструкции, доступные в (экспериментальной) нулевой кодовой странице TVM, как описано в 5.3.

Перечислим инструкции в лексикографическом порядке опкода. Однако пространство опкодов распределено таким образом, чтобы все инструкции в каждой категории (например, арифметические примитивы) имели соседние опкоды. Итак, мы сначала перечислим ряд примитивов манипулирования стеком, затем константные примитивы, арифметические примитивы, примитивы сравнения, клеточные примитивы, примитивы продолжения, словарные примитивы и, наконец, примитивные примитивы, специфичные для приложения.

Мы используем шестнадцатеричную нотацию (ср. 1.0) для битовых строк. Регистры стека $\mathrm{s}(i)$ обычно имеют $0 \leq i \leq 15$, а i кодируется в 4-битном поле (или, в некоторых редких случаях, в 8-битном поле). Другие непосредственные параметры обычно 4-битные, 8-битные или переменной длины.

Нотация стека, описанная в подразделе 2.1.10, широко используется в настоящем добавлении.

A.1 Цены на газ

Цена газа для большинства примитивов равна *базовой цене газа*, вычисляемой как $P_b := 10 + b + 5\mathrm{r}$, где b — длина инструкции в битах, а r — количество ссылок на ячейки, включенных в инструкцию. Когда цена на газ инструкции отличается от этой базовой цены, она указывается в скобках после ее мнемоники либо как *(x),* что означает, что общая цена газа равна x, либо как *(+x),* что означает $P_b + x$. Помимо целочисленных констант, могут появляться следующие выражения:

- C_r — общая стоимость «чтения» ячеек (т. е. преобразования ссылок на ячейки в срезы ячеек). В настоящее время это равно 100 или 25 газовым единицам на ячейку в зависимости от того, впервые ли ячейка с этим хэшем «считывается» во время текущего запуска виртуальной машины или нет.
- L — Общая стоимость нагрузочных ячеек. Зависит от требуемого действия по загрузке.
- B_w — Общая стоимость создания новых *строителей. В настоящее*

время это равно 0 газовым единицам на одного застройщика.

- C_w — Общая стоимость создания новых *ячеек от Builders. В настоящее время составляет 500 газовых единиц на ячейку.*

По умолчанию цена газа инструкции равна $P := P_b + C_r + \mathrm{L} + \mathrm{B}_w + \mathrm{C}_w$.

A.2 Примитивы манипулирования стеком

Этот раздел включает как базовые (см. 2.2.1), так и составные (см. 2.2.3) примитивы манипулирования стеком, а также некоторые «несистематические». Некоторые примитивы манипуляции составным стеком, такие как `XCPU` или `XCHG2`, имеют ту же длину, что и эквивалентная последовательность более простых операций. Мы включили эти примитивы независимо от того, что они могут быть легко выделены более короткими опкодами в будущей ревизии TVM — или удалены навсегда.

Некоторые инструкции по манипулированию стеком имеют две мнемоники: одна Forthstyle (например, `-ROT`), другая соответствует обычным правилам для идентификаторов (например, `ROTREV`). Всякий раз, когда примитив манипуляции стеком (например, `PICK`) принимает целочисленный параметр n из стека, он должен находиться в диапазоне $0 \ldots 255$; в противном случае перед дальнейшими проверками возникает исключение проверки диапазона.

A.2.1. Базовые примитивы манипулирования стеком.

- 00 — NOP, ничего не делает.
- 01 — XCHG s1, также известный как SWAP.
- 0*i* — XCHG s(*i*) или XCHG s0,s(*i*), заменяет верхнюю часть стека на s(*i*), $1 \le i \le 15$.
- 10*ij* — XCHG s(*i*),s(*j*), $1 \le i < j \le 15$, развязки s(*i*) с s(*j*).
- 11*ii* — XCHG s0,s(*ii*), с $0 \le ii \le 255$.
- 1*i* — XCHG s1,s(*i*), $2 \le i \le 15$.
- 2*i* — PUSH s(*i*), $0 \le i \le 15$, толкает копию старого s(*i*) в стек.
- 20 — PUSH s0, также известный как DUP.
- 21 — PUSH s1, также известный как OVER.

- 3*i* — POP *s(i), 0 ≤ i* ≤ 15, помещает старое верхнее значение стека в старое s(*i*).
- 30 — POP s0, также известный как DROP, отбрасывает верхнее значение стека.
- 31 — POP s1, также известный как NIP.

A.2.2. Примитивы манипуляций со составным стеком. Параметры i, j и k следующих примитивов являются 4-разрядными целыми числами в диапазоне $0\ldots 15$.

- 4*ijk* — XCHG3 s(*i*),s(*j*),s(*k*), эквивалент XCHG s2,s(*i*); XCHG s1, s(*j*); XCHG s0,s(*k*), с 0 ≤ *i,j,k* ≤ 15.
- 50*ij* — XCHG2 s(*i*),s(*j*), эквивалент XCHG s1,s(*i*); XCHG s(*j*).
- 51*ij* — XCPU s(*i*),s(*j*), эквивалентный XCHG s(*i*); PUSH s(*j*).
- 52*ij* — PUXC s(*i*),s(*j* − 1), эквивалент PUSH s(*i*); SWAP; XCHG s(*j*).
- 53*ij* — PUSH2 s(*i*),s(*j*), эквивалент PUSH s(*i*); PUSH s(*j* + 1).
- 540*ijk* — XCHG3 s(*i*),s(*j*),s(*k*) (длинная форма).
- 541*ijk* — XC2PU s(*i*),s(*j*),s(*k*), эквивалентный XCHG2 s(*i*),s(*j*); PUSH s(*k*).
- 542*ijk* — XCPUXC s(*i*),s(*j*),s(*k*−1), эквивалент XCHG s1,s(*i*); PUXC s(*j*),s(*k* − 1).
- 543*ijk* — XCPU2 s(*i*),s(*j*),s(*k*), эквивалентный XCHG s(*i*); PUSH2 s(*j*), s(*k*).
- 544*ijk* — PUXC2 s(*i*),s(*j* −1),s(*k*−1), эквивалент PUSH s(*i*); XCHG s2; XCHG2 s(*j*),s(*k*).
- 545*ijk* — PUXCPU s(*i*),s(*j*−1),s(*k*−1), эквивалент PUXC s(*i*),s(*j*−1); PUSH s(*k*).
- 546*ijk* — PU2XC s(*i*),s(*j*−1),s(*k*−2), эквивалент PUSH s(*i*); SWAP; PUXC s(*j*),s(*k* − 1).
- 547*ijk* — PUSH3 s(*i*),s(*j*),s(*k*), эквивалент PUSH s(*i*); PUSH2 s(*j*+ 1),s(*k* + 1).
- 54C_ — не используется.

A.2.3. Экзотические примитивы манипулирования стеком.

- 55*ij* — BLKSWAP *i+1,j+1, переставляет два блока s(j+i+1)... s(j+1) и s(j)... s0, для 0 ≤ i,j* ≤ 15. Эквивалент REVERSE *i* + 1,*j* + 1; REVERSE *j* + 1,0;

REVERSE *i* + *j* + 2,0.

- 5513 — ROT2 или 2ROT (a b c d e f – *c d e f a b), вращает три самые верхние пары записей стека.*
- 550*i* — ROLL *i* + 1, поворачивает верхние *i* + 1 записи стека. Эквивалент BLKSWAP 1,*i* + 1.
- 55*i*0 — ROLLREV *i*+1 *или* -ROLL *i*+1*, поворачивает верхние входы стека i*+1 *в другую сторону. Эквивалент* BLKSWAP *i* + 1,1.
- 56*ii* — PUSH s(*ii*) для 0 ≤ *ii* ≤ 255.
- 57*ii* — POP s(*ii*) для 0 ≤ *ii* ≤ 255.
- 58 — ROT (*a b c – b c a), эквивалент* BLKSWAP 1,2 или XCHG2 s2,s1.
- 59 — ROTREV или -ROT (*a b c – c a b), эквивалентный* BLKSWAP 2,1 или XCHG2 s2,s2.
- 5A — SWAP2 или 2SWAP (*a b c d – c d a b),* эквивалентный BLKSWAP 2,2 или XCHG2 s3,s2.
- 5B — DROP2 или 2DROP (*a b* –), эквивалентно DROP; DROP.
- 5C — DUP2 или 2DUP (*a b — a b a b)*, эквивалент PUSH2 s1,s0.
- 5D — OVER2 или 2OVER (*a b c d – a b c d a b),* эквивалентно PUSH2 s3,s2.
- 5E*ij* — REVERSE *i* + 2,j, обратный порядок s(*j* + *i* + 1)... s(*j*) для 0 ≤ *i,j* ≤ 15; эквивалентно последовательности ⌊*i*/2⌋ + 1 XCHGs.
- 5F0*i* — BLKDROP *i,* эквивалент DROP выполняется *i* раз.
- 5F*ij* — BLKPUSH *i,j*, эквивалент PUSH s(j) выполняется *i раз,* 1 ≤ *i* ≤ 15, 0 ≤ *j* ≤ 15.
- 60 — PICK или PUSHX, выскакивает из стека целое число *i* , затем выполняет PUSH s(*i*).
- 61 — ROLLX, выскакивает целое число *i* из стека, затем выполняет BLKSWAP 1,*i*.
- 62 — -ROLLX или ROLLREVX, выскакивает целое число *i* из стека, затем выполняет BLKSWAP *i,*1.
- 63 — BLKSWX, выскакивает целые числа *i,j* из стека, затем выполняет

BLKSWAP *i,j.*

- 64 — REVX, выскакивает из стека целые числа *i,j*, затем выполняет REVERSE *i,j.*
- 65 — DROPX, выскакивает из стека целое число *i*, затем выполняет BLKDROP *i.*
- 66 — TUCK (*ab − bab), эквивалент* SWAP; OVER или в XCPU s1,s1.
- 67 — XCHGX, выскакивает из стека целое число *i*, затем выполняет XCHG s(*i*).
- 68 — DEPTH, раздвигает текущую глубину стека.
- 69 — CHKDEPTH, выскакивает из стека целое число *i*, затем проверяет, есть ли хотя бы *i*-элементы, генерируя исключение недолива стека в противном случае.
- 6A — ONLYTOPX, выскакивает из стека целое число *i*, а затем удаляет все, кроме верхних *i* элементов.
- 6B — ONLYX, выскакивает из стека целое число *i* , затем оставляет только нижние *i* элементы. Приблизительно эквивалентно DEPTH; SWAP; SUB; DROPX.
- 6C00–6C0F — зарезервирован для стековых операций.
- 6C*ij* — BLKDROP2 *i,j,* сбрасывает *i* стековые элементы под верхними элементами *j* , где $1 \le i \le 15$ и $0 \le j \le 15$. Эквивалент REVERSE *i + j,0;* BLKDROP *i;* REVERSE *j,0.*

A.3 Кортеж, список и примитивы Null

Кортежи — это упорядоченные коллекции, состоящие не более чем из 255 значений стека TVM произвольных типов (не обязательно одинаковых). Примитивы кортежей создают, изменяют и распаковывают кортежи; они манипулируют значениями произвольных типов в процессе, подобно примитивам стека. Мы не рекомендуем использовать кортежи более 15 элементов.

Когда *кортеж t* содержит элементы $x_1, ..., x_n$ (в указанном порядке), мы записываем $t = (x_1,...,x_n)$; число $n \ge 0$ — длина кортежа *t.* Он также обозначается как |t|. Кортежи длиной два называются парами, а кортежи длиной три — тройными.

Списки в стиле Lisp представлены с помощью пар, т.е. кортежей, состоящих ровно из двух элементов. Пустой список представлен значением *Null*, а непустой список представлен парой (h,t),где h — первый элемент списка, а t — его хвост.

A.3.1. *Пустые* примитивы. Следующие примитивы работают с (единственным) значением $\bot$ типа *Null, полезным для представления пустых списков, пустых ветвей двоичных деревьев и отсутствия значений в типах Maybe X* . Пустой *кортеж,* созданный NIL, мог бы быть использован для той же цели; однако *Null* более эффективен и стоит меньше газа.

- `6D` — `NULL` или `PUSHNULL` (– ⊥), отправляет единственное значение типа *Null*.
- `6E` — `ISNULL` (*x* – ?), проверяет, является ли x является *Null* и возвращает −1 или 0 соответственно.

A.3.2. Кортежные примитивы.

- 6F0*n* — TUPLE *n* (x_1 ... x_n – *t),* создает новый *кортеж t* = (x_1,...,x_n), содержащий *n* значений x_1, ..., x_n, где 0 ≤ *n* ≤ 15.
- 6F00 — NIL (– *t),* толкает единственный *кортеж t* = () нулевой длины.
- 6F01 — SINGLE (*x — t), создает синглтон t* := (*x*), т.е. *кортеж* длиной один.
- 6F02 — PAIR или CONS (*x y – t),* создает пару *t* := (*x,y*).
- 6F03 — TRIPLE (*x y z – t),* создает тройное *t* := (*x,y,z*).
- 6F1*k* — INDEX *k* (*t — x),* возвращает *k-й* элемент *кортежа t,* где 0 ≤ *k* ≤ 15. Другими словами, возвращает x_{k+1} , если *t* = (x_1,...,x_n). Если *k* ≥ *n,* создает исключение проверки диапазона.
- 6F10 — FIRST или CAR (*t – x),* возвращает первый элемент *кортежа.*
- 6F11 — SECOND или CDR (*t – y),* возвращает второй элемент кортежа.
- 6F12 — THIRD (*t — z),* возвращает третий элемент *кортежа.*
- 6F2*n* – UNTUPLE *n* (*t* – x_1 ... x_n), распаковывает *кортеж t* = (x_1,...,x_n) длиной, равной 0 ≤ *n* ≤ 15. Если *t* не является *кортежем*, если |*t*| ≠ *n,* создается исключение проверки типа.

- 6F21 — UNSINGLE (*t – x),* распаковывает синглетон *t* = (*x*).
- 6F22 — UNPAIR или UNCONS (*t – x y*), распаковывает пару *t* = (*x,y*).
- 6F23 — UNTRIPLE (*t – x y z),* распаковывает тройной *t* = (*x,y,z*).
- 6F3*k* — UNPACKFIRST *k* (*t* – x_1 ... x_k), распаковывает первые 0 ≤ *k* ≤ 15 элементов кортежа *t.* Если |*t*| < *k,* создает исключение проверки типов.
- 6F30 — CHKTUPLE (*t* –), проверяет, является ли *t* кортежем.
- 6F4*n* — EXPLODE *n* (*t* – x_1 ... x_m *m),* распаковывает *кортеж t* = (x_1,...,x_m) и возвращает его длину *m,* но только в том случае, если *m* ≤ *n* ≤ 15. В противном случае создается исключение проверки типов.
- 6F5*k* — SETINDEX *k* (*t x — t'),* вычисляет *кортеж t'*, который отличается от *t* только в положении, которое установлено в *x.* Другими словами, |*t'*| = |*t*|, $t'_i = t_i$ для $i \neq k + 1$, а $t'_{k+1} = x$ для заданного 0 ≤ *k* ≤ 15. Если *k* ≥ |*t*|, создает исключение проверки диапазона.
- 6F50 — SETFIRST (*t x – t'),* устанавливает первый компонент *Кортежа t* в *x* и возвращает полученный *Кортеж t'.*
- 6F51 — SETSECOND (*t x – t'),* устанавливает второй компонент *Кортежа t* в *x* и возвращает полученный *Кортеж t'.*
- 6F52 — SETTHIRD (*t x – t'),* устанавливает третий компонент *Кортежа t* в *x* и возвращает полученный *Кортеж t'.*
- 6F6*k* — INDEXQ *k* (*t – x),* возвращает *k-й* элемент *кортежа t,* где 0 ≤ *k* ≤ 15. Другими словами, возвращает x_{k+1} , если *t* = (x_1,...,x_n). Если *k* ≥ *n* или если *t* равно *Null,* возвращает значение *Null* вместо *x.*
- 6F7k — SETINDEXQ *k* (*t x — t'),* устанавливает *k-ю* составляющую *Кортежа t* в *x,* где 0 ≤ *k* < 16, и возвращает полученный *кортеж t'.* Если |*t*| ≤ *k,* сначала расширяет исходный *кортеж* до длины *k+1,* установив для всех новых компонентов значение *Null.* Если исходное значение *t* равно *Null,* обрабатывает его как пустой *кортеж.* Если *t* не имеет значения *Null* или *Tuple,* создается исключение. Если значение *x* равно *Null* и |*t*| ≤ *k* или *t* равно *Null,* то всегда возвращает *t'* = *t* (и не потребляет газ для создания кортежа).

- 6F80 — TUPLEVAR (x_1 ... x_n n – t), создает новый *кортеж* t длиной n аналогично TUPLE, но с $0 \le n \le 255$, взятый из стека.
- 6F81 — INDEXVAR (t k – x), аналогично INDEX k, но с $0 \le k \le 254$, взятыми из стека.
- 6F82 — UNTUPLEVAR (t n – x_1 ... x_n), аналогично UNTUPLE n, но с $0 \le n \le 255$, взятых из стека.
- 6F83 — UNPACKFIRSTVAR (t n – x_1 ... x_n), аналогично UNPACKFIRST n, но с $0 \le n \le 255$, взятых из стека.
- 6F84 — EXPLODEVAR (t n – x_1 ... x_m m), похожий на *EXPLODE n*, но с $0 \le n \le 255$, взятыми из стека.
- 6F85 — SETINDEXVAR (t x k — t'), аналогично *SETINDEX k*, но с $0 \le k \le 254$, взятыми из стека.
- 6F86 — INDEXVARQ (t k – x), аналогичный *INDEXQ n*, но с $0 \le k \le 254$, взятыми из стека.
- 6F87 — SETINDEXVARQ (t x k — t'), похожий на *SETINDEXQ k*, но с $0 \le k \le 254$, взятый из стека.
- 6F88 — TLEN (t – n), возвращает длину *кортежа.*
- 6F89 — QTLEN (t — n или −1), аналогично TLEN, но возвращает −1, если t не является *кортежем.*
- 6F8A — ISTUPLE (t – ?), возвращает −1 или 0 в зависимости от того, является ли t *Кортежем.*
- 6F8B — LAST (t — x), *возвращает последний элемент* $t_{|t|}$ непустого *кортежа t.*
- 6F8C — TPUSH или COMMA (t x — t'), добавляет значение x к *кортежу* $t = (x_1,...,x_n)$, но только если результирующий *кортеж* $t' = (x_1,...,x_n,x)$ имеет длину не более 255. В противном случае создается исключение проверки типов.
- 6F8D TPOP (t – t' x), отделяет последний элемент $x = x_n$ от непустого *кортежа* $t = (x_1,...,x_n)$ и возвращает как результирующий *кортеж* $t' = (x_1,...,x_{n-1})$, так и исходный последний элемент x.
- 6FA0 — NULLSWAPIF (x – x или ⊥ x), помещает значение *Null* под самым верхним целым числом x, но только если $x \neq 0$.

- 6FA1 — NULLSWAPIFNOT (x — *x* или ⊥ *x*), помещает значение *Null* под самым верхним целым числом *x,* но только если *x* = 0. Может использоваться для выравнивания стека после тихих примитивов, таких как PLDUXQ.
- 6FA2 — NULLROTRIF (*x y – x y* или ⊥ *x y*), толкает *Null* под второй записью стека сверху, но только если самое верхнее *целое число y* не равно нулю.
- 6FA3 — NULLROTRIFNOT (*x y – x y* или ⊥ *x y*), толкает *Null* под второй записью стека сверху, но только если самое *верхнее целое число y* равно нулю. Может использоваться для выравнивания стека после тихих примитивов, таких как LDUXQ.
- 6FA4 — NULLSWAPIF2 (*x — x* или ⊥ ⊥ *x*), помещает два *Null под самое верхнее целое число x*, но только если *x* ≠ 0. Эквивалент NULLSWAPIF; NULLSWAPIF.
- 6FA5 — NULLSWAPIFNOT2 (*x — x* или ⊥ ⊥ *x*), помещает два *Null* под самое верхнее *целое число x,* но только если *x* = 0. Эквивалент NULLSWAPIFNOT; NULLSWAPIFNOT.
- 6FA6 — NULLROTRIF2 (*x y — x y* или ⊥ ⊥ *x y*), толкает два *Nulls* под второй записью стека сверху, но только если самое верхнее *целое число y* не равно нулю. Эквивалент NULLROTRIF; NULLROTRIF.
- 6FA7 — NULLROTRIFNOT2 (*x y – x y* или ⊥ ⊥ *x y*), толкает два *Null* под вторую запись стека сверху, но только если самое верхнее *целое число y* равно нулю. Эквивалент NULLROTRIFNOT; NULLROTRIFNOT.
- 6FB*ij* — INDEX2 *i,j* (*t – x*), восстанавливает *x* = $(t_{i+1})_{j+1}$ за 0 ≤ *i,j* ≤ 3. Эквивалент INDEX *i;* INDEX *j.*
- 6FB4 — CADR (*t – x*), восстанавливает *x* = $(t_2)_1$.
- 6FB5 – CDDR (*t – x*), восстанавливает *x* = $(t_2)_2$.
- 6FE_*ijk* — INDEX3 *i,j,k* (*t – x*), восстанавливается за 0 ≤ *i,j,k* ≤ 3. Эквивалент INDEX2 *i,j;* INDEX *k.*
- 6FD4 — CADDR (*t – x*), восстанавливает *x* = $((t_2)_2)_1$.
- 6FD5 — CDDDR (t – *x*), восстанавливает *x* = $((t_2)_2)_2$.

A.4 Константные, или буквальные примитивы

Следующие примитивы вводят в стек один литерал (или безымянную константу) некоторого типа и диапазона, хранящийся как часть (непосредственный аргумент) инструкции. Поэтому, если немедленный аргумент отсутствует или слишком короткий, создается исключение «недопустимый или слишком короткий опкод» (код 6).

A.4.1. Целочисленные и логические константы.

- 7*i* — PUSHINT *x* с −5 ≤ *x* ≤ 10, выталкивает целое число *x* в стек; здесь *i* равно четырем битам x более низкого порядка (т.е. *i* = *x* mod 16).
- 70 — ZERO, FALSE или PUSHINT 0, толкает ноль.
- 71 — ONE или PUSHINT 1.
- 72 — TWO или PUSHINT 2.
- 7A — TEN или PUSHINT 10.
- 7F — TRUE или PUSHINT -1.
- 80*xx* — PUSHINT *xx* с −128 ≤ *xx* ≤ 127.
- 81*xxxx* — PUSHINT *xxxx* с −215 ≤ *xxxx* < 215 знаковое 16-битное целое число с большим порядковым числом байтов.
- 81FC18 — PUSHINT −1000.
- 82*lxxx* — PUSHINT *xxx,* где 5-бит 0 ≤ *l* ≤ 30 определяет длину *n* = 8*l* + 19 знакового большого порядкового числа *xxx.* Общая длина этой инструкции равна *l* + 4 байта или *n* + 13 = 8*l* + 32 бита.
- 821005F5E100 — PUSHINT 10^8.
- 83*xx* — PUSHPOW2 *xx* + 1, (тихо) толкает 2^{xx+1} за 0 ≤ *xx* ≤ 255.
- 83FF — PUSHNAN, толкает NaN.
- 84*xx* — PUSHPOW2DEC *xx* + 1, толкает $2^{xx+1} − 1$ за 0 ≤ *xx* ≤ 255.
- 85*xx* — PUSHNEGPOW2 *xx* + 1, толкает $−2^{xx+1}$ за 0 ≤ *xx* ≤ 255.
- 86, 87 — зарезервировано для целочисленных констант.

A.4.2. Постоянные срезы, продолжения, ячейки и ссылки. Большинство инструкций, перечисленных ниже, передают литеральные фрагменты, продолжения, ячейки и ссылки на ячейки, хранящиеся в качестве

непосредственных аргументов к инструкции. Поэтому, если немедленный аргумент отсутствует или слишком короткий, создается исключение «недопустимый или слишком короткий опкод» (код 6).

- 88 — PUSHREF, помещает первую ссылку cc.code в стек в виде *ячейки* (и удаляет эту ссылку из текущего продолжения).
- 89 — PUSHREFSLICE, похожий на PUSHREF, но преобразует ячейку в *Slice*.
- 8A — PUSHREFCONT, похожий на PUSHREFSLICE, но делающий из ячейки простое обычное *продолжение*.
- 8Bxsss — PUSHSLICE sss, толкает (префикс) подсрез cc.code, состоящий из его первых 8*x* + 4 бит и без ссылок (т.е. по существу битовой строки), где $0 \le x \le 15$. Предполагается тег завершения, означающий, что все конечные нули и последний двоичный файл (если он присутствует) удаляются из этой битовой строки. Если исходная битовая строка состоит только из нулей, будет отправлен пустой срез.
- 8B08 — PUSHSLICE x8_, толкает пустой срез (bitstring ' ').
- 8B04 — PUSHSLICE x4_, толкает битстру '0'.
- 8B0C — PUSHSLICE xC_, толкает битстру '1'.
- 8Crxxssss — PUSHSLICE ssss, толкает (префикс) подсрез cc.code, состоящий из его первых $1 \le r + 1 \le 4$ ссылок и до первых 8*xx* + 1 бит данных, с $0 \le xx \le 31$. Также предполагается наличие тега завершения.
- 8C01 эквивалентен PUSHREFSLICE.
- 8Drxxssssss — PUSHSLICE sssss, проталкивает подсрез cc.code, состоящий из $0 \le r \le 4$ ссылок и до 8*xx* + 6 бит данных, с $0 \le xx \le 127$. Предполагается наличие тега завершения.
- 8DE_ — неиспользуемый (зарезервированный).
- 8F_rxxcccc — PUSHCONT cccc, где *cccc* — простое обычное продолжение, сделанное из первых $0 \le r \le 3$ ссылок и первых $0 \le xx \le 127$ байт cc.code.
- 9xccc — PUSHCONT ccc, проталкивает продолжение *x-байта* в

течение $0 \le x \le 15$.

A.5 Арифметические примитивы

A.5.1. Сложение, вычитание, умножение.

- A0 — ADD ($x\ y - x + y$), складывает вместе два целых числа.
- A1 — SUB ($x\ y - x - y$).
- A2 — SUBR ($x\ y - y - x$), эквивалент SWAP; SUB.
- A3 — NEGATE ($x - -x$), эквивалентный MULCONST −1 или ZERO; SUBR. Обратите внимание, что он инициирует целочисленное исключение переполнения, если $x = -2^{256}$.
- A4 — INC ($x - x + 1$), эквивалент ADDCONST 1.
- A5 — DEC ($x — x - 1$), эквивалент 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.5.2. Разделение.

Общая кодировка операции `DIV`, `DIVMOD` или `MOD` — `A`*`9mscdf`* с необязательным предварительным умножением и опциональной заменой деления или умножения на сдвиг. Переменные одно- или двухбитные поля m, s, c, d и f выглядят следующим образом:

- $0 \le m \le 1$ — указывает, есть ли предварительное умножение (операция MULDIV и ее варианты), возможно, замененное сдвигом влево.
- $0 \le s \le 2$ — Указывает, было ли умножение или деление заменены сдвигами: $s = 0$ — без замены, $s = 1$ — деление заменено правым сдвигом, $s = 2$ — умножение заменено левым сдвигом (возможно только для $m = 1$).
- $0 \le c \le 1$ — указывает, существует ли постоянный однобайтовый аргумент *tt* для оператора shift (если $s \ne 0$). Для $s = 0$, $c = 0$. Если $c = 1$, то $0 \le tt \le 255$, а сдвиг выполняется $tt + 1$ бит. Если $s \ne 0$ и $c = 0$, то величина сдвига предоставляется инструкции в виде верхнего целого *числа* в диапазоне 0...256.

- $1 \le d \le 3$ — Указывает, какие результаты деления необходимы: 1 — только частное, 2 — только остаток, 3 — оба.
- $0 \le f \le 2$ — режим округления: 0 — этаж, 1 — ближайшее целое число, 2 — потолок (см. пункт 1.5.6).

Примеры:

- 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 ($x\ y - r$), где $q := \lfloor x/y \rfloor$, $r := x \bmod y := x - yq$.
- A90C — DIVMOD ($x\ y - q\ r$), где $q := \lfloor x/y \rfloor$, $r := x - yq$.
- A90D — DIVMODR ($x\ y - q'\ r'$), где $q' := \lfloor x/y + 1/2 \rfloor$, $r' := x - yq'$.
- A90E — DIVMODC ($x\ y - q''\ r''$), где $q'' := \lceil x/y \rceil$, $r'' := x - yq''$.
- A924 — то же, что и RSHIFT: ($x\ y$ — $\lfloor x \cdot 2^{-y} \rfloor$) для $0 \le y \le 256$.
- A934*tt* — то же, что и RSHIFT *tt* + 1: (x — $\lfloor x \cdot 2^{-tt-1} \rfloor$).
- A938*tt* — MODPOW2 *tt* + 1: ($x - x \bmod 2^{tt+1}$).
- A985 — MULDIVR ($x\ y\ z - q'$), где $q' = \lfloor xy/z + 1/2 \rfloor$.
- A98C — MULDIVMOD ($x\ y\ z - q\ r$), где $q := \lfloor x \cdot y/z \rfloor$, $r := x \cdot y \bmod z$ (то же самое, что */MOD в Forth).
- A9A4 — MULRSHIFT ($x\ y\ z - \lfloor xy \cdot 2^{-z} \rfloor$) для $0 \le z \le 256$.
- A9A5 — MULRSHIFTR ($x\ y\ z - \lfloor xy \cdot 2^{-z} + 1/2 \rfloor$) на $0 \le z \le 256$.
- A9B4*tt* — MULRSHIFT *tt* + 1 ($x\ y$ — $\lfloor xy \cdot 2^{-tt-1} \rfloor$).
- A9B5*tt* — MULRSHIFTR *tt* + 1 ($x\ y$ — $\lfloor xy \cdot 2^{-tt-1} + 1/2 \rfloor$).
- A9C4 — LSHIFTDIV ($x\ y\ z - \lfloor 2^z x/y \rfloor$) для $0 \le z \le 256$.
- A9C5 — LSHIFTDIVR ($x\ y\ z - \lfloor 2^z x/y + 1/2 \rfloor$) на $0 \le z \le 256$.
- A9D4*tt* — LSHIFTDIV *tt* + 1 ($x\ y - \lfloor 2^{tt+1} x/y \rfloor$).
- A9D5*tt* — LSHIFTDIVR *tt* + 1 ($x\ y - \lfloor 2^{tt+1} x/y + 1/2 \rfloor$).

Наиболее полезными из этих операций являются `DIV`, `DIVMOD`, `MOD`, `DIVR`, `DIVC`, `MODPOW2` *t* и `RSHIFTR` *t* (для целочисленной арифметики); и `MULDIVMOD`, `MULDIV`, `MULDIVR`, `LSHIFTDIVR` *t* и `MULRSHIFTR` *t* (для арифметики с фиксированной точкой).

A.5.3. Сдвиги, логические операции.

- AA*cc* — LSHIFT *cc* + 1 ($x - x \cdot 2^{cc+1}$), $0 \le cc \le 255$.
- AA00 — LSHIFT 1, эквивалентный MULCONST 2 или Forth's 2*.
- AB*cc* — RSHIFT *cc* + 1 ($x \mapsto \lfloor x \cdot 2^{-cc-1} \rfloor$), $0 \le cc \le 255$.
- 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 ($y - 2^y$), $0 \le y \le 1023$, эквивалент ONE; SWAP; LSHIFT.
- AF — зарезервировано.
- B0 — AND ($x\ y \mapsto x\&y$), побитовое «и» из двух знаковых целых чисел *x* и *y,* знаковое расширение до бесконечности.
- B1 — OR ($x\ y - x \vee y$), побитовое «или» из двух целых чисел.

  B2 — XOR ($x\ y - x \oplus y$), побитовое «xor» из двух целых чисел.
- B3 — NOT ($x \mapsto x \oplus -1 = -1 - x$), побитовое «не» целого числа.
- B4*cc* — FITS *cc* + 1 ($x \mapsto x$), проверяет, является ли *x* *cc* + 1-битным знаковым целым числом для $0 \le cc \le 255$ (т. е. является ли $-2^{cc} \le x < 2^{cc}$). Если нет, либо запускает целочисленное исключение переполнения, либо заменяет *x* на NaN (тихая версия).
- B400 — FITS 1 или CHKBOOL ($x \mapsto x$), проверяет, является ли *x* «логическим значением» (т.е. либо 0, либо -1).
- B5*cc* — UFITS *cc+1* ($x - x$), проверяет, является ли *x* целым числом *cc+1 бит без знака для* $0 \le cc \le 255$ (т.е. является ли $0 \le x < 2^{cc+1}$).
- B500 — UFITS 1 или CHKBIT, проверяет, является ли *x* двоичной цифрой (т.е. нулем или единицей).
- B600 — FITSX ($x\ c - x$), проверяет, является ли *x* целым числом со знаком *c*-бит для $0 \le c \le 1023$.

- B601 — UFITSX (*x c — x*), проверяет, является ли *x* целым числом без знака *c*-бита для $0 \le c \le 1023$.
- B602 — BITSIZE (*x — c*), вычисляет наименьший $c \ge 0$ таким образом, что *x* помещается в знаковое целое число *c-бита* ($-2^{c-1} \le c < 2^{c-1}$).
- B603 — UBITSIZE (*x – c*), вычисляет наименьшее $c \ge 0$ таким образом, что *x* помещается *в целое число без знака c-бита* ($0 \le x < 2^c$), или выдает исключение проверки диапазона.
- B608 — MIN (*x y – x* или *y*), вычисляет минимум два целых числа *x* и *y.*
- B609 — MAX (*x y – x* или *y*), вычисляет максимум двух целых чисел *x* и *y.*
- B60A — MINMAX или INTSORT2 (*x y – x y* или *y x*), сортирует два целых числа. Тихая версия этой операции возвращает два *NaN,* если какой-либо из аргументов является *NaNs.*
- B60B — ABS (*x — |x|*), вычисляет абсолютное значение целого числа *x.*

A.5.4. Тихие арифметические примитивы. Мы решили сделать все арифметические операции «нетиховыми» (сигнальными) по умолчанию, а их тихие аналоги создавать с помощью префикса. Такая кодировка, безусловно, неоптимальна. Пока неясно, следует ли это делать таким образом, или наоборот, делая все арифметические операции тихими по умолчанию, или тихим и нетиговым операциям следует давать опкоды равной длины; это может быть решено только на практике.

- B7xx — префикс QUIET, преобразующий любую арифметическую операцию в ее «тихий» вариант, обозначаемый префиксом Q к его мнемонике. Такие операции возвращают NaNs вместо того, чтобы создавать целые исключения переполнения, если результаты не вписываются в *целые числа или если один из их аргументов является NaN.* Обратите внимание, что это не распространяется на суммы сдвига и другие параметры, которые должны находиться в небольшом диапазоне (например, 0–1023). Также обратите внимание, что это не отключает исключения проверки типов, если указано значение типа, отличного *от Integer* .

- B7A0 — QADD (*x y – x + y*), всегда работает, если *x* и *y* являются целыми числами, но возвращает *NaN,* если сложение не может быть выполнено.
- B7A904 — QDIV (*x y* — ⌊*x*/*y*⌋), возвращает NaN, если *y* = 0, или если *y* = −1 и *x* = -2^{256}, или если любой из *x* или *y* является NaN.
- B7B0 — QAND (x y – *x&y*), побитовое «и» (аналогично AND), но возвращает NaN, если *x* или *y* является NaN, вместо того, чтобы выдавать целочисленное исключение переполнения. Однако, если один из аргументов равен нулю, а другой — NaN, результат равен нулю.
- B7B1 — QOR (*x y – x* ∨ *y*), побитовое «или». Если *x* = −1 или *y* = −1, результат всегда равен −1, даже если другим аргументом является NaN.
- B7B507 — QUFITS 8 (*x* — *x'*), проверяет, является ли *x* беззнаковым байтом (т. е. является ли 0 ≤ *x* < 28), и заменяет *x* NaN, если это не так; оставляет *x* нетронутым в противном случае (т. е. если *x* является беззнаковым байтом).

•

A.6 Сравнение примитивов

A.6.1. Сравнение целых чисел. Все целочисленные примитивы сравнения возвращают целое число 1 («true») или 0 («false»), чтобы указать результат сравнения. Мы не определяем их аналоги «булевой схемы», которые передавали бы управление `c0` или `c1` в зависимости от результата сравнения. При необходимости такие инструкции могут быть смоделированы с помощью `RETBOOL`.

Также доступны тихие версии целочисленных примитивов сравнения, закодированные с помощью префикса `QUIET` (`B7`). Если какое-либо из сравниваемых целых чисел является `NaNs`, `результатом спокойного сравнения также будет NaN` («неопределенный»), а не 1 («да») или 0 («нет»), что эффективно поддерживает троичную логику.

- B8 — SGN (*x* — sgn(*x*)), вычисляет знак целого числа *x: −1, если x* < 0, 0, если *x* = 0, 1, если *x* > 0.
- B9 — LESS (*x y* – *x* < *y*), возвращает −1, если *x* < *y,* 0 в противном случае.
- BA — EQUAL (*x y* — *x* = *y*), возвращает −1, если *x* = *y,* 0 в противном случае.
- BB — LEQ (*x y* – *x* ≤ *y*).
- BC — GREATER (*x y* – *x* > *y*).
- BD — NEQ (*x y* – *x* ≠ *y*), эквивалентно EQUAL; NOT.
- BE — GEQ (*x y* – *x* ≥ *y*), эквивалентный LESS; NOT.
- BF — CMP (*x y* — sgn(*x* − *y*)), вычисляет знак *x* − *y:* −1, если *x* < *y,* 0, если *x* = *y,* 1, если *x* > *y.* Здесь не может произойти целочисленное переполнение, если x или y не являются NaN.
- C0*yy* — EQINT *yy* (*x* – *x* = *yy*) для $-2^7 \le yy < 27$.
- C000 — ISZERO, проверяет, равно ли целое число нулю. Соответствует Forth 0=.
- C1*yy* — LESSINT *yy* (*x* – *x* < *yy*) для −27 ≤ *yy* < 27.
- C100 — ISNEG, проверяет, является ли целое число отрицательным.

Соответствует 0< Форта.

- C101 — ISNPOS, проверяет, является ли целое число неположительным.
- C2*yy* — GTINT *yy* (*x* – *x* > *yy*) для −27 ≤ *yy* < 27.
- C200 — ISPOS, проверяет, является ли целое число положительным. Соответствует 0> Форта.
- C2FF – ISNNEG проверяет, является ли целое число неотрицательным.
- C3*yy* — NEQINT *yy* (*x* – *x* ≠ *yy*) для −27 ≤ *yy* < 27.
- C4 — ISNAN (*x* – *x* = NaN), проверяет, является ли *x* NaN.
- C5 — CHKNAN (*x* — *x*), выдает исключение арифметического переполнения, если *x* является NaN.
- C6 — зарезервирован для целочисленного сравнения.

A.6.2. Прочее сравнение.

Большинство из этих примитивов «другого сравнения» фактически сравнивают части данных *Slices* как битовые строки.

- C700 — SEMPTY (s – *s* = ∅) проверяет является ли *фрагмент s* пустым (т.е. не содержит битов данных и ссылок на ячейки).
- C701 — SDEMPTY (*s* – *s* ≈ ∅) проверяет не содержит ли *Slice s* битов данных.
- C702 — SREMPTY (*s* – *r*(*s*) = 0), проверяет, не имеет ли *Slice s* ссылок.
- C703 — SDFIRST (*s* – s_0 = 1), проверяет, является ли первый бит *Slice s* единицей.
- C704 — SDLEXCMP (*s s'* – *c*), сравнивает данные s лексикографически с данными *s'*, возвращая −1, 0 или 1 в зависимости от результата.
- C705 — SDEQ (*s s'* – *s* ≈ *s'*), проверяет, совпадают ли части данных *s* и *s'* , эквивалентные SDLEXCMP; ISZERO.
- C708 — SDPFX (*s s'* – ?), проверяет, является ли *s* префиксом *s'.*
- C709 — SDPFXREV (*s s'* – ?), проверяет, является ли *s'* префиксом *s,* эквивалентным SWAP; SDPFX.
- C70A — SDPPFX (*s s'* – ?), проверяет, является ли *s* правильным префиксом *s'* (т.е. префиксом, отличным от *s').*

- C70B — SDPPFXREV (*s s’* – ?), проверяет, является ли *s’* правильным префиксом *s.*
- C70C — SDSFX (*s s’* – ?), проверяет, является ли *s* суффиксом *s’.*
- C70D — SDSFXREV (*s s’* – ?), проверяет, является ли *s’* суффиксом *s.*
- C70E — SDPSFX (*s s’* – ?), проверяет, является ли *s* правильным суффиксом *s’.*
- C70F — SDPSFXREV (*s s’* — ?), проверяет, является ли *s’* правильным суффиксом *s.*
- C710 — SDCNTLEAD0 (*s* – *n*), возвращает число начальных нулей в *s.*
- C711 — SDCNTLEAD1 (*s* – *n*), возвращает число ведущих в *s.*
- C712 — SDCNTTRAIL0 (*s* – *n*), возвращает количество конечных нулей в *s.*
- C713 — SDCNTTRAIL1 (*s* – *n*), возвращает количество замыкающих в *s.*

A.7 Ячейковыепримитивы

Ячейковые примитивы в основном являются либо *примитивами сериализации клеток*, которые работают с *Builders, либо примитивами десериализации клеток*, которые работают с *Slices.*

A.7.1. Примитивы сериализации клеток. Все эти примитивы сначала проверяют, достаточно ли места в построителе, и только потом проверяют диапазон сериализуемого значения.

- C8 — NEWC (– *b*), создает новый пустой *Builder.*
- C9 — ENDC (*b* – *c*), преобразует *Builder* в обычную Ячейку .
- CA*cc* — STI *cc* + 1 (*x b* — *b’*), сохраняет подписанное *cc* + 1-битное целое число *x* в *Builder b* для 0 ≤ *cc* ≤ 255, выдает исключение проверки диапазона, если *x* не вписывается в *cc* + 1 бит.
- CB*cc* — STU *cc* + 1 (*x b* — *b’*), сохраняет беззнаковое *cc* + 1-битное целое число *x* в *Builder b*. Во всем остальном он похож на STI.
- CC — STREF (*c b* – *b’*), хранит ссылку на Ячейку *c* в *Builder b.*
- CD — STBREFR или ENDCST (*b b’’* – *b*), эквивалентный ENDC; SWAP; STREF.
- CE — STSLICE (*s b* – *b’*), хранит *Slice s* в *Builder b.*
- CF00 STIX (*x b l* – *b’*), хранит знаковое *l*-битное целое число *x* в *b* для 0 ≤ *l* ≤ 257.
- CF01 — STUX (*x b l* – *b’*), хранит беззнаковое *l*-битное целое число *x* в

b для 0 ≤ *l* ≤ 256.

- CF02 — STIXR (*b x l — b'*), похожий на STIX, но с аргументами в другом порядке.
- CF03 — STUXR (*b x l – b'*), похожий на STUX, но с аргументами в другом порядке.
- CF04 — STIXQ (*x b l – x b f* или *b'* 0), тихая версия STIX. Если в *b* нет пробела, множества *b'* = *b* и *f* = −1. Если *x* не помещается в биты l, устанавливает *b'* = *b* и *f* = 1. Если операция выполнена успешно, *b'* является новым конструктором, а *f* = 0. Тем не менее, 0 ≤ *l* ≤ 257, с исключением проверки диапазона, если это не так.
- CF05 — STUXQ (*x b l – b' f*).
- CF06 — STIXRQ (*b x l – b x f* или *b'* 0).
- CF07 — STUXRQ (*b x l – b x f* или *b'* 0).
- CF08*cc* — более длинная версия STI *cc* + 1.
- CF09*cc* — более длинная версия STU *cc* + 1.
- CF0A*cc* — STIR *cc* + 1 (*b x – b'*), эквивалент SWAP; STI *cc* + 1.
- CF0B*cc* — STUR *cc* + 1 (*b x – b'*), эквивалент SWAP; STU *cc* + 1.
- CF0C*cc* — STIQ *cc* + 1 (*x b – x b f* или *b'* 0).
- CF0D*cc* — STUQ *cc* + 1 (*x b – x b f* или *b'* 0).
- CF0E*cc* — STIRQ *cc* + 1 (*b x – b x f* или *b'* 0).
- CF0F*cc* — STURQ *cc* + 1 (*b x – b x f* или *b'* 0).
- CF10 — более длинная версия STREF (*c b – b'*).
- CF11 — STBREF (*b' b – b''*), эквивалент SWAP; STBREFREV.
- CF12 — более длинная версия STSLICE (*s b — b'*).
- CF13 — STB (*b' b – b''*), добавляет все данные из *Builder b'* в *Builder b.*
- CF14 — STREFR (*b c – b'*).
- CF15 — STBREFR (*b b' – b''*), более длинная кодировка STBREFR.
- CF16 — STSLICER (*b s – b'*).
- CF17 — STBR (*b b' – b''*), объединяет двух *Builders*, эквивалентных SWAP; STB.
- CF18 — STREFQ (*c b – c b* −1 или *b'* 0).
- CF19 — STBREFQ (*b' b – b0 b* −1 или *b''* 0).
- CF1A — STSLICEQ (*s b – s b* −1 или *b'* 0).
- CF1B — STBQ (*b' b – b' b* −1 или *b''* 0).
- CF1C — STREFRQ (*b c – b c* −1 или *b'* 0).
- CF1D — STBREFRQ (*b b' – b b'* −1 или *b''* 0).
- CF1E — STSLICERQ (*b s – b s* −1 или *b''* 0).

- CF1F — STBRQ (*b b′ – b b′* −1 или *b″* 0).
- CF20 — STREFCONST, эквивалент PUSHREF; STREFR.
- CF21 — STREF2CONST, эквивалент STREFCONST; STREFCONST.
- CF23 — ENDXC (*b x – c*), если *x* ≠ 0, создает специальную или *экзотическую* ячейку (ср. 3.1.2) из *Builder b. Тип экзотической ячейки* должен храниться в первых 8 битах *b. Если x* = 0, это эквивалентно ENDC. В противном случае перед созданием экзотической ячейки выполняются некоторые проверки достоверности данных и ссылок *b* .
- CF28 — STILE4 (*x b – b′*), хранит 32-битное целое число с младшим порядком байтов.
- CF29 — STULE4 (*x b – b′*), хранит 32-разрядное целое число без знака с младшим порядком байтов.
- CF2A — STILE8 (*x b — b′*), хранит 64-битное целое число с младшим порядком байтов.
- CF2B — STULE8 (*x b — b′*), хранит 64-битное целое число без знака с младшим порядком байтов.
- •CF30 — BDEPTH (*b – x*), возвращает глубину *Builder b.* Если ссылки на ячейки не хранятся в *b,* то *x* = 0; в противном случае *x* — это единица плюс максимум глубин ячеек, упомянутых из *b.*
- CF31 — BBITS (*b – x*), возвращает количество битов данных, уже хранящихся в *Builder b.*
- CF32 — BREFS (*b – y*), возвращает количество ссылок на ячейки, уже хранящихся в *b.*
- CF33 — BBITREFS (*b – x y*), возвращает числа как битов данных, так и ссылок на ячейки в *b.*
- CF35 — BREMBITS (*b – x′*), возвращает количество битов данных, которые все еще могут храниться в *b.*
- CF36 — BREMREFS (*b – y′*).
- CF37 — BREMBITREFS (*b – x′ y′*).
- CF38*cc* — BCHKBITS *cc* + 1 (*b* –), проверяет, можно ли хранить *cc* + 1 бит в *b*, где 0 ≤ *cc* ≤ 255.
- CF39 — BCHKBITS (*b x* –), проверяет, могут ли *x* биты быть сохранены в *b, 0 ≤ x* ≤ 1023. Если в b нет места для *x* больше битов или если *x* не находится в диапазоне 0...1023, возникает исключение.
- CF3A — BCHKREFS (*b y* –), проверяет, могут ли ссылки *y* храниться в *b, 0 ≤ y* ≤ 7.

- CF3B — BCHKBITREFS (*b x y* –), проверяет, могут ли ссылки *x* битов и *y* храниться в *b,* 0 ≤ *x* ≤ 1023, 0 ≤ *y* ≤ 7.
- CF3C*cc* — BCHKBITSQ *cc* + 1 (*b* — ?), проверяет, можно ли хранить *cc* + 1 бит в *b*, где 0 ≤ *cc* ≤ 255.
- CF3D — BCHKBITSQ (*b x* – ?), проверяет, можно ли хранить *x* битов в *b,* 0 ≤ *x* ≤ 1023.
- CF3E — BCHKREFSQ (*b y* – ?), проверяет, могут ли ссылки *y* храниться в *b*, 0 ≤ *y* ≤ 7.
- CF3F — BCHKBITREFSQ (*b x y* — ?), проверяет, могут ли ссылки *x* bits и *y* храниться в *b,* 0 ≤ *x* ≤ 1023, 0 ≤ *y* ≤ 7.
- CF40 — STZEROES (*b n – b'*), хранит *n* двоичных нулей в *Builder b.*
- CF41 — STONES (*b n – b'*), хранит *n* бинарных в *Builder b.*
- CF42 — STSAME (*b n x – b'*), хранит *n* двоичных *x-ов* (0 ≤ *x* ≤ 1) в *Builder b.*
- CFC0_*xysss* — STSLICECONST *sss* (*b – b'*), хранит константу подсрезов *sss* , состоящую из 0 ≤ *x* ≤ 3 ссылок и до 8*y* + 1 бит данных, с 0 ≤ *y* ≤ 7. Предполагается бит завершения.
- CF81 — STSLICECONST '0' или STZERO (*b – b'*), хранит один двоичный ноль.
- CF83 — STSLICECONST '1' или STONE (*b – b'*), хранит один двоичный.
- CFA2 — эквивалент STREFCONST.
- CFA3 — почти эквивалент STSLICECONST '1'; STREFCONST.
- CFC2 — эквивалент STREF2CONST.
- CFE2 — STREF3CONST.

A.7.2. Примитивы десериализации клеток.

- D0 — CTOS (*c – s*), преобразует *ячейку* в фрагмент. Обратите внимание, что *c* должна быть либо обычной клеткой, либо экзотической клеткой (ср. 3.1.2), которая автоматически загружается, чтобы получить обычную ячейку *c', преобразованную в Slice* впоследствии.
- D1 — ENDS (*s* –), удаляет *Slice s* из стека и создает исключение, если он не пуст.
- D2*cc* — LDI *cc* + 1 (*s – x s'*), загружает (т.е. разбирает) знак *cc* + 1-битное целое число *x* из *Slice s, и возвращает остаток s* как *s'.*
- D3*cc* — LDU *cc* + 1 (*s — x s'*), загружает беззнаковое *cc* + 1-битное

целое число *x* из *Slice s.*

- D4 — LDREF (*s — c s'*), нагружает опорную ячейку *c* из *s.*
- D5 — LDREFRTOS (*s – s' s''*), эквивалент LDREF; SWAP; CTOS.
- D6*cc* — LDSLICE *cc*+1 (*s — s'' s'*), разрезает следующие *cc*+1 бит *s* в отдельный *Slice s''.*
- D700 — LDIX (*s l – x s'*), загружает знаковое *l*-бит (0 ≤ *l* ≤ 257) целое число *x* из *Slice s и возвращает остаток s* как *s'.*
- D701 — LDUX (*s l — x s'*), загружает беззнаковое *l*-битное целое число *x* из (первых *l* битов) *s*, с 0 ≤ *l* ≤ 256.
- D702 — PLDIX (*s l – x*), предварительно загружает знаковое *l*-битное целое число из *Slice s,* для 0 ≤ *l* ≤ 257.
- D703 — PLDUX (*s l – x*), предварительно загружает целое число *l*-бит без знака из *s,* для 0 ≤ *l* ≤ 256.
- D704 — LDIXQ (*s l — x s'* −1 или *s* 0), тихая версия LDIX: загружает знаковое *l-битное целое число из s* аналогично LDIX, но возвращает флаг успеха, равный −1 при успехе или до 0 при сбое (если *s* не имеет *l* битов), вместо того, чтобы выбрасывать исключение недостаточности ячейки.
- D705 — LDUXQ (*s l – x s'* −1 или *s* 0), тихая версия LDUX.
- D706 — PLDIXQ (*s l – x* −1 или 0), бесшумная версия PLDIX.
- D707 — PLDUXQ (*s l – x* −1 или 0), бесшумная версия PLDUX.
- D708*cc* — LDI *cc* + 1 (*s — x s'*), более длинная кодировка для LDI.
- D709*cc* — LDU *cc* + 1 (*s — x s'*), более длинная кодировка для LDU.
- D70A*cc* — PLDI *cc*+1 (*s – x*), предварительно загружает знаковое *cc*+1-битное целое число из *Slice c.*
- D70B*cc* — PLDU *cc* + 1 (*s – x*), предварительно загружает беззнаковое *cc* + 1-битное целое число из *s.*
- D70C*cc* — LDIQ *cc* + 1 (*s — x s'* −1 или *s* 0), тихая версия LDI.
- D70D*cc* — LDUQ *cc* + 1 (*s — x s'* −1 или *s* 0), бесшумная версия LDU.
- D70E*cc* — PLDIQ *cc* + 1 (*s — x* −1 или 0), тихая версия PLDI.
- D70F*cc* — PLDUQ *cc* + 1 (*s — x* −1 или 0), бесшумная версия PLDU.
- D714_*c* — PLDUZ 32(*c* + 1) (*s – s x*), предварительно загружает первые 32(*c* + 1) бит *Slice s* в целое число без знака *x,* за 0 ≤ *c* ≤ 7. Если *s* короче, чем необходимо, недостающие биты считаются равными нулю. Эта операция предназначена для использования вместе с IFBITJMP и аналогичными инструкциями.
- D718 — LDSLICEX (*s l – s'' s'*), загружает первые 0 ≤ *l* ≤ 1023 бита из

Slice s в отдельный *Slice s'', возвращая остаток s* как *s'*.

- D719 — PLDSLICEX (*s l – s''*), возвращает первые 0 ≤ *l* ≤ 1023 бит *s* как *s''*.
- D71A — LDSLICEXQ (*s l – s'' s'* −1 или *s* 0), тихая версия LDSLICEX.
- D71B — PLDSLICEXQ (*s l – s'* −1 или 0), тихая версия LDSLICEXQ.
- D71C*cc* — LDSLICE *cc* + 1 (*s — s'' s'*), более длинная кодировка для LDSLICE.
- D71D*cc* — PLDSLICE *cc* + 1 (*s — s''*), возвращает первые 0 < *cc* + 1 ≤ 256 бит *s* как *s''*.
- D71E*cc* — LDSLICEQ *cc* + 1 (*s — s'' s'* −1 или *s* 0), тихая версия LDSLICE.
- D71F*cc* — PLDSLICEQ *cc* + 1 (*s — s''* −1 или 0), бесшумная версия PLDSLICE.
- D720 — SDCUTFIRST (*s l – s'*), возвращает первые 0 ≤ *l* ≤ 1023 бит *c.* Это эквивалентно PLDSLICEX.
- D721 — SDSKIPFIRST (*s l — s'*), возвращает все, кроме первых 0 ≤ *l* ≤ 1023 бит *c.* Это эквивалентно LDSLICEX; NIP.
- D722 — SDCUTLAST (*s l – s'*), возвращает последние 0 ≤ *l* ≤ 1023 бит *c.*
- D723 — SDSKIPLAST (*s l — s'*), возвращает все, кроме последних 0 ≤ *l* ≤ 1023 бит *c.*
- D724 — SDSUBSTR (*s l l' – s'*), возвращает 0 ≤ *l'* ≤ 1023 бит *s* , начиная со смещения 0 ≤ *l* ≤ 1023, таким образом извлекая битовую подстроку из данных *s.*
- D726 SDBEGINSX (*s s' – s''*), проверяет, начинается ли *s* с (биты данных) *s'*, и удаляет *s'* из *s* при успешном завершении. При сбое возникает исключение десериализации ячейки. Примитивный SDPFXREV можно считать тихой версией SDBEGINSX.
- D727 — SDBEGINSXQ (*s s' — s''* −1 или *s* 0), тихая версия SDBEGINSX.
- D72A_*xsss* — SDBEGINS (*s — s''*), проверяет, начинается ли *s* с константы bitstring *sss* длиной 8*x* + 3 (с предполагаемым продолжением бита), где 0 ≤ *x* ≤ 127, и удаляет *sss* из *s* при успехе.
- D72802 — SDBEGINS '0' (*s – s''*), проверяет, начинается ли *s* с двоичного нуля.
- D72806 — SDBEGINS '1' (*s – s''*), проверяет, начинается ли *s* с двоичного.
- D72E_*xsss* — SDBEGINSQ (*s — s''* −1 или *s* 0), тихая версия SDBEGINS.
- D730 — SCUTFIRST (*s l r – s'*), возвращает первые 0 ≤ *l* ≤ 1023 бит и первые 0 ≤ *r* ≤ 4 ссылки *s.*

- D731 — SSKIPFIRST (*s l r – s’).*
- D732 — SCUTLAST (*s l r – s’*), возвращает последние 0 ≤ *l* ≤ 1023 бит данных и последние 0 ≤ *r* ≤ 4 ссылки *s.*
- D733 — SSKIPLAST (*s l r – s’*).
- D734 — SUBSLICE (*s l r l’ r’ – s’*), возвращает 0 ≤ *l’* ≤ 1023 бит и 0 ≤ *r’* ≤ 4 ссылки из *Slice s*, после пропуска первых 0 ≤ *l* ≤ 1023 бит и первых 0 ≤ *r* ≤ 4 ссылок.
- D736 — SPLIT (*s l r – s’ s’’*), разбивает первые 0 ≤ *l* ≤ 1023 бит данных и первые 0 ≤ *r* ≤ 4 ссылки из *s* в *s’*, возвращая оставшуюся часть *s* как *s’’.*
- D737 — SPLITQ (*s l r – s’ s’’* −1 или *s* 0), тихая версия SPLIT.
- D739 — XCTOS (*c – s* ?), превращает обычную или экзотическую клетку в *Slice,* как если бы это была обычная клетка. Возвращается флаг, указывающий, является ли *c* экзотическим. Если это так, его тип может быть позже десериализован от первых восьми битов *s.*
- D73A — XLOAD (*c – c’*), загружает экзотическую ячейку *c* и возвращает обычную ячейку *c’. Если c* уже обычный, ничего не делает. Если *c* не может быть загружен, создается исключение.
- D73B — XLOADQ (*c — c’* −1 или *c* 0), загружает экзотическую ячейку *c* как XLOAD, но возвращает 0 при сбое.
- D741 — SCHKBITS (*s l* –), проверяет, есть ли в Slice s хотя бы *l* битов данных. Если это не так, создает исключение десериализации ячейки (т.е. недостаточного потока ячеек).
- D742 — SCHKREFS (*s r* –), проверяет, есть ли хотя бы *r* ссылки в *Slice s.*
- D743 — SCHKBITREFS (*s l r* –), проверяет, есть ли в Slice s по крайней мере *l* битов данных и r ссылок *.*
- D745 — SCHKBITSQ (*s l* – ?), проверяет, есть ли в Slice s хотя бы *l* битов данных *.*
- D746 — SCHKREFSQ (*s r* – ?), проверяет, есть ли хотя бы *r* ссылки в *Slice s.*
- D747 — SCHKBITREFSQ (*s l r* – ?), проверяет, есть ли в хотя бы *l* битов данных и *r* ссылок в *Slice s.*
- D748 — PLDREFVAR (*s n – c*), возвращает *n-ю* ячейку ссылки *Slice s* для 0 ≤ *n* ≤ 3.
- D749 — SBITS (*s – l*), возвращает количество битов данных в *Slice s.*
- D74A — SREFS (*s – r*), возвращает количество ссылок в *Slice s.*
- D74B — SBITREFS (*s – l r*), возвращает как количество битов данных, так и количество ссылок в *s.*

- D74E_*n* — PLDREFIDX *n* (*s* – *c*), возвращает *n-ю* ссылку на ячейку *Slice s,* где 0 ≤ *n* ≤ 3.
- D74C — PLDREF (*s* – *c*), предварительно загружает первую ссылку на ячейку *Slice.*
- D750 — LDILE4 (*s* – *x s'*), загружает 32-битное целое число с младшим порядком байтов.
- D751 — LDULE4 (*s* – *x s'*), загружает 32-битное целое число без знака little-endian.
- D752 — LDILE8 (*s* – *x s'*), загружает 64-битное целое число с младшим порядком байтов.
- D753 — LDULE8 (*s* – *x s'*), загружает 64-битное целое число без знака с младшим порядком байтов.
- D754 — PLDILE4 (*s* – *x*), предварительно загружает 32-разрядное целое число с младшим порядком байтов.
- D755 — PLDULE4 (*s* – *x*), предварительно загружает 32-разрядное целое число без знака с младшим порядком байтов.
- D756 — PLDILE8 (*s* – *x*), предварительно загружает 64-битное целое число с младшим порядком байтов.
- D757 — PLDULE8 (*s* – *x*), предварительно загружает 64-битное целое число без знака с младшим порядком байтов.
- D758 — LDILE4Q (*s* — *x s'* −1 или *s* 0), тихо загружает 32-битное целое число с младшим порядком байтов.
- D759 — LDULE4Q (*s* — *x s'* −1 или *s* 0), тихо загружает 32-разрядное целое число без знака с младшим порядком байтов.
- D75A — LDILE8Q (*s* — *x s'* −1 или *s* 0), тихо загружает 64-битное целое число с младшим порядком байтов.
- D75B — LDULE8Q (*s* — *x s'* −1 или *s* 0), тихо загружает 64-битное целое число без знака с младшим порядком байтов.
- D75C — PLDILE4Q (*s* — *x* −1 или 0), тихо предварительно загружает 32-битное целое число с младшим порядком байтов.
- D75D — PLDULE4Q (*s* — *x* −1 или 0), тихо предварительно загружает 32-разрядное целое число без знака с младшим порядком байтов.
- D75E — PLDILE8Q (*s* — *x* −1 или 0), тихо предварительно загружает 64-битное целое число с младшим порядком байтов.
- D75F — PLDULE8Q (*s* — *x* −1 или 0), тихо предварительно загружает 64-битное целое число без знака с младшим порядком байтов.
- D760 — LDZEROES (*s* – *n s'*), возвращает число *n* начальных нулевых

битов в *s*, и удаляет эти биты из *s.*

- D761 LDONES (*s – n s'*), возвращает число *n* начальных битов в *s* и удаляет эти биты из *s.*
- D762 — LDSAME (*s x – n s'*), возвращает число *n* начальных битов, равное $0 \le x \le 1$ в *s*, и удаляет эти биты из *s.*
- D764 — SDEPTH (*s – x*), возвращает глубину *Slice s.* Если *s* не имеет ссылок, то *x* = 0; в противном случае *x* — это единица плюс максимум глубин ячеек, упоминаемых из *s.*
- D765 — CDEPTH (*c – x*), возвращает глубину ячейки *c.* Если *c* не имеет ссылок, то *x* = 0; в противном случае *x* — это единица плюс максимум глубин ячеек, упомянутых из *c. Если c* имеет значение *Null* вместо *Cell,* возвращает ноль.

A.8 Примитивы потока продолжения и управления

A.8.1. Безусловные примитивы потока управления.

- D8 — EXECUTE или CALLX (*c* –), *вызывает* или *выполняет* продолжение *c* (т.е. *cc* ← *c* $\circ_0$ *cc*).
- D9 — JMPX (*c* –), *прыжки* или передача управления в продолжение *c* (т.е. *cc* ← *c* $\circ_0$ c0, а точнее *cc* ← (c $\circ_0$ c0) $\circ_1$ c1). Остальная часть предыдущего текущего продолжения cc отбрасывается.
- DA*pr* — CALLXARGS *p,r* (*c* –), *вызывает* продолжение *c* с параметрами p и ожиданием возвращаемых значений r, $0 \le p \le 15$, $0 \le r \le 15$.
- DB0*p* — CALLXARGS *p,−1 (c* –), *вызывает* продолжение *c* с $0 \le p \le 15$ параметров, ожидая произвольное количество возвращаемых значений.
- DB1*p* — JMPXARGS *p* (*c* –), переходит к продолжению *c,* передавая только топ $0 \le p \le 15$ значений из текущего стека в него (остальная часть текущего стека отбрасывается).
- DB2*r* — RETARGS *r, возвращает* значение c0, с $0 \le r \le 15$ возвращаемыми значениями, взятыми из текущего стека.
- DB30 — RET или RETTRUE, *возвращается* к продолжению при c0 (т.е. выполняет cc ← c0). Остальная часть текущего продолжения cc отбрасывается. Приблизительно эквивалентно PUSH c0; JMPX.

- DB31 — RETALT или RETFALSE, возвращается к продолжению при c1 (т.е. cc ← c1). Приблизительно эквивалентно PUSH c1; JMPX.
- DB32 — BRANCH или RETBOOL (*f* –), выполняет RETTRUE, если целое число $f \neq 0$, или RETFALSE, если $f = 0$.
- DB34 — CALLCC (*c* –), *вызов с текущим продолжением,* передает управление *c*, проталкивая старое значение cc в стек *c* (вместо того, чтобы отбрасывать его или записывать в новый c0).
- DB35 — JMPXDATA (*c* –), аналогично CALLCC, но оставшаяся часть текущего продолжения (старое значение cc) преобразуется в *Slice* перед отправкой его в стек *c.*
- DB36*pr* — CALLCCARGS *p,r* (*c* –), аналогично CALLXARGS, но перемещает старое значение cc (вместе с верхним $0 \le p \le 15$ значениями из исходного стека) в стек вновь вызываемого продолжения *c, устанавливая cc.nargs на* $-1 \le r \le 14$.
- DB38 — CALLXVARARGS (*c p r* –), аналогично CALLXARGS, но принимает $-1 \le p,r \le 254$ из стека. Следующие три операции также берут *p* и *r* из стека, оба в диапазоне −1... 254.
- DB39 — RETVARARGS (*p r* –), аналогично RETARGS.
- DB3A — JMPXVARARGS (*c p r* –), аналогично JMPXARGS.
- DB3B — CALLCCVARARGS (*c p r* —), аналогично CALLCCARGS.
- DB3C — CALLREF, эквивалент PUSHREFCONT; CALLX.
- DB3D — JMPREF, эквивалент PUSHREFCONT; JMPX.
- DB3E — JMPREFDATA, эквивалент PUSHREFCONT; JMPXDATA.
- DB3F — RETDATA, эквивалент PUSH c0; JMPXDATA. Таким образом, оставшаяся часть текущего продолжения преобразуется в *Slice* и возвращается вызывающему объекту.

A.8.2. Условные примитивы потока управления.

- DC — IFRET (*f* –), выполняет RET, но только если целое число *f* не равно нулю. Если *f* является NaN, создается целочисленное исключение переполнения.
- DD — IFNOTRET (*f* –), выполняет RET, но только если целое число *f*

равно нулю.

- DE — IF (*f c* –), выполняет EXECUTE для *c* (т.е. *выполняет c), но* только если целое число *f* не равно нулю. В противном случае оба значения просто отбрасываются.
- DF — IFNOT (*f c* –), выполняет продолжение *c,* но только если целое число *f* равно нулю. В противном случае оба значения просто отбрасываются.
- E0 — IFJMP (*f c* –), переходит к *c* (аналогично JMPX), но только если *f* не равно нулю.
- E1 — IFNOTJMP (*f c* —), переходит к *c* (аналогично JMPX), но только если *f* равно нулю.
- E2 — IFELSE (*f c c'* –), если целое число *f* не равно нулю, выполняет *c,* в противном случае выполняет *c'*. Эквивалент CONDSELCHK; EXECUTE.
- E300 — IFREF (*f* –), эквивалент PUSHREFCONT; IF, с оптимизацией, что ссылка на ячейку фактически не загружается в *Slice* , а затем преобразуется в обычное *продолжение*, если *f* = 0. Аналогичные замечания относятся и к следующим трем примитивам.
- E301 — IFNOTREF (*f* –), эквивалент PUSHREFCONT; IFNOT.
- E302 — IFJMPREF (*f* –), эквивалент PUSHREFCONT; IFJMP.
- E303 — IFNOTJMPREF (*f* –), эквивалент PUSHREFCONT; IFNOTJMP.
- E304 — CONDSEL (*f x y — x* или *y*), если целое число *f* не равно нулю, возвращает *x,* в противном случае возвращает *y.* Обратите внимание, что проверки типов не выполняются для *x* и *y;* поэтому это больше похоже на условную операцию стека. Примерно эквивалентно ROT; ISZERO; INC; ROLLX; NIP.
- E305 — CONDSELCHK (*f x y – x* или *y*), то же самое, что и CONDSEL, но сначала проверяет, имеют ли *x* и *y* один и тот же тип.
- E308 — IFRETALT (*f* –), выполняет RETALT, если целое число *f* ≠ 0.
- E309 — IFNOTRETALT (*f* –), выполняет RETALT, если целое число *f* = 0.
- E30D IFREFELSE (*f c* –), эквивалент PUSHREFCONT; SWAP; IFELSE, с оптимизацией, что ссылка на ячейку фактически не загружается в

Slice , а затем преобразуется в обычное *продолжение*, если *f* = 0. Аналогичные замечания относятся и к следующим двум примитивам: *ячейки преобразуются в продолжения только при необходимости.*

- E30E — IFELSEREF (*f c* –), эквивалент PUSHREFCONT; IFELSE.
- E30F — IFREFELSEREF (*f* –), эквивалент PUSHREFCONT; PUSHREFCONT; IFELSE.
- E310–E31F — зарезервировано для циклов с операторами разрыва, см. A.8.3 ниже.
- E39_*n* — IFBITJMP *n* (*x c* – *x*), проверяет, установлен ли бит 0 ≤ *n* ≤ 31 в целое число *x,* и если да, то выполняет JMPX для продолжения *c.* Значение *x* остается в стеке.
- E3B_*n* — IFNBITJMP *n* (*x c* – *x*), переходит в *c,* если бит 0 ≤ *n* ≤ 31 не установлен в целое число *x.*
- E3D_*n* — IFBITJMPREF *n* (*x* – *x*), выполняет JMPREF, если бит 0 ≤ *n* ≤ 31 задан в целое число *x.*
- E3F_*n* — IFNBITJMPREF *n* (*x* – *x*), выполняет JMPREF, если бит 0 ≤ *n* ≤ 31 не установлен в целое число *x.*

A.8.3. Примитивы потока управления: циклы. Большинство перечисленных ниже примитивов цикла реализованы с помощью экстраординарных продолжений, таких как `ec_until` (см. 4.1.5), с телом цикла и исходным текущим продолжением `cc` , хранящимся в качестве аргументов к этому экстраординарному продолжению. Обычно строится подходящее экстраординарное продолжение, а затем сохраняется в файле продолжения цикла savelist как `c0`; после этого модифицированное продолжение тела цикла загружается в `cc` и выполняется обычным способом. Все эти примитивы цикла имеют `*BRK` версии, адаптированные для выхода из цикла; они дополнительно устанавливают `c1` на исходное текущее продолжение (или оригинальное `c0` для версий `* ENDBRK`) и сохраняют старый `c1` в список сохранения исходного текущего продолжения (или оригинального `c0` для версий `* ENDBRK`).

- E4 — REPEAT (*n c* –), выполняет продолжение *c n* раз, если целое число *n* неотрицательно. Если $n \geq 2^{31}$ или $n < -2^{31}$, создается исключение проверки диапазона. Обратите внимание, что RET

внутри кода *c* работает как продолжение, не в качестве перерыва. Следует использовать либо альтернативные (экспериментальные) циклы, либо альтернативный RETALT (вместе с SETEXITALT перед циклом), чтобы вырваться из цикла.

- E5 — REPEATEND (*n* –), аналогично REPEAT, но применяется к текущему продолжению cc.
- E6 — UNTIL (*c* –), не выполнит продолжение *c,* а затем выдаст целое число *x* из результирующего стека. Если *x* равно нулю, выполняет другую итерацию этого цикла. Фактическая реализация этого примитива включает в себя экстраординарное продолжение ec_until (см. 4.1.5) с его аргументами, установленными в теле цикла (продолжение *c*) и исходном текущем продолжении cc. Это экстраординарное продолжение затем сохраняется в списке сохранения с как *c*.c0, а затем выполняется измененное *c*. Другие примитивные циклы реализованы аналогичным образом с помощью подходящих экстраординарных продолжений.
- E7 — UNTILEND (–), аналогично UNTIL, но выполняет текущее продолжение cc в цикле. Когда условие выхода из цикла выполнено, выполняет RET.
- E8 — WHILE (*c' c* –), выполняет *c'* и выскакивает целое число *x* из результирующего стека. Если *x* равно нулю, существует цикл и передает управление исходному cc. Если *x* не равен нулю, выполняется *c,* а затем начинается новая итерация.
- E9 — WHILEEND (*c'* –), аналогично WHILE, но использует текущее продолжение cc в качестве тела цикла.
- EA — AGAIN (*c* –), аналогично REPEAT, но выполняет *c* бесконечно много раз. RET начинает только новую итерацию бесконечного цикла, которая может быть завершена только исключением, или RETALT (или явным JMPX).
- EB — AGAINEND (–), аналогично AGAIN, но выполнено относительно текущего продолжения cc.
- E314 — REPEATBRK (*n c* –), аналогично REPEAT, но также устанавливает c1 в исходный cc после сохранения старого значения

c1 в список сохранений исходного cc. Таким образом, RETALT может быть использован для прорыва из тела петли.

- E315 — REPEATENDBRK (*n* –), аналогично REPEATEND, но также устанавливает c1 в исходный c0 после сохранения старого значения c1 в список сохранений исходного c0. Эквивалент SAMEALTSAVE; REPEATEND.
- E316 — UNTILBRK (*c* –), аналогично UNTIL, но также изменяет c1 так же, как REPEATBRK.
- E317 — UNTILENDBRK (–), эквивалент SAMEALTSAVE; UNTILEND.
- E318 — WHILEBRK (*c' c* –), аналогично WHILE, но также модифицирует c1 так же, как REPEATBRK.
- E319 — WHILEENDBRK (*c* –), эквивалент SAMEALTSAVE; WHILEEND.
- E31A — AGAINBRK (*c* –), похожий на AGAIN, но также модифицирует c1 так же, как REPEATBRK.
- E31B — AGAINENDBRK (–), эквивалент SAMEALTSAVE; AGAINEND.

A.8.4. Манипулирование стеком продолжений.

- EC*rn* — SETCONTARGS *r,n* (x_1 x_2... x_r *c* – *c'*), аналогично SETCONTARGS *r,* но устанавливает *c.*nargs в конечный размер стека *c'* плюс *n.* Другими словами, преобразует *c* в *замыкающую* или *частично примененную функцию,* при этом $0 \le n \le 14$ аргументов отсутствуют.
- EC0*n* — SETNUMARGS *n* или SETCONTARGS 0,*n* (*c* — *c'*), устанавливает *c.*nargs в *n* плюс текущая глубина стека *c,* где $0 \le n \le 14$. Если *c.*nargs уже установлено на неотрицательное значение, ничего не делает.
- EC*r*F — SETCONTARGS *r* или SETCONTARGS *r,−1* (x_1 x_2... x_r *c* – *c'*), толкает $0 \le r \le 15$ значений x_1 ... x_r в стек (копию) продолжения *c,* начиная с x_1. Если конечная глубина стека с оказывается больше *c.*nargs, создается исключение переполнения стека.
- ED0*p* — RETURNARGS *p* (–), оставляет в текущем стеке только верхние значения $0 \le p \le 15$ (что-то похоже на ONLYTOPX), при этом все неиспользуемые нижние значения не отбрасываются, а сохраняются в продолжение c0 так же, как это делает SETCONTARGS.

- ED10 — RETURNVARARGS (p –), аналогично RETURNARGS, но с целым числом $0 \le p \le 255$, взятым из стека.
- ED11 SETCONTVARARGS ($x_1\ x_2 ...\ x_r\ c\ r\ n$ – c'), аналогично SETCONTARGS, но с $0 \le r \le 255$ и $-1 \le n \le 255$, взятыми из стека.
- ED12 — SETNUMVARARGS ($c\ n$ – c'), где $-1 \le n \le 255$. Если $n = -1$, эта операция ничего не делает ($c' = c$). В остальном его действие аналогично SETNUMARGS n, но с n, взятым из стека.

A.8.5. Создание простых продолжений и замыканий.

- ED1E — BLESS (s – c), преобразует *Slice* s в простое обычное продолжение c, $c.$code $= s$ и пустым стеком и списком сохранений.
- ED1F — BLESSVARARGS ($x_1 ...\ x_r\ s\ r\ n$ – c), эквивалентный ROT; BLESS; ROTREV; SETCONTVARARGS.
- EErn — BLESSARGS r,n ($x_1 ...\ x_r\ s$ – c), где $0 \le r \le 15$, $-1 \le n \le 14$, эквивалентно BLESS; SETCONTRAGS r,n. Значение n представлено внутри инструкции 4-битным целым числом n mod 16.
- EE0n — BLESSNUMARGS n или BLESSARGS 0,n (s – c), также преобразует *Slice* s в *Продолжение* c, но устанавливает $c.$nargs в $0 \le n \le 14$.

A.8.6. Операции со списками сохранений продолжения и контрольными регистрами.

- ED4i — PUSH c(i) или PUSHCTR c(i) (– x), толкает текущее значение контрольного регистра c(i). Если регистр элемента управления не поддерживается в текущей кодовой странице или не имеет значения, инициируется исключение.
- ED44 — PUSH c4 или PUSHROOT, отправляет ссылку на ячейку «глобального корня данных», тем самым обеспечивая доступ к постоянным данным смарт-контракта.
- ED5i — POP c(i) или POPCTR c(i) (x –), выскакивает из *стека значение* x и сохраняет его в управляющем регистре c(i), если оно поддерживается текущей кодовой страницей. Обратите внимание, что если регистр элементов управления принимает только значения определенного типа, может возникнуть исключение проверки типов.

- ED54 — POP c4 или POPROOT устанавливает ссылку на ячейку «глобального корня данных», что позволяет модифицировать постоянные данные смарт-контрактов.
- ED6*i* — SETCONT c(*i*) или SETCONTCTR c(*i*) (*x c – c'*), сохраняет *x* в списке сохранения продолжения *c* как c(*i*) и возвращает результирующее продолжение *c'*. Почти все операции с продолжениями могут быть выражены в терминах SETCONTCTR, POPCTR и PUSHCTR.
- ED7*i* — SETRETCTR c(*i*) (*x* –), эквивалентный PUSH c0; SETCONTCTR c(*i*); POP c0.
- ED8*i* — SETALTCTR c(*i*) (*x* –), эквивалентный PUSH c1; SETCONTCTR c(*i*); POP c0.
- ED9*i* — POPSAVE c(*i*) или POPCTRSAVE c(*i*) (*x* –), аналогично POP c(*i*), но также сохраняет старое значение c(*i*) в продолжение c0. Эквивалент (вплоть до исключений) SAVECTR c(*i*); POP c(*i*).
- EDA*i* — SAVE c(*i*) или SAVECTR c(*i*) (–), сохраняет текущее значение c(*i*) в список сохранений продолжения c0. Если запись для c(*i*) уже присутствует в списке сохранения c0, ничего не делается. Эквивалент PUSH c(*i*); SETRETCTR c(*i*).
- EDB*i* — SAVEALT c(*i*) или SAVEALTCTR c(*i*) (–), аналогично SAVE c(*i*), но сохраняет текущее значение c(*i*) в списке сохранения c1, а не c0.
- EDC*i* — SAVEBOTH c(*i*) или SAVEBOTHCTR c(*i*) (–), эквивалентный DUP; SAVE c(*i*); SAVEALT c(*i*).
- EDE0 — PUSHCTRX (*i* — *x*), похожий на PUSHCTR c(*i*), но с *i*, $0 \le i \le 255$, взятый из стека. Обратите внимание, что этот примитив является одним из немногих «экзотических» примитивов, которые не являются полиморфными, как примитивы манипулирования стеком, и в то же время не имеют четко определенных типов параметров и возвращаемых значений, поскольку тип *x* зависит от *i.*
- EDE1 — POPCTRX (*x i* –), аналогично POPCTR c(*i*), но с $0 \le i \le 255$ из стека.
- EDE2 — SETCONTCTRX (*x c i – c'*), аналогично SETCONTCTR c(*i*), но с $0 \le i \le 255$ из стека.

- EDF0 — COMPOS или BOOLAND (*c c′ — c′′*), вычисляет композицию $c \circ_0 c'$, которая имеет значение «выполнить *c,* и, в случае успеха, выполнить *c′»* (если *c* — логическая схема) или просто «выполнить *c,* то *c′».* Эквивалент SWAP; SETCONT c0.
- EDF1 — COMPOSALT или BOOLOR (*c c′ — c′′*), вычисляет альтернативную композицию $c \circ_1 c'$, которая имеет значение «выполнять *c,* и, если не успешно, выполнять *c′*» (если *c* — логическая схема). Эквивалент SWAP; SETCONT c1.
- EDF2 — COMPOSBOTH (*c c′ – c′′*), вычисляет $(c \circ_0 c') \circ_1 c'$, что имеет значение «вычислять логическую схему *c*, затем вычислять *c′*, независимо от результата *c*».
- EDF3 — ATEXIT (*c* –), наборы c0 ← $c \circ_0$ c0. Другими словами, *c* будет выполнен перед выходом из текущей подпрограммы.
- EDF4 — ATEXITALT (*c* –), наборы c1 ← $c \circ_1$ c1. Другими словами, *c* будет выполнен перед выходом из текущей подпрограммы по ее альтернативному обратному пути.
- EDF5 — SETEXITALT (*c* –), наборы c1 ← (c $\circ_0$ c0) $\circ_1$ c1. Таким образом, последующий RETALT сначала выполнит *c,* а затем передаст управление исходному c0. Это можно использовать, например, для выхода из вложенных циклов.
- EDF6 — THENRET (*c – c′*), вычисляет *c′* := $c \circ_0$ c0
- EDF7 — THENRETALT (*c – c′*), вычисляет *c′* := $c \circ_0$ c1
- EDF8 — INVERT (–), развязки c0 и c1.
- EDF9 — BOOLEVAL (*c* – ?), выполняет cc ← $c \circ_0$ ((PUSH — 1) $\circ_0$cc)) $\circ_1$

  ((PUSH0) $\circ_0$ cc). Если *c* представляет собой логическую схему, чистый эффект заключается в том, чтобы оценить ее и протолкнуть либо −1, либо 0 в стек, прежде чем продолжить.
- EDFA — SAMEALT (–), наборы $c_1 := c_0$. Эквивалент PUSH c0; POP c1.
- EDFB — SAMEALTSAVE (–), устанавливает $c_1 := c_0$, но сначала сохраняет старое значение c_1 в список сохранения c_0. Эквивалент SAVE c1; SAMEALT.

- EE*rn* — BLESSARGS *r,n* (x_1... x_r *s* – *c*), описанный в A.8.4.

A.8.7. Вызовы и переходы подпрограмм словаря.

- F0*n* — CALL *n* или CALLDICT *n* (– *n*), вызывает продолжение в c3, выталкивая в качестве аргумента целое число 0 ≤ *n* ≤ 255. Приблизительно эквивалентно PUSHINT *n;* PUSH c3; EXECUTE.
- F12_*n* — CALL *n* для 0 ≤ *n* < 214 (– *n*), кодировка CALL *n* для больших значений *n.*
- F16_*n* — JMP *n* или JMPDICT *n* (— *n*), переходит к продолжению в c3, выдвигая в качестве аргумента целое число 0 ≤ *n* < 214. Приблизительно эквивалентно PUSHINT *n;* PUSH c3; JMPX.
- F1A_*n* — PREPARE *n* или PREPAREDICT *n* (– *n c*), эквивалентно PUSHINT *n;* PUSH c3, для 0 ≤ *n* < 214. Таким образом, CALL *n* приблизительно эквивалентен PREPARE *n;* EXECUTE, а JMP *n* приблизительно эквивалентен PREPARE *n;* JMPX. Здесь можно использовать, например, CALLARGS или CALLCC вместо EXECUTE.

A.9 Создание исключений и обработка примитивов

A.9.1. Создание исключений.

- F22_*nn* — THROW *nn* (– 0 *nn*), выдает исключение 0 ≤ *nn* ≤ 63 с нулевым параметром. Другими словами, он передает контроль продолжению в c2, вталкивая 0 и *nn* в свой стек и полностью отбрасывая старый стек.
- F26_*nn* — THROWIF *nn* (*f* –), выдает исключение 0 ≤ *nn* ≤ 63 с нулевым параметром только в том случае, если целое число *f* ≠ 0.
- F2A_*nn* — THROWIFNOT *nn* (*f* –), выдает исключение 0 ≤ *nn* ≤ 63 с нулевым параметром только в том случае, если целое число *f* = 0.
- F2C4_*nn* — THROW *nn* для 0 ≤ *nn* < 2^{11}, кодировка THROW *nn* для больших значений *nn.*
- F2CC_*nn* — THROWARG *nn* (*x* – *x nn*), выдает исключение 0 ≤ *nn* < 2^{11} с параметром *x,* копируя *x* и *nn* в стек c2 и передавая управление c2.
- F2D4_*nn* — THROWIF *nn* (*f* –) за 0 ≤ *nn* < 211.
- F2DC_*nn* — THROWARGIF *nn* (*x f* –), выдает исключение 0 ≤ *nn* < 2^{11} с

параметром *x* только в том случае, если целое число *f* ≠ 0.

- F2E4_*nn* — THROWIFNOT *nn* (*f* –) за 0 ≤ *nn* < 211.
- F2EC_*nn* — THROWARGIFNOT *nn* (*x f* –), выдает исключение 0 ≤ *nn* < 2^{11} с параметром *x* только в том случае, если целое число *f* = 0.
- F2F0 THROWANY (*n* – 0 *n), выдает исключение 0 ≤ n* < 2^{16} с нулевым параметром. Приблизительно эквивалентно PUSHINT 0; SWAP; THROWARGANY.
- F2F1 — THROWARGANY (*x n* – *x n*), выдает исключение 0 ≤ *n* < 216 с параметром *x,* передавая управление продолжению в c2. Приблизительно эквивалентно PUSH c2; JMPXARGS 2.
- F2F2 — THROWANYIF (*n f* –), выдает исключение 0 ≤ *n* < 216 с нулевым параметром только в том случае, если *f* ≠ 0.
- F2F3 — THROWARGANYIF (*x n f* –), выдает исключение 0 ≤ *n* < 216 с параметром *x* только в том случае, если *f* ≠ 0.
- F2F4 — THROWANYIFNOT (*n f* –), выдает исключение 0 ≤ *n* < 216 с нулевым параметром только в том случае, если *f* = 0.
- F2F5 — THROWARGANYIFNOT (*x n f* –), выдает исключение 0 ≤ *n* < 216 с параметром *x* только в том случае, если *f* = 0.

A.9.2. Перехват и обработка исключений.

- F2FF — TRY (*c c'* –), устанавливает c2 в *c'*, сначала сохраняя старое значение c2 как в список сохранений *c'*, так и в список сохранений текущего продолжения, который хранится в *c*.c0 *u c'*.c0. Затем выполняется с аналогично EXECUTE. Если *c* не создает исключений, исходное значение c2 автоматически восстанавливается при возврате из *c.* При возникновении исключения выполнение передается в *c'*, но исходное значение c2 восстанавливается в процессе, так что *c'* может повторно вызвать исключение с помощью THROWANY, если он не может обработать его самостоятельно.
- F3*pr* — TRYARGS *p,r* (*c c'* –), аналогично TRY, но с CALLARGS *p,r* внутренне используется вместо EXECUTE. Таким образом, все, кроме верхних 0 ≤ *p* ≤ 15 элементов стека, будут сохранены в стеке текущего продолжения, а затем восстановлены при возврате из *c* или *c'*,

причем верхние $0 \le r \le 15$ значений результирующего стека c или c' скопированы в качестве возвращаемых значений.

A.10 Примитивы манипуляций со словарем

Поддержка словаря TVM подробно обсуждается в 3.3. Основные операции со словарями перечислены в 3.3.10, а таксономия примитивов словарных манипуляций приведена в 3.3.11. Здесь мы используем понятия и обозначения, представленные в этих разделах.

Словари допускают два различных представления в качестве значений стека TVM:

- Slice *s* с сериализацией значения TL-B типа *HashmapE(n,X).* Другими словами, *s* состоит либо из одного бита, равного нулю (если словарь пуст), либо из одного бита, равного единице, и ссылки на ячейку, содержащую корень двоичного дерева, т. е. сериализованное значение типа *Hashmap(n,X).*
- Значение "maybe *Cell*" $c^?$, т.е. значение, которое является либо *Cell* (содержащим сериализованное значение типа *Hashmap(n,X),* как и раньше), либо *Null* (соответствует пустому словарю). Когда «может быть, *клетка»* $c^?$ используется для представления словаря, мы обычно обозначаем его *D* в стековой нотации.

Большинство словарных примитивов, перечисленных ниже, принимают и возвращают словари во второй форме, что более удобно для работы со стеком. Однако сериализованные словари внутри более крупных объектов TL-B используют первое представление.

Опкоды, начинающиеся с `F4` и `F5` , зарезервированы для словарных операций.

A.10.1. Создание словаря.

- 6D — NEWDICT (– *D*), возвращает новый пустой словарь. Это альтернативная мнемоника для PUSHNULL, ср. A.3.1.
- 6E — DICTEMPTY (*D* — ?), проверяет, пуст ли словарь *D* , и возвращает −1 или 0 соответственно. Это альтернативная мнемоника для ISNULL, ср. A.3.1.

A.10.2. Сериализация и десериализация словаря.

- CE — STDICTS (*s b – b'*), хранит в *Builder b* словарь, представленный *Slice* . На самом деле это синоним STSLICE.
- F400 — STDICT или STOPTREF (*D b – b'*), сохраняет словарь *D* в *Builder b,* восстанавливая полученный *Builder b'*. Другими словами, если *D* является ячейкой, выполняет STONE и STREF; если *D* имеет значение *Null,* выполняет NIP и STZERO; в противном случае создает исключение проверки типов.
- F401 — SKIPDICT или SKIPOPTREF (*s – s'*), эквивалентный LDDICT; NIP.
- F402 LDDICTS (*s – s' s''*), загружает (разбирает) словарь *(Slice-represented) s'* из *Slice s* и возвращает остаток *s* как *s''*. Это "разделенная функция" для всех типов словарей *HashmapE(n,X).*
- F403 — PLDDICTS (*s – s'*), предварительно загружает словарь *(Slice-represented) s'* из *Slice s.* Приблизительно эквивалент LDDICTS; DROP.
- F404 — LDDICT или LDOPTREF (*s — D s'*), загружает (анализирует) словарь *D* из *Slice s* и возвращает остаток *s* как *s'*. Может применяться к словарям или к значениям произвольного *(^Y*)? Типы.
- F405 — PLDDICT или PLDOPTREF (*s – D*), предварительно загружает словарь *D* из *Slice s*. Приблизительно эквивалент LDDICT; DROP.
- F406 — LDDICTQ (*s – D s'* −1 или *s* 0), тихая версия LDDICT.
- F407 — PLDDICTQ (*s — D* −1 или 0), тихая версия PLDDICT.

A.10.3. Получение словарных операций.

- F40A — DICTGET (*k D n – x* −1 или 0), ищет ключ *k* (представлен *Slice,* первые $0 \le n \le 1023$ бита данных, из которых используются в качестве ключа) в словаре *D* типа *HashmapE(n,X) c n*-битными ключами. При успешном выполнении возвращает значение, найденное как *Фрагмент x.*
- F40B — DICTGETREF (*k D n — c* −1 или 0), аналогичный DICTGET, но с LDREF; ENDS применяется к *x* при успехе. Эта операция полезна для словарей типа *HashmapE(n,^Y*).
- F40C — DICTIGET (*i D n — x* −1 или 0), похожий на DICTGET, но со

знаковым (big-endian) *n-битным целым числом i в* качестве ключа. Если *i* не помещается в *n* бит, возвращает 0. Если *i* является NaN, создает целочисленное исключение переполнения.

- F40D — DICTIGETREF (*i D n — c* −1 или 0), объединяет DICTIGET с DICTGETREF: он использует знаковое *n*-битное целое число *i* в качестве ключа и возвращает ячейку вместо *Slice* при успешном выполнении.
- F40E — DICTUGET (*i D n — x* −1 или 0), похожий на DICTIGET, но с *беззнаковым* (big-endian) *n*-битным целым числом *i* , используемым в качестве ключа.
- F40F — DICTUGETREF (*i D n — c* −1 или 0), похожий на DICTIGETREF, но с беззнаковым *n*-битным ключом *Integer i.*

A.10.4. Установка/замена/добавление словарных операций. Мнемоника следующих примитивов словаря строится систематическим образом в соответствии с регулярным выражением `DICT[,I,U](SET,REPLACE,ADD)[GET][REF]` в зависимости от типа используемого ключа (*Slice* или знаковое или беззнаковое *Integer*), выполняемая операция словаря, а также способ принятия и возврата значений (в виде *Cells* или *Slices*). Поэтому мы приводим подробное описание только для некоторых примитивов, предполагая, что этой информации достаточно для того, чтобы читатель понял точное действие оставшихся примитивов.

- F412 — DICTSET (*x k D n – D'*), задает значение, связанное с *n-битным* ключом *k* (представленным *Slice*, как в DICTGET) в словаре *D* (также представленном *Slice*), значение *x* (снова *Slice*), и возвращает результирующий словарь как *D'.*
- F413 — DICTSETREF (*c k D n — D'*), аналогичный DICTSET, но со значением, установленным на ссылку на *Cell c.*
- F414 — DICTISET (*x i D n — D'*), похожий на DICTSET, но с ключом, представленным (*big-endian*) с *n*-битным *Integer i*. Если *i* не помещается в *n* битов, создается исключение проверки диапазона.
- F415 — DICTISETREF (*c i D n — D'*), похожий на DICTSETREF, но с ключом знаковое *n*-битное целое число, как в DICTISET.

- F416 — DICTUSET (*x i D n — D'*), похожий на DICTISET, но с *i беззнаковым n*-битным целым числом.
- F417 — DICTUSETREF (*c i D n — D'*), похожий на DICTISETREF, но с *i* без знака.
- F41A — DICTSETGET (*x k D n — D' y* −1 или *D'* 0), объединяет DICTSET с DICTGET: он устанавливает значение, соответствующее ключу *k,* в *x,* но также возвращает старое значение *y,* связанное с рассматриваемым ключом, если оно есть.
- F41B — DICTSETGETREF (*c k D n — D' c'* −1 или *D'* 0), комбайны DICTSETREF с DICTGETREF аналогично DICTSETGET.
- F41C — DICTISETGET (*x i D n — D' y* −1 или *D'* 0), похожий на DICTSETGET, но с ключом, представленным большим порядком байтов с n-битным целым *числом i.*
- F41D DICTISETGETREF (*c i D n – D' c'* −1 или *D'* 0), версия DICTSETGETREF со знаком *Integer i в* качестве ключа.
- F41E — DICTUSETGET (*x i D n — D' y* −1 или *D'* 0), аналогично DICTISETGET, но с *i* беззнаковым n-битным целым числом.
- F41F — DICTUSETGETREF (*c i D n — D' c'* −1 или *D'* 0).
- F422 — DICTREPLACE (*x k D n — D'* −1 или *D* 0), операция Replace, аналогичная DICTSET, но устанавливает значение ключа *k* в словаре *D* в *x* только в том случае, если ключ *k* уже присутствовал в *D.*
- F423 — DICTREPLACEREF (*c k D n — D'* −1 или *D* 0), заменяющий аналог DICTSETREF.
- F424 — DICTIREPLACE (*x i D n — D'* −1 или *D* 0), версия DICTREPLACE со знаковым *n-битным целым числом i* , используемым в качестве ключа.
- F425 — DICTIREPLACEREF (*c i D n — D'* −1 или *D* 0).
- F426 — DICTUREPLACE (*x i D n — D'* −1 или *D* 0).
- F427 — DICTUREPLACEREF (*c i D n – D'* −1 или *D* 0).
- F42A — DICTREPLACEGET (*x k D n – D' y* −1 или *D* 0), заменяющий аналог DICTSETGET: on success, также возвращает старое значение,

связанное с рассматриваемым ключом.

- F42B — DICTREPLACEGETREF (*c k D n – D′ c′* −1 или *D* 0).
- F42C — DICTIREPLACEGET (*x i D n — D′ y* −1 или *D* 0).
- F42D — DICTIREPLACEGETREF (*c i D n – D′ c′* −1 или *D* 0).
- F42E — DICTUREPLACEGET (*x i D n – D′ y* −1 или *D* 0).
- F42F — DICTUREPLACEGETREF (*c i D n – D′ c′* −1 или *D* 0).
- F432 — DICTADD (*x k D n — D′* −1 или *D* 0), аналог Add DICTSET: устанавливает значение, связанное с ключом *k* в словаре *D,* в *x,* но только если оно еще не присутствует в *D.*
- F433 — DICTADDREF (*c k D n — D′* −1 или *D* 0).
- F434 — DICTIADD (*x i D n – D′* −1 или *D* 0).
- F435 — DICTIADDREF (*c i D n — D′* −1 или *D* 0).
- F436 — DICTUADD (*x i D n — D′* −1 или *D* 0).
- F437 — DICTUADDREF (*c i D n — D′* −1 или *D* 0).
- F43A — DICTADDGET (*x k D n — D′* −1 или *D y* 0), аналог Add DICTSETGET: задает значение, связанное с ключом *k* в словаре *D* , в *x,* но только если ключ *k* еще не присутствует в *D.* В противном случае просто возвращает старое значение *y* без изменения словаря.
- F43B — DICTADDGETREF (*c k D n — D′* −1 или *D c′* 0), дополнительный аналог DICTSETGETREF.
- F43C — DICTIADDGET (*x i D n — D′* −1 или *D y* 0).
- F43D — DICTIADDGETREF (*c i D n — D′* −1 или *D c′* 0).
- F43E — DICTUADDGET (*x i D n — D′* −1 или *D y* 0).
- F43F — DICTUADDGETREF (*c i D n — D′* −1 или *D c′* 0).

A.10.5. Builder-принимающие варианты словарных операций Set. Следующие примитивы принимают новое значение как *Builder* b вместо *Slice* x, что часто более удобно, если значение необходимо сериализовать из нескольких компонентов, вычисляемых в стеке. (Это отражается добавлением `B` к мнемонике соответствующих примитивов

Set, работающих с *Slices.)* Чистый эффект примерно эквивалентен преобразованию *b* в *Slice* с помощью `ENDC`; `CTOS` и выполнение соответствующего примитива, перечисленного в A.10.4.

- F441 — DICTSETB (*b k D n – D'*).
- F442 — DICTISETB (*b i D n — D'*).
- F443 — DICTUSETB (*b i D n — D'*).
- F445 — DICTSETGETB (*b k D n – D' y* −1 или *D'* 0).
- F446 — DICTISETGETB (*b i D n — D' y* −1 или *D'* 0).
- F447 — DICTUSETGETB (*b i D n — D' y* −1 или *D'* 0).
- F449 — DICTREPLACEB (*b k D n – D'* −1 или *D* 0).
- F44A — DICTIREPLACEB (*b i D n — D'* −1 или *D* 0).
- F44B — DICTUREPLACEB (*b i D n — D'* −1 или *D* 0).
- F44D — DICTREPLACEGETB (*b k D n — D' y* −1 или *D* 0).
- F44E — DICTIREPLACEGETB (*b i D n – D' y* −1 или *D* 0).
- F44F — DICTUREPLACEGETB (*b i D n – D' y* −1 или *D* 0).
- F451 — DICTADDB (*b k D n — D'* −1 или *D* 0).
- F452 — DICTIADDB (*b i D n — D'* −1 или *D* 0).
- F453 — DICTUADDB (*b i D n — D'* −1 или *D* 0).
- F455 — DICTADDGETB (*b k D n — D'* −1 или *D y* 0).
- F456 — DICTIADDGETB (*b i D n — D'* −1 или *D y* 0).
- F457 — DICTUADDGETB (*b i D n — D'* −1 или *D y* 0).

A.10.6. Удаление словарных операций.

- F459 — DICTDEL (*k D n – D'* −1 или *D* 0), удаляет *n*-битный ключ, представленный *Slice k,* из словаря *D.* Если ключ присутствует, возвращает измененный словарь *D'* и флаг успеха −1. В противном случае возвращает исходный словарь *D* и 0.

- F45A — DICTIDEL (*i D n — D'* ?), версия DICTDEL с ключом, представленным знаковым n-битным целым *числом i.* Если *i* не помещается в *n* битов, просто возвращает *D* 0 («ключ не найден, словарь не изменен»).
- F45B — DICTUDEL (*i D n — D'* ?), похожий на DICTIDEL, но с *i* беззнаковым *n*-битным целым числом.
- F462 — DICTDELGET (*k D n – D' x* −1 или *D* 0), удаляет *n*-битный ключ, представленный *Slice k*, из словаря *D*. Если ключ присутствует, возвращает измененный словарь *D',* исходное значение *x,* связанное с ключом *k* (представленным *Slice*), и флаг успеха −1. В противном случае возвращает исходный словарь *D* и 0.
- F463 DICTDELGETREF (*k D n – D' c* −1 или *D* 0), аналогично DICTDELGET, но с LDREF; ENDS применяется к *x* при успешном завершении, так что возвращаемое значение *c* является ячейкой.
- F464 — DICTIDELGET (*i D n — D' x* −1 или *D* 0), вариант примитива DICTDELGET со знаковым *n*-битным целым числом *i* в качестве ключа.
- F465 — DICTIDELGETREF (*i D n — D' c* −1 или *D* 0), вариант примитива DICTIDELGET, возвращающего ячейку вместо *Slice.*
- F466 — DICTUDELGET (*i D n — D' x* −1 или *D* 0), вариант примитива DICTDELGET с беззнаковым *n*-битным целым числом *i* в качестве ключа.
- F467 — DICTUDELGETREF (*i D n — D' c* −1 или *D* 0), вариант примитива DICTUDELGET, возвращающего ячейку вместо *Slice.*

A.10.7. Словарные операции «Может быть, ссылка». Следующие операции предполагают, что словарь используется для хранения значений $c^?$ типа Ячейка$^?$ («*Maybe Cell*»), который может быть использован, в частности, для хранения словарей в качестве значений в других словарях. Представление выглядит следующим образом: если $c^?$ является *ячейкой*, она хранится в виде значения без битов данных и ровно одной ссылки на эту ячейку. Если $c^?$ равно *Null*, то соответствующий ключ должен вообще отсутствовать в словаре.

- F469 — DICTGETOPTREF (*k D n – c*~?), вариант DICTGETREF, который возвращает *Null* вместо значения $c^?$ если клавиша *k* отсутствует в

словаре *D.*

- F46A — DICTIGETOPTREF (*i D n — c*[?]), похожий на DICTGETOPTREF, но с ключом, заданным знаковым *n*-битным *Integer i.* Если ключ *i* находится вне диапазона, также возвращает значение *Null.*
- F46B — DICTUGETOPTREF (*i D n — c*[?]), похожий на DICTGETOPTREF, но с ключом, заданным беззнаковым *n-битным Integer i.*
- F46D — DICTSETGETOPTREF (*c*[?] *k D n – D' c̃*?), вариант как DICTGETOPTREF, так и DICTSETGETREF, который устанавливает значение, соответствующее ключу *k* в словаре *D* в *c*[?] (если *c*[?] имеет значение *Null,* то вместо этого ключ удаляется) и возвращает старое значение *c̃*? (если ключ *k* отсутствовал ранее, вместо *значение Null).*
- F46E ДИКТИСЕТГЕТОПТРЕФ (c? *i D n – D0 c̃?), похожий на примитив DICTSETGETOPTREF, но использующий в качестве ключа знаковое n-битное целое число i*. Если *i* не помещается в *n* битов, создает исключение проверки диапазона.
- F46F — DICTUSETGETOPTREF (*c*[?] *i D n – D' c̃*[?]), похожий на примитивный DICTSETGETOPTREF, но использующий без знака *n-битное Integer i* в качестве ключа.

A.10.8. Операции со словарем префиксных кодов. Это некоторые основные операции для построения словарей префиксных кодов (см. 3.4.2). Основным применением словарей префиксных кодов является десериализация сериализованных структур данных TL-B или, в более общем плане, синтаксический анализ префиксных кодов. Поэтому большинство словарей префиксного кода будут постоянными и создаваться во время компиляции, а не следующими примитивами.

Некоторые операции Get для словарей префиксных кодов можно найти в A.10.11. Другие операции словаря префиксных кодов включают в себя:

- F470 — PFXDICTSET (*x k D n — D'* −1 или *D* 0).
- F471 — PFXDICTREPLACE (*x k D n — D'* −1 или *D* 0).
- F472 — PFXDICTADD (*x k D n — D'* −1 или *D* 0).
- F473 — PFXDICTDEL (*k D n — D'* −1 или *D* 0).

Эти примитивы полностью похожи на их непрефиксные аналоги кода `DICTSET` и т. Д. (ср. A.10.4), с очевидным отличием, что даже Set может выйти из строя в словаре кодов префиксов, поэтому флаг успеха также должен быть возвращен `PFXDICTSET` .

A.10.9. Варианты операций GetNext и GetPrev.

- F474 — DICTGETNEXT (*k D n — x' k'* −1 или 0), вычисляет минимальный ключ *k'* в словаре *D*, который лексикографически больше *k, и возвращает k'* (представленный *Slice*) вместе со связанным значением *x'* (также представленным *Slice*).
- F475 — DICTGETNEXTEQ (*k D n — x' k'* −1 или 0), аналогично DICTGETNEXT, но вычисляет минимальный ключ *k'* , который лексикографически больше или равен *k.*
- F476 — DICTGETPREV (*k D n — x' k'* −1 или 0), аналогично DICTGETNEXT, но вычисляет максимальный ключ *k'* лексикографически меньше *k.*
- F477 — DICTGETPREVEQ (*k D n – x' k'* −1 или 0), аналогично DICTGETPREV, но вычисляет максимальный ключ *k'* лексикографически меньше или равен *k.*
- F478 — DICTIGETNEXT (*i D n — x' i'* −1 или 0), похожий на DICTGETNEXT, но интерпретирует все ключи в словаре *D* как n-битные целые числа с большим порядком байтов и вычисляет минимальный ключ *i',* который больше *Integer i* (что не обязательно укладывается в *n* битов).
- F479 — DICTIGETNEXTEQ (*i D n — x' i'* −1 или 0).
- F47A — DICTIGETPREV (*i D n — x' i'* −1 или 0).
- F47B — DICTIGETPREVEQ (*i D n — x' i'* −1 или 0).
- F47C — DICTUGETNEXT (*i D n — x' I'* −1 или 0), похожий на DICTGETNEXT, но интерпретирует все ключи в словаре *D* как n-битные целые числа с большим порядком байтов и вычисляет минимальный ключ *i',* который больше *Integer i* (который не обязательно укладывается в *n* битов и не обязательно является неотрицательным).

- F47D — DICTUGETNEXTEQ (*i D n — x′ i′* −1 или 0).
- F47E — DICTUGETPREV (*i D n — x′ i′* −1 или 0).
- F47F — DICTUGETPREVEQ (*i D n — x′ i′* −1 или 0).

A.10.10. Операции GetMin, GetMax, RemoveMin, RemoveMax.

- F482 — DICTMIN (*D n – x k* −1 или 0), вычисляет минимальную клавишу *k* (представленный *Slice* с *n* битами данных) в словаре *D* и возвращает *k* вместе со связанным значением *x.*
- F483 — DICTMINREF (*D n — c k* −1 или 0), похожий на DICTMIN, но возвращает единственную ссылку в значении как *Cell c.*
- F484 — DICTIMIN (*D n — x i* −1 или 0), чем-то похожий на DICTMIN, но вычисляет минимальный ключ *i* при предположении, что все ключи являются *n*-битными целыми числами с биг-эндианом. Обратите внимание, что возвращаемые ключ и значение могут отличаться от вычисленных DICTMIN и DICTUMIN.
- F485DICTIMINREF (*D n – c i* −1 или 0).
- F486 — DICTUMIN (*D n – x i* −1 или 0), аналогично DICTMIN, но возвращает ключ в виде n-битного *Integer i* без знака.
- F487 — DICTUMINREF (*D n – c i* −1 или 0).
- F48A — DICTMAX (*D n – x k* −1 или 0), вычисляет максимальную клавишу *k* (представленный *Slice* с *n* битами данных) в словаре *D* и возвращает *k* вместе со связанным значением *x.*
- F48B — DICTMAXREF (*D n — c k* −1 или 0).
- F48C — DICTIMAX (*D n – x i* −1 или 0).
- F48D — DICTIMAXREF (*D n – c i* −1 или 0).
- F48E — DICTUMAX (*D n – x i* −1 или 0).
- F48F — DICTUMAXREF (*D n – c i* −1 или 0).
- F492 — DICTREMMIN (*D n – D′ x k* −1 или *D* 0), вычисляет минимальную клавишу *k* (представленную *срезом* с *n* битами данных) в словаре *D,* удаляет *k* из словаря и возвращает *k* вместе со связанным значением *x* и модифицированным словарем *D′.*

- F493 — DICTREMMINREF (*D n — D' c k* −1 или *D* 0), аналогично DICTREMMIN, но возвращает единственную ссылку в значении в виде *ячейки c.*
- F494 — DICTIREMMIN (*D n — D' x i* −1 или *D* 0), чем-то похожий на DICTREMMIN, но вычисляет минимальный ключ *i* при предположении, что все ключи являются *n*-битными целыми числами с биг-эндианом. Обратите внимание, что возвращаемые ключ и значение могут отличаться от вычисленных DICTREMMIN и DICTUREMMIN.
- F495 — DICTIREMMINREF (*D n – D' c i* −1 или *D* 0).
- F496 — DICTUREMMIN (*D n — D' x i* −1 или *D* 0), аналогично DICTREMMIN, но возвращает ключ в виде n-битного *Integer i* без знака.
- F497 — DICTUREMMINREF (*D n — D' c i* −1 или *D* 0).
- F49A DICTREMMAX (*D n – D' x k* −1 или *D* 0), вычисляет максимальный ключ *k* (представленный *срезом* с *n* битами данных) в словаре *D,* удаляет *k* из словаря и возвращает *k* вместе со связанным значением *x* и модифицированным словарем *D'.*
- F49B — DICTREMMAXREF (*D n – D' c k* −1 или *D* 0).
- F49C — DICTIREMMAX (*D n — D' x i* −1 или *D* 0).
- F49D — DICTIREMMAXREF (*D n – D' c i* −1 или *D* 0).
- F49E — DICTUREMMAX (*D n – D' x i* −1 или *D* 0).
- F49F — DICTUREMMAXREF (*D n – D' c i* −1 или *D* 0).

A.10.11. Специальные словарные операции Get dictionary и префиксного кода, а также константные словари.

- F4A0 — DICTIGETJMP (*i D n* —), похожий на DICTIGET (ср. A.10.12), но с *x* BLESSed в продолжение с последующим JMPX к нему при успехе. При неудаче ничего не делает. Это полезно для реализации конструкций переключателей/корпусов.
- F4A1 — DICTUGETJMP (*i D n* –), аналогично DICTIGETJMP, но выполняет DICTUGET вместо DICTIGET.

- F4A2 — DICTIGETEXEC (*i D n* –), аналогично DICTIGETJMP, но с EXECUTE вместо JMPX.
- F4A3 — DICTUGETEXEC (*i D n* –), аналогично DICTUGETJMP, но с EXECUTE вместо JMPX.
- F4A6_*n* — DICTPUSHCONST *n* (– *D n*), проталкивает непустой константный словарь *D* (как *Cell*?) вместе с его ключом длиной 0 ≤ *n* ≤ 1023, хранящимся как часть инструкции. Сам словарь создается из первого из оставшихся ссылок текущего продолжения. Таким образом, полную инструкцию DICTPUSHCONST можно получить, сначала сериализовав xF4A8_, затем сам непустой словарь (один 1 бит и ссылка на ячейку), а затем беззнаковое 10-битное целое число *n* (как бы инструкцией STU 10). Пустой словарь может быть отправлен примитивом NEWDICT (см. A.10.1).
- F4A8 PFXDICTGETQ (*s D n – s' x s''* −1 или *s* 0), ищет уникальный префикс *Slice s*, присутствующий в словаре кодов префиксов (ср. 3.4.2), представленном *Cell*? *D* и 0 ≤ *n* ≤ 1023. Если он найден, префикс *s* возвращается как *s'*, а соответствующее значение (также *Slice*) как *x.* Оставшаяся часть *s* возвращается в виде *Slice s''*. Если префикс *s* не является ключом в словаре кодов префиксов *D,* возвращает неизменный *s* и нулевой флаг, указывающий на сбой.
- F4A9 — PFXDICTGET (*s D n — s' x s''*), аналогично PFXDICTGET, но при сбое создает исключение сбоя десериализации ячейки.
- F4AA — PFXDICTGETJMP (*s D n — s' s''* или *s*), похожий на PFXDICTGETQ, но при успешном успехе BLESSes значение *x* переходит в *Продолжение* и передает ему управление как бы JMPX. При сбое возвращает *s* без изменений и продолжает выполнение.
- F4AB — PFXDICTGETEXEC (*s D n — s' s''*), похожий на PFXDICTGETJMP, но EXECutes найденное продолжение, а не переход к нему. При сбое создается исключение десериализации ячейки.
- F4AE_*n* — PFXDICTCONSTGETJMP *n* или PFXDICTSWITCH *n* (*s — s' s''* или *s*), объединяет DICTPUSHCONST *n* для 0 ≤ *n* ≤ 1023 с PFXDICTGETJMP.
- F4BC — DICTIGETJMPZ (*i D n — i* или ничего), вариант DICTIGETJMP, который возвращает индекс *i* при сбое.

- F4BD — DICTUGETJMPZ (*i D n – i* или ничего), вариант DICTUGETJMP, который возвращает индекс *i* при сбое.
- F4BE — DICTIGETEXECZ (*i D n – i* или ничего), вариант DICTIGETEXEC, который возвращает индекс *i* при сбое.
- F4BF — DICTUGETEXECZ (*i D n – i* или ничего), вариант DICTUGETEXEC, который возвращает индекс *i* при сбое.

A.10.12. Словарные операции SubDict.

- F4B1 — SUBDICTGET (*k l D n – D'*), конструирует поддиционарий, состоящий из всех ключей, начинающихся с префикса *k* (представленного *Slice*, первые $0 \le l \le n \le 1023$ бита данных, из которых используются в качестве ключа) длины *l* в словаре *D* типа *HashmapE(n,X)* с *n*-битными ключами. Об успехе, возвращает новый подраздел того же типа *HashmapE(n,X)* в виде *Slice D'.*
- F4B2 — SUBDICTIGET (*x l D n – D'*), вариант SUBDICTGET с префиксом, представленным знаковым большим порядком байта *l-bit Integer x,* где обязательно $l \le 257$.
- F4B3 — SUBDICTUGET (*x l D n – D'*), вариант SUBDICTGET с префиксом, представленным беззнаковым большим порядком байт *l-bit Integer x*, где обязательно $l \le 256$.
- F4B5 — SUBDICTRPGET (*k l D n — D'*), похожий на SUBDICTGET, но удаляющий общий префикс *k* из всех ключей нового словаря *D'*, который становится типа *HashmapE(n − l,X).*
- F4B6 — SUBDICTIRPGET (*x l D n – D'*), вариант SUBDICTRPGET с префиксом, представленным знаковым большим порядком байта *l-bit Integer x*, где обязательно $l \le 257$.
- F4B7 — SUBDICTURPGET (*x l D n – D'*), вариант SUBDICTRPGET с префиксом, представленным беззнаковым большим порядком байт *l-bit Integer x*, где обязательно $l \le 256$.
- F4BC–F4BF — используется DICT... Z-примитивы в A.10.11.

A.11 Специализированные примитивы

Диапазон опкодов `F8`... `FB` зарезервирован для *примитивов, специфичных для приложения*. Когда TVM используется для выполнения

смарт-контрактов TON Blockchain, эти специализированные примитивы приложений на самом деле специфичны для TON Blockchain.

A.11.1. Внешние действия и доступ к данным конфигурации блокчейна. Некоторые из примитивов, перечисленных ниже, притворяются, что производят некоторые внешне видимые действия, такие как отправка сообщения на другой смарт-контракт. Фактически, выполнение смарт-контракта в TVM никогда не имеет никакого эффекта, кроме модификации состояния TVM. Все внешние действия собираются в связанный список, хранящийся в специальном реестре `c5` («выходные действия»). Кроме того, некоторые примитивы используют данные, хранящиеся в первом компоненте *Tuple,* хранящемся в `c7` («корень временных данных», см. 1.3.2). Смарт-контракты могут свободно изменять любые другие данные, хранящиеся в ячейке `c7`, при условии, что первая ссылка остается нетронутой (в противном случае некоторые специфичные для приложения примитивы, скорее всего, будут создавать исключения при вызове).

Большинство примитивов, перечисленных ниже, используют 16-битные опкоды.

A.11.2. Газовые примитивы. Из следующих примитивов только первые два являются «чистыми» в том смысле, что они не используют `c5` или `c7`.

- F800 — ACCEPT, устанавливает текущий газовый предел g_l на его максимально допустимое значение g_m и сбрасывает газовый кредит g_c до нуля (ср. 1,4), уменьшая значение g_r на g_c в процессе. Другими словами, текущий смарт-контракт соглашается купить немного газа, чтобы завершить текущую сделку. Это действие требуется для обработки внешних сообщений, которые сами по себе не приносят никакой ценности (следовательно, никакого газа).
- F801 — SETGASLIMIT (g –), устанавливает текущий газовый предел g_l до минимума g и g_m, и сбрасывает газовый кредит g_c до нуля. Если газ, потребленный до настоящего времени (включая настоящую инструкцию), превышает результирующее значение g_l, перед установкой новых предельных значений газа создается (необработанное) исключение из газа. Обратите внимание, что SETGASLIMIT с аргументом $g \geq 263 - 1$ эквивалентен ACCEPT.

- F802 — BUYGAS (*x* –), вычисляет количество газа, которое можно купить за *x* нанограммов, и устанавливает g_l соответственно так же, как SETGASLIMIT.
- F804 — GRAMTOGAS (*x* — *g*), вычисляет количество газа, которое можно купить за *x* нанограммов. Если значение *x* отрицательно, возвращает значение 0. Если *g* превышает 2^{63}–1, оно заменяется этим значением.
- F805 — GASTOGRAM (*g* – *x*), вычисляет цену *g* газа в нанограммах.
- F806–F80E — Зарезервировано для примитивов, связанных с газом.
- F80F — COMMIT (–), фиксирует текущее состояние регистров c4 («постоянные данные») и c5 («действия»), так что текущее выполнение считается «успешным» с сохраненными значениями, даже если исключение будет выброшено позже.

A.11.3. Примитивы генератора псевдослучайных чисел. Генератор псевдослучайных чисел использует случайное начальное число (параметр #6, ср. A.11.4), 256-битное *целое число без* знака и (иногда) другие данные, хранящиеся в `c7`. Начальное значение случайного начального значения перед выполнением смарт-контракта в TON Blockchain представляет собой хэш адреса смарт-контракта и случайного семя глобального блока. Если внутри блока есть несколько запусков одного и того же смарт-контракта, то все эти запуски будут иметь одно и то же случайное семя. Это можно исправить, например, запустив `LTIME; ADDRAND` перед первым использованием генератора псевдослучайных чисел.

- F810 — RANDU256 (– *x*), генерирует новое псевдослучайное беззнаковое 256-битное *Integer x*. Алгоритм выглядит следующим образом: если *r* — старое значение случайного начального значения, рассматриваемого как 32-байтовый массив (путем построения биг-порядкового представления беззнакового 256-битного целого числа), то вычисляется его sha512(*r*); первые 32 байта этого хэша сохраняются как новое значение *r'* случайного семя, а остальные 32 байта возвращаются в качестве следующего случайного значения *x.*
- F811 — RAND (*y* — *z*), генерирует новое псевдослучайное целое число *z* в диапазоне 0...*y* − 1 (или *y* ... − 1, если *y* < 0). Точнее,

генерируется беззнаковое случайное значение *x*, как в RAND256U; затем вычисляется *z* := ⌊*xy*/2^{256}⌋. Эквивалент RANDU256; MULRSHIFT 256.

- F814 — SETRAND (*x* –), задает для случайного начального значения значение без знака 256 бит *Integer x.*
- F815 — ADDRAND (*x* –), смешивает беззнаковое 256-битное *Integer x* в случайное начальное число *r*, устанавливая случайное семяизложение sha256 сцепления двух 32-байтовых строк: первая с большим порядковым представлением старого семя *r,* а вторая с большим порядковым представлением *x.*
- F810–F81F — Зарезервировано для примитивов генератора псевдослучайных чисел.

A.11.4. Конфигурационные примитивы. Следующие примитивы считывают данные конфигурации, предоставленные в *Tuple,* хранящемся в первом компоненте *Tuple* в `c7`. Всякий раз, когда TVM вызывается для выполнения смарт-контрактов TON Blockchain, этот *Tuple* инициализируется структурой *SmartContractInfo* ; примитивы конфигурации предполагают, что он остался нетронутым.

- F82*i* — GETPARAM *i* (– *x*), возвращает *i-й* параметр из *Tuple,* предоставленного в c7 для 0 ≤ *i* < 16. Эквивалент PUSH c7; FIRST; INDEX *i.* Если одна из этих внутренних операций завершается ошибкой, создается соответствующее исключение проверки типов или проверки диапазона.
- F823 — NOW (– *x*), возвращает текущее время Unix в *Integer*. Если невозможно восстановить запрошенное значение, начиная с c7, создает исключение проверки типа или проверки диапазона в зависимости от обстоятельств. Эквивалент GETPARAM 3.
- F824 — BLOCKLT (– *x*), возвращает начальное логическое время текущего блока. Эквивалент GETPARAM 4.
- F825 — LTIME (– *x*), возвращает логическое время текущей транзакции. Эквивалент GETPARAM 5.
- F826 — RANDSEED (– *x*), возвращает текущее случайное начальное значение в виде 256-битного *Integer* без знака. Эквивалент

GETPARAM 6.

- F827 — BALANCE (– *t*), возвращает оставшийся баланс смарт-контракта в виде *Tuple,* состоящего из целого *числа* (остаток граммового баланса в нанограммах) и *Maybe Cell* (словарь с 32-битными ключами, представляющими баланс «лишних валют»). Эквивалент GETPARAM 7. Обратите внимание, что примитивы RAW, такие как SENDRAWMSG, не обновляют это поле.
- F828 — MYADDR (– *s*), возвращает внутренний адрес текущего смарт-контракта в виде *фрагмента* с MsgAddressInt. При необходимости его можно дополнительно проанализировать с помощью примитивов, таких как PARSESTDADDR или REWRITESTDADDR. Эквивалент GETPARAM 8.
- F829 — CONFIGROOT (– *D*), возвращает ячейку *Maybe Cell D* с текущим словарем глобальной конфигурации. Эквивалент GETPARAM 9.
- F830 — CONFIGDICT (— *D* 32), возвращает словарь глобальной конфигурации вместе с длиной ключа (32). Эквивалент CONFIGROOT; PUSHINT 32.
- F832 — CONFIGPARAM (*i* – *c* −1 или 0), возвращает значение параметра глобальной конфигурации с целочисленным индексом *i* в качестве *ячейки c* и флагом, указывающим на успех. Эквивалент CONFIGDICT; DICTIGETREF.
- F833 — CONFIGOPTPARAM (*i* – $c^?$), возвращает значение параметра глобальной конфигурации с целочисленным индексом *i* как *Maybe Cell* $c^?$. Эквивалент CONFIGDICT; DICTIGETOPTREF.
- F820—F83F — Зарезервировано для примитивов конфигурации.

A.11.5. Глобальные переменные примитивы. «Глобальные переменные» могут быть полезны при реализации некоторых высокоуровневых языков смарт-контрактов. На самом деле они хранятся как компоненты *кортежа* при $\circ 7$: *k*-я глобальная переменная просто является для *k*-й составляющей этого *Tuple* $1 \le k \le 254$. По соглашению, A.11.4, поэтому он не является 0-м компонентом, используется для «параметров конфигурации» доступных в качестве глобальной переменной.

- F840 — GETGLOBVAR (*k* – *x*), возвращает *k-ю* глобальную переменную для 0 ≤ *k* < 255. Эквивалент PUSH c7; SWAP; INDEXVARQ (см. A.3.2).
- F85_*k* — GETGLOB *k* (– *x*), возвращает *k-ю* глобальную переменную для 1 ≤ *k* ≤ 31. Эквивалент PUSH c7; INDEX *k.*
- F860 — SETGLOBVAR (*x k* –), присваивает *x k-й* глобальной переменной для 0 ≤ *k* < 255. Эквивалент PUSH c7; ROTREV; SETINDEXVARQ; POP c7.
- F87_*k* — SETGLOB *k* (*x* –), присваивает *x k-й* глобальной переменной для 1 ≤ *k* ≤ 31. Эквивалент PUSH c7; SWAP; SETINDEXQ *k;* POP c7.

A.11.6. Примитивы хеширования и криптографии.

- F900 — HASHCU (*c* – *x*), вычисляет хэш представления (ср. 3.1.5) *Cell c* и возвращает его в виде 256-битного целого числа без знака *x.* Полезно для подписи и проверки подписей произвольных сущностей, представленных деревом ячеек.
- F901 — HASHSU (*s* – *x*), вычисляет хэш *Slice s* и возвращает его в виде 256-битного целого числа без знака *x.* Результат такой же, как если бы была создана обычная ячейка, содержащая только данные и ссылки из *s* , и ее хэш вычисляется HASHCU.
- F902 — SHA256U (*s* – *x*), вычисляет sha256 битов данных *Slice s.* Если длина бита *s* не делится на восемь, возникает исключение переполнения ячейки. Хэш-значение возвращается в виде 256-разрядного целого числа x без знака.
- F910 — CHKSIGNU (*h s k* — ?), проверяет Ed25519-сигнатуру *s* хэш *h* (256-битное целое число без знака, обычно вычисляемое как хэш некоторых данных) с помощью открытого ключа *k* (также представленного 256-битным целым числом без знака). Сигнатура *s* должна быть *Slice,* содержащим не менее 512 бит данных; используются только первые 512 бит. Результат равен −1, если подпись действительна, 0 в противном случае. Обратите внимание, что CHKSIGNU эквивалентен ROT; NEWB; STU 256; ENDB; NEWC; ROTREV; CHKSIGNS, т.е. в CHKSIGNS с первым аргументом *d*, установленным в 256-битный *Slice,* содержащий *h.* Поэтому, если *h* вычисляется как хэш некоторых данных, эти данные хэшируются

дважды, причем второе хеширование происходит внутри CHKSIGNS.

- F911 — CHKSIGNS (*d s k* – ?), проверяет, является ли *s* допустимой Ed25519signature части данных *Slice d* с использованием открытого ключа *k,* аналогично CHKSIGNU. Если длина бита *Slice d* не делится на восемь, создается исключение недолива ячейки. Проверка подписей Ed25519 является стандартной, а sha256 используется для уменьшения *d* до 256-битного числа, которое фактически подписано.
- F912–F93F — Зарезервировано для хеширования и криптографических примитивов.

A.11.7. Прочие примитивы.

- F940 — CDATASIZEQ (*c n — x y z* −1 или 0), рекурсивно вычисляет количество отдельных ячеек *x,* битов данных *y* и ссылок на ячейки *z* в даге, укорененном в *Cell c,* эффективно возвращая общее хранилище, используемое этим dag, с учетом идентификации равных ячеек. Значения *x, y и z* вычисляются путем прохождения этого дага сначала по глубине, с хэш-таблицей посещенных хэшей клеток, используемых для предотвращения посещений уже посещенных клеток. Общее количество посещенных ячеек *x* не может превышать неотрицательное *целое число n;* в противном случае вычисление прерывается перед посещением *ячейки* (*n* + 1) и возвращается ноль, указывающий на сбой. Если *c* имеет значение *Null,* возвращает *x* = *y* = *z* = 0.
- F941 — CDATASIZE (*c n – x y z*), нетимая версия CDATASIZEQ, которая при сбое создает исключение переполнения ячейки (8).
- F942 — SDATASIZEQ (*s n – x y z* −1 или 0), похожий на CDATASIZEQ, но принимающий *Slice s* вместо *Cell.* Возвращаемое значение *x* не учитывает ячейку, содержащую сам фрагмент *s* ; однако биты данных и ссылки на ячейки *s* учитываются в *y* и *z.*
- F943 — SDATASIZE (*s n – x y z*), нетимная версия SDATASIZEQ, которая создает исключение переполнения ячейки (8) при сбое.
- F944–F97F — Зарезервировано для различных TON-специфических примитивов, которые не попадают в какую-либо другую конкретную категорию.

A.11.8. Примитивы валютных манипуляций.

- FA00 — LDGRAMS или LDVARUINT16 (*s – x s'*), загружает (десериализует) граммовую или VarUInteger 16 сумму из *CellSlice* s и возвращает сумму в виде *целого числа x* вместе с остатком *s'* из s . Ожидаемая сериализация *x* состоит из 4-битного целого числа больших порядковых чисел *l* без знака, за которым следует 8-битное беззнаковое представление большого порядкового числа *x.* Чистый эффект приблизительно эквивалентен LDU 4; SWAP; LSHIFT 3; LDUX.
- FA01 — LDVARINT16 (*s — x s'*), аналогично LDVARUINT16, но загружает знаковое целое число *x.* Приблизительно эквивалентно LDU 4; SWAP; LSHIFT 3; LDIX.
- FA02 — STGRAMS или STVARUINT16 (*b x – b'*), сохраняет (сериализует) *целое число x* в диапазоне $0...2^{120}–1$ в *Builder b* и возвращает полученный *Builder b'*. Сериализация *x* состоит из 4-битного целого числа *l* с большим порядком байтов , которое является наименьшим *целым числом* $l \geq 0$, таким образом, что $x < 2^{8l}$, за которым следует *8l*-битное беззнаковое представление большого порядкового числа *x.* Если *x* не принадлежит к поддерживаемому диапазону, создается исключение проверки диапазона.
- FA03 — STVARINT16 (*b x — b'*), аналогично STVARUINT16, но сериализует знаковое *целое число x* в диапазоне $-2^{119} ... 2^{119} - 1$.
- FA04 — LDVARUINT32 (*s – x s'*), загружает (десериализует) VarUInteger 32 из *CellSlice s* и возвращает десериализованное значение как *целое число* $0 \leq x < 2248$. Ожидаемая сериализация *x* состоит из 5-битного целого числа больших порядковых чисел с большим порядком байтов *l,* за которым следует 8-битное беззнаковое представление большого порядкового числа *x.* Чистый эффект приблизительно эквивалентен LDU 5; SWAP; SHIFT 3; LDUX.
- FA05 — LDVARINT32 (*s – x s'*), десериализует VarInteger 32 из *CellSlice s* и возвращает десериализованное значение в виде *целого числа* $-2^{247} \leq x < 2^{247}$.
- FA06 — STVARUINT32 (*b x – b'*), сериализует целое число $0 \leq x < 2^{248}$ как VarUInteger 32.

- FA07 STVARINT32 (*b x – b'*), сериализует *целое число* $-2^{247} \le x < 2^{247}$ как VarInteger 32.
- FA08–FA1F — Зарезервировано для примитивов валютных манипуляций.

A.11.9. Примитивы манипулирования сообщениями и адресами. Перечисленные ниже примитивы манипулирования сообщениями и адресами сериализуют и десериализуют значения в соответствии со следующей схемой TL-B (см. 3.3.4):

```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 8) external_address:(bits len)
          = MsgAddressExt;
anycast_info$_ depth:(#<= 30) { depth >= 1 }
     rewrite_pfx:(bits depth) = Anycast;
addr_std$10 anycast:(Maybe Anycast)
     workchain_id:int8 address:bits256 =
   MsgAddressInt;
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9)
   workchain_id:int32 address:(bits addr_len) =
   MsgAddressInt;
_ _:MsgAddressInt = MsgAddress;
_ _:MsgAddressExt = MsgAddress;

int_msg_info$0 ihr_disabled:Bool bounce:Bool
bounced:Bool
  src:MsgAddress dest:MsgAddressInt
  value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
  created_lt:uint64 created_at:uint32 =
  CommonMsgInfoRelaxed;
ext_out_msg_info$11 src:MsgAddress dest:MsgAddressExt
   created_lt:uint64 created_at:uint32 =
CommonMsgInfoRelaxed;
```

Десериализованный `MsgAddress` представлен *Tuple* *t* следующим образом:

- addr_none представлен *t* = (0), т.е. *Tuple*, содержащим ровно одно

Integer, равное нулю.

- addr_extern представлен *t* = (1,*s*), где *Slice s* содержит поле external_address. Другими словами, *t* — это пара (*Tuple*, состоящий из двух записей), содержащий *целое число*, равное единице, и *Slice s.*
- addr_std представлен *t* = (2,*u,x,s*), где *u* — это либо *Null* (если отсутствует anycast), либо *Slice s'* , содержащий rewrite_pfx (если присутствует anycast). Далее, *целое число x* — это workchain_id, а *Slice s* содержит адрес.
- addr_var представлен t = (3,*u,x,s*), где *u, x и s* имеют то же значение, что и для addr_std.

Определены следующие примитивы, которые используют вышеуказанные соглашения:

- FA40 — LDMSGADDR (*s – s' s''*), загружает из *CellSlice s* единственный префикс, который является допустимым MsgAddress, и возвращает как этот префикс *s'*, так и остаток *s''* *s* как *CellSlices.*
- FA41 — LDMSGADDRQ (*s — s' s''* −1 или *s* 0), тихая версия LDMSGADDR: на успех толкает лишний −1; при отказе толкает оригинал *s* и ноль.
- FA42 — PARSEMSGADDR (*s – t*), разлагает *CellSlice s* , содержащий допустимый MsgAddress, на *Tuple t* с отдельными полями этого MsgAddress. Если *s* не является допустимым MsgAddress, создается исключение десериализации ячейки.
- FA43 — PARSEMSGADDRQ (*s – t* −1 или 0), тихая версия PARSEMSGADDR: возвращает ноль по ошибке вместо того, чтобы выдавать исключение.
- FA44 — REWRITESTDADDR (*s – x y*), анализирует *CellSlice s*, содержащий допустимый MsgAddressInt (обычно msg_addr_std), применяет перезапись с anycast (если присутствует) к префиксу адреса одинаковой длины и возвращает как рабочую цепочку x, так и 256-битный адрес *y* в виде *целых чисел.* Если адрес не является 256-битным или если *s* не является допустимой сериализацией MsgAddressInt, создается исключение десериализации ячейки.

- FA45 — REWRITESTDADDRQ (*s* — *x y* −1 или 0), тихая версия примитива REWRITESTDADDR.
- FA46 — REWRITEVARADDR (*s* — *x s'*), вариант REWRITESTDADDR, который возвращает (переписанный) адрес в виде *Slice* s, даже если он не имеет длины 256 бит (представлен msg_addr_var).
- FA47 — REWRITEVARADDRQ (*s* — *x s'* −1 или 0), тихая версия примитива REWRITEVARADDR.
- FA48–FA5F — Зарезервировано для примитивов манипулирования сообщениями и адресами.

A.11.10. Примитивы исходящих сообщений и выходных действий.

- FB00 SENDRAWMSG (*c x* –), отправляет необработанное сообщение, содержащееся в *ячейке c,* которое должно содержать правильно сериализованный объект Message X, за исключением того, что исходный адрес может иметь фиктивное значение addr_none (автоматически заменяется текущим адресом smartcontract), а ihr_fee, fwd_fee, created_lt и created_at поля могут иметь произвольные значения (перезаписываться с правильными значениями на этапе действия текущей транзакции). Целочисленный параметр *x* содержит флаги. В настоящее время *x* = 0 используется для обычных сообщений; *x* = 128 используется для сообщений, которые должны нести весь оставшийся баланс текущего смарт-контракта (вместо значения, первоначально указанного в сообщении); *x* = 64 используется для сообщений, которые несут все оставшееся значение входящего сообщения в дополнение к значению, первоначально указанному в новом сообщении (если бит 0 не установлен, плата за газ вычитается из этой суммы); *x'* = *x* + 1 означает, что отправитель хочет оплатить комиссию за перевод отдельно; *x'* = *x* + 2 означает, что любые ошибки, возникающие при обработке этого сообщения на этапе действия, должны быть проигнорированы. Наконец, *x'* = *x* + 32 означает, что текущий счет должен быть уничтожен, если его итоговый баланс равен нулю. Этот флаг обычно используется вместе с +128.
- FB02 — RAWRESERVE (*x y* –), создает выходное действие, которое

резервирует ровно *x* нанограммов (если *y* = 0), не более *x* нанограммов (если *y* = 2) или все, кроме *x* нанограммов (если *y* = 1 или *y* = 3), из оставшегося баланса счета. Это примерно эквивалентно созданию исходящего сообщения, несущего *себе x* нанограммов (или *b* − *x* нанограммов, где *b* — оставшийся баланс), так что последующие выходные действия не смогут потратить больше денег, чем остальные. Бит +2 в *y* означает, что внешнее действие не завершается сбоем, если указанная сумма не может быть зарезервирована; вместо этого резервируется весь оставшийся баланс. Бит +8 в *y* означает *x* ← −*x* перед выполнением каких-либо дальнейших действий. Бит +4 в *y* означает, что *x* увеличивается на исходный баланс текущего счета (до этапа вычисления), включая все дополнительные валюты, перед выполнением любых других проверок и действий. В настоящее время *x* должно быть неотрицательным целым числом, а *y* должно находиться в диапазоне 0...15.

- FB03 — RAWRESERVEX (*x D y* –), похожий на RAWRESERVE, но также принимает словарь *D* (представленный ячейкой или *null)* с дополнительными валютами. Таким образом, можно зарезервировать валюты, отличные от граммов.
- FB04 SETCODE (*c* –), создает выходное действие, которое изменит этот код смарт-контракта на код, заданный *Cell c.* Обратите внимание, что это изменение вступит в силу только после успешного завершения текущего запуска смарт-контракта.
- FB06 — SETLIBCODE (*c x* –), создает выходное действие, которое изменяет коллекцию этих библиотек смарт-контрактов путем добавления или удаления библиотеки с кодом, приведенным в *ячейке c.* Если *x* = 0, библиотека фактически удаляется, если она ранее присутствовала в коллекции (если нет, это действие ничего не делает). Если *x* = 1, библиотека добавляется как частная библиотека, а если *x* = 2, библиотека добавляется как публичная библиотека (и становится доступной для всех смарт-контрактов, если текущий смарт-контракт находится в мастерчейне); если библиотека присутствовала в коллекции ранее, ее публичный/частный статус изменяется в соответствии с *x. Значения x* , отличные от 0...2,

являются недопустимыми.

- FB07 — CHANGELIB (*h x* –), создает выходное действие аналогично SETLIBCODE, но вместо кода библиотеки принимает его хэш как беззнаковое 256-битное целое число *h.* Если *x* 6= 0 и библиотека с хэшем *h* отсутствует в коллекции библиотеки этого смарт-контракта, это выходное действие завершится ошибкой.
- FB08–FB3F — зарезервировано для примитивов выходных действий.

A.12 Отладка примитивов

Опкоды, начинающиеся с `FE`, зарезервированы для *примитивов отладки*. Эти примитивы имеют фиксированную длину операции и ведут себя как (многобайтовые) операции NOP. В частности, они никогда не изменяют содержимое стека и никогда не генерируют исключения, если только не хватает битов для полного декодирования опкода. Однако при вызове в экземпляре TVM с включенным режимом отладки эти примитивы могут выдавать определенные выходные данные в текстовый журнал отладки экземпляра TVM, никогда не влияя на состояние TVM (так что с точки зрения TVM поведение отладочных примитивов в режиме отладки точно такое же). Например, отладочный примитив может сбрасывать все или некоторые значения в верхней части стека, отображать текущее состояние TVM и так далее.

A.12.1. Отладка примитивов в виде многобайтовых NOP.

- FE*nn* — DEBUG *nn*, для $0 \le nn < 240$, является двухбайтовым NOP.
- FEF*nssss* — DEBUGSTR *ssss*, для $0 \le n < 16$, представляет собой $(n + 3)$-байт NOP, при этом $(n + 1)$-байт "строка содержимого" *ssss* также пропущена.

A.12.2. Отладка примитивов как операций без побочных эффектов. Далее мы опишем отладочные примитивы, которые могут (и фактически реализованы) в версии TVM. Обратите внимание, что другая реализация TVM может свободно использовать эти коды для других целей отладки или рассматривать их как многобайтовые NOP. Всякий раз, когда этим примитивам нужны некоторые аргументы из стека, они проверяют эти аргументы, но оставляют их нетронутыми в стеке. Если в стеке недостаточно значений или они имеют неправильные типы, отладочные

примитивы могут выводить сообщения об ошибках в журнал отладки или вести себя как NOP, но они не могут создавать исключения.

- FE00 — DUMPSTK, сбрасывает стек (максимум 255 верхних значений) и показывает общую глубину стека.
- FE0n — DUMPSTKTOP *n,* $1 \le n < 15$, сбрасывает верхние *n* значений из стека, начиная с самого глубокого из них. Если доступны *значения d* $< n$, сбрасывает только значения *d*.
- FE10 — HEXDUMP, сбрасывает s0 в шестнадцатеричной форме, будь то *Slice* или *Целое число.*
- FE11 — HEXPRINT, аналогично HEXDUMP, за исключением того, что шестнадцатеричное представление s0 не выводится сразу, а скорее объединяется с выходным текстовым буфером.
- FE12 — BINDUMP, сбрасывает s0 в двоичном виде, аналогично HEXDUMP.
- FE13 — BINPRINT, выводит двоичное представление s0 в текстовый буфер.
- FE14 — STRDUMP, сбрасывает *Slice* в s0 в виде строки UTF-8.
- FE15 — STRPRINT, похожий на STRDUMP, но выводящий строку в текстовый буфер (без возврата каретки).
- FE1E — DEBUGOFF, отключает все выходные данные отладки до тех пор, пока они не будут повторно включены DEBUGON. Точнее, этот примитив увеличивает внутренний счетчик, который отключает все операции отладки (кроме DEBUGOFF и DEBUGON) при строго положительном значении.
- FE1F — DEBUGON, включает отладочный вывод (в отладочной версии TVM).
- FE2*n* — DUMP s(*n*), $0 \le n < 15$, дампы s(*n*).
- FE3*n* — PRINT s(*n*), $0 \le n < 15$, объединяет текстовое представление s(*n*) (без каких-либо начальных или конечных пробелов или возвратов каретки) в текстовый буфер, который будет выведен перед выводом любой другой операции отладки.

- FEC0–FEEF — используйте эти операционные коды для пользовательских/экспериментальных операций отладки.
- FEF*nssss* — DUMPTOSFMT *ssss,* дампы s0, отформатированные в соответствии с (*n* + 1)-байтовой строкой *ssss.* Эта строка может содержать (префикс) имя типа TL-B, поддерживаемого отладчиком. Если строка начинается с нулевого байта, просто выводит ее (без первого байта) в журнал отладки. Если строка начинается с байта, равного единице, объединяет ее в буфер, который будет выводиться перед выводом любой другой операции отладки (фактически выводит строку без возврата каретки).
- FEF*n*00*ssss* — LOGSTR *ssss,* строка *ssss* имеет *n* байт длиной.
- FEF000 — LOGFLUSH, сбрасывает все ожидающие отладочные выходные данные из буфера в журнал отладки.
- FEF*n*01*ssss* — PRINTSTR *ssss,* строка *ssss* имеет *n* байт длиной.

A.13 Примитивы кодовой страницы

Следующие примитивы, которые начинаются с байтового `FF`, обычно используются в самом начале кода смарт-контракта или подпрограммы библиотеки для выбора другой кодовой страницы TVM. Обратите внимание, что мы ожидаем, что все кодовые страницы будут содержать эти примитивы с одинаковыми кодами, в противном случае переключение обратно на другую кодовую страницу может быть невозможным (см. 5.1.8).

- FF*nn* — SETCP *nn*, выбирает кодовую страницу TVM $0 \le nn < 240$. Если кодовая страница не поддерживается, создается недопустимое исключение опкода.
- FF00 — SETCP0, выбирает нулевую кодовую страницу TVM (тестовая), как описано в этом документе.
- FFF*z* — SETCP $z - 16$, выбирает кодовую страницу TVM $z - 16$ для $1 \le z \le 15$. Отрицательные кодовые страницы $-13 \ldots -1$ зарезервированы для ограниченных версий TVM, необходимых для проверки запусков TVM в других кодовых страницах, как описано в B.2.6. Отрицательная кодовая страница −14 зарезервирована для экспериментальных кодовых страниц, не обязательно совместима между различными

реализациями TVM, и должна быть отключена в производственных версиях TVM.

- FFF0 — SETCPX (*c* –), выбирает кодовую страницу *c* с $-2^{15} \le c < 2^{15}$, переданных в верхней части стека.

B Формальные свойства и спецификации TVM

В этом приложении обсуждаются некоторые формальные свойства TVM, которые необходимы для выполнения смарт-контрактов в блокчейне TON и последующей проверки таких исполнений.

B.1 Сериализация состояния TVM

Напомним, что виртуальная машина, используемая для выполнения смарт-контрактов в блокчейне, должна быть *детерминированной*, иначе проверка каждого выполнения потребует включения всех промежуточных этапов выполнения в блок или, по крайней мере, выбора, сделанного при выполнении индетерминированных операций.

Кроме того, *состояние* такой виртуальной машины должно быть (уникально) сериализуемым, так что даже если само состояние обычно не включается в блок, его *хэш* все равно четко определен и может быть включен в блок для целей проверки.

B.1.1. Значения стека TVM. Значения стека TVM могут быть сериализованы следующим образом:

```
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;
```

Из них `vm_stk_tinyint` никогда не используется TVM в нулевой кодовой странице; он используется только в ограниченных режимах.

B.1.2. Стек TVM. Стек TVM может быть сериализован следующим образом:

```
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. Управляющие регистры TVM. Управляющие регистры в TVM могут быть сериализованы следующим образом:

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

B.1.4. Ограничения по газу TVM. Ограничения по газу в TVM могут быть сериализованы следующим образом:

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

B.1.5. Библиотечная среда TVM. Библиотечная среда TVM может быть сериализована следующим образом:

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

B.1.6. TVM продолжения. Продолжения в TVM могут быть сериализованы следующим образом:

```
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. Состояние ТВМ. Общее состояние TVM может быть сериализовано следующим образом:

```
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;
```

При инициализации TVM его состояние описывается `vms_init`, обычно с `шагом`, установленным равным нулю. Ступенчатая функция TVM ничего не делает с vms_finished состоянием и преобразует все другие состояния в `vms_running`, `vms_exception` или `vms_finished`, с `шагом`, увеличенным на единицу.

B.2 Пошаговая функция TVM

Формальная спецификация TVM будет дополнена определением ступенчатой функции f: *VmState* → *VmState*. Эта функция детерминированно преобразует допустимое состояние виртуальной машины в допустимое последующее состояние виртуальной машины и может вызывать исключения или возвращать недопустимое последующее состояние, если исходное состояние было недопустимым.

B.2.1. Высокоуровневое определение ступенчатой функции. Мы могли бы представить очень длинное формальное определение ступенчатой функции TVM в высокоуровневом функциональном языке программирования. Такая спецификация, однако, была бы в основном полезна в качестве справочного материала для (человеческих)

разработчиков. Мы выбрали другой подход, лучше адаптированный к автоматизированной формальной верификации компьютерами.

B.2.2. Эксплуатационное определение ступенчатой функции. Обратите внимание, что пошаговая функция f является четко определенной вычислимой функцией из деревьев клеток в деревья ячеек. Таким образом, он может быть вычислен универсальной машиной Тьюринга. Тогда программа P , вычисляющая f на такой машине, предоставит машинопроверяемую спецификацию пошаговой функции f. Эта программа P фактически является *эмулятором* TVM на этой машине Тьюринга.

B.2.3. Эталонная реализация эмулятора TVM внутри TVM. Мы видим, что пошаговая функция TVM может быть определена эталонной реализацией эмулятора TVM на другой машине. Очевидной идеей является использование самого TBM, так как он хорошо приспособлен для работы с деревьями клеток. Однако эмулятор TVM внутри себя не очень полезен, если у нас есть сомнения по поводу конкретной реализации TVM и мы хотим ее проверить. Например, если бы такой эмулятор интерпретировал `инструкцию DICTISET`, просто вызвав саму эту инструкцию, то ошибка в базовой реализации TVM осталась бы незамеченной.

B.2.4. Эталонная реализация внутри минимальной версии TVM. Мы видим, что использование самого TVM в качестве хост-машины для эталонной реализации эмулятора TVM даст мало понимания. Лучше всего определить *урезанную версию TVM*, которая поддерживает только минимум примитивов и 64-разрядную целочисленную арифметику, и предоставить эталонную реализацию P пошаговой функции TVM f для этой урезанной версии TVM.

В этом случае необходимо тщательно реализовать и проверить только несколько примитивов, чтобы получить урезанную версию TVM, и сравнить эталонную реализацию P, работающую на этой урезанной версии, с полной пользовательской реализацией TVM, которая проверяется. В частности, если есть какие-либо сомнения относительно обоснованности конкретного запуска пользовательской реализации TVM, они теперь могут быть легко разрешены с помощью эталонной реализации.

B.2.5. Актуальность для блокчейна TON. Блокчейн TON использует этот подход для проверки запусков TVM (например, тех, которые используются для обработки входящих сообщений смарт-контрактами), когда результаты валидаторов не совпадают друг с другом. В этом случае для получения правильного результата используется эталонная реализация TVM, хранящаяся внутри мастерчейна в виде настраиваемого параметра (таким образом определяющего текущую ревизию TVM).

B.2.6. Кодовая страница -1. *Кодовая страница* -1 TVM зарезервирована для урезанной версии TVM. Его основной целью является выполнение эталонной реализации пошаговой функции полного TVM. Эта кодовая страница содержит только специальные версии арифметических примитивов, работающих с «крошечными целыми числами» (64-битными знаковыми целыми числами); поэтому 257-битная *целочисленная* арифметика TVM должна быть определена в терминах 64-битной арифметики. Примитивы криптографии эллиптических кривых также реализуются непосредственно в кодовой странице 1, без использования каких-либо сторонних библиотек. Наконец, эталонная реализация хэш-функции sha256 также представлена в кодовой странице -1.

B.2.7. Кодовая страница -2. Этот процесс начальной загрузки можно было бы итерировать еще дальше, предоставив эмулятор урезанной версии TVM, написанной для еще более простой версии TVM, которая поддерживает только логические значения (или целые числа 0 и 1) — «кодовую страницу -2». Вся 64-битная арифметика, используемая в кодовой странице -1 , должна быть определена с помощью логических операций, что обеспечивает эталонную реализацию для урезанной версии TVM, используемой в кодовой странице -1. Таким образом, если некоторые из валидаторов TON Blockchain не согласятся с результатами своей 64-битной арифметики, они могут регрессировать к этой эталонной реализации, чтобы найти правильный ответ.[30]

[30] Предварительная версия TVM не использует кодовую страницу -2 для этой цели. Это может измениться в будущем.

C Плотность кода стековых и регистровых машин

Это приложение расширяет общее рассмотрение примитивов манипулирования стеком, приведенное в 2.2, объясняя выбор таких примитивов для TVM, со сравнением стековых машин и регистровых машин с точки зрения количества используемых примитивов и плотности кода. Мы делаем это, сравнивая машинный код, который может быть сгенерирован оптимизирующим компилятором для одних и тех же исходных файлов, для разных (абстрактных) стеков и регистровых машин.

Оказывается, что стековые машины (по крайней мере, те, которые оснащены базовыми примитивами манипулирования стеком, описанными в 2.2.1) имеют гораздо более высокую плотность кода. Кроме того, стековые машины обладают отличной расширяемостью в отношении дополнительных арифметических и произвольных операций обработки данных, особенно если учесть машинный код, автоматически генерируемый оптимизацией компиляторов.

C.1 Функция листа образца

Начнем со сравнения машинного кода, сгенерированного (воображаемым) оптимизирующим компилятором для нескольких абстрактных регистровых и стековых машин, соответствующих одному и тому же высокоуровневому исходному коду языка, который содержит определение конечной функции (т.е. функции, которая не вызывает никаких других функций). Как для регистровых машин, так и для стековых машин мы соблюдаем обозначения и соглашения, представленные в 2.1.

C.1.1. Образец исходного файла для функции листа. Рассматриваемый нами исходный файл содержит одну *функцию* f, которая принимает шесть (целочисленных) аргументов, a, b, c, , d, e, f, и возвращает два (целочисленных) значения, x и y, которые являются решениями системы двух линейных уравнений

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

Исходный код функции на языке программирования, аналогичном C, может выглядеть следующим образом:

```
(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);
}
```

Мы предполагаем (ср. 2.1), что регистровые машины, которые мы рассматриваем, принимают шесть параметров… f в регистрах `r0`... `r5` и возвращает два значения x и y в `r0` и `r1`. Мы также предполагаем, что регистровые машины имеют 16 регистров и что машина стека может напрямую обращаться к `s0` к `s15` с помощью примитивов манипулирования стеком; машина стека будет принимать параметры от `s5` до `s0` и возвращать два значения в `s0` и `s1` , что-то похожее на регистр машину. Наконец, сначала мы предполагаем, что регистровой машине позволено уничтожать значения во всех регистрах (что немного несправедливо по отношению к стековой машине); это предположение будет пересмотрено позднее.

C.1.2. Трехадресная регистровая машина. Машинный код (или, скорее, соответствующий ассемблерный код) для машины с тремя адресами регистра (см. 2.1.7) может выглядеть следующим образом:

```
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
```

Мы использовали 12 операций и не менее 23 байт (каждая операция использует $3\times4 = 12$ бит для указания трех задействованных регистров и не менее 4 бит для обозначения выполненной операции; таким образом, нам нужно два или три байта для кодирования каждой операции). Более реалистичной оценкой было бы 34 (три байта для каждой арифметической операции) или 31 байт (два байта для сложения и вычитания, три байта для умножения и деления).

С.1.3. Двухадресная регистровая машина. Машинный код для регистрового компьютера с двумя адресами может выглядеть следующим образом:

```
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
```

Мы использовали 16 операций; оптимистично предполагая, что каждая из них (за исключением `RET`) может быть закодирована двумя байтами, этот код потребует 31 байт.[31]

[31] Интересно сравнить этот код с кодом, созданным путем оптимизации компиляторов C для архитектуры x86-64.

Прежде всего, операция целочисленного деления для x86-64 использует одноадресную форму, при этом дивиденд (двойной длины) подается в аккумуляторной паре `r2:r0`.

C.1.4. Одноадресная регистровая машина. Машинный код для машины регистрации одного адреса может выглядеть следующим образом:

```
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
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
```

Мы использовали 23 операции; если предположить однобайтовое кодирование для всех арифметических операций и `XCHG`, и двухбайтовое кодирование для `MOV`, то общий размер кода составит 29 байт. Обратите

Частное также возвращается в `r0`. Как следствие, необходимо добавить две операции расширения от одного до двух (`CDQ` или `CQO`) и по крайней мере одну операцию перемещения.

Во-вторых, кодировка, используемая для арифметических и движущихся операций, менее оптимистична, чем в нашем примере выше, требуя в среднем около трех байт на операцию. В результате мы получаем в общей сложности 43 байта для 32-битных целых чисел и 68 байт для 64-битных целых чисел.

внимание, однако, что для получения компактного кода, показанного выше, мы должны были выбрать определенный порядок вычислений и интенсивно использовать коммутативность умножения. (Например, мы вычисляем *bc* перед *af*, и *af bc* сразу после *af*.) Неясно, сможет ли компилятор сделать все такие оптимизации самостоятельно.

С.1.5. Стековая машина с базовыми примитивами стека. Машинный код стековой машины, оснащенной базовыми примитивами управления стеком, описанными в разделе 2.2.1, может выглядеть следующим образом:

```
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
```

Мы использовали 29 операций; предполагая однобайтовые кодировки для всех задействованных операций стека (включая `XCHG s1,s(`i`)`), мы также использовали 29 байт кода. Обратите внимание, что при однобайтовом кодировании «несистематическая» операция `ROT` (эквивалентная `XCHG s1; XCHG s2`) уменьшит количество операций и байтов до 28. Это показывает, что такие «несистематические» операции, заимствованные у Forth, действительно могут в некоторых случаях уменьшить размер кода.

Обратите внимание, что мы неявно использовали коммутативность умножения в этом коде, вычисляя $de - bf$ вместо $ed - bf$, как указано в исходном коде языка высокого уровня. Если бы нам не разрешили это сделать, перед третьим `XCHG s1` необходимо было бы вставить дополнительный `IMUL`, увеличив общий размер кода на одну операцию и один байт.

Код, представленный выше, мог быть создан довольно несложным компилятором, который просто вычислял все выражения и подвыражения в том порядке, в котором они появляются, а затем переставлял аргументы в верхней части стека перед каждой операцией, как описано в 2.2.2. Единственная «ручная» оптимизация, выполненная здесь, включает в себя вычисления ec перед af; можно проверить, что другой порядок приведет к немного более короткому коду из 28 операций и байтов (или 29, если нам не разрешено использовать коммутативность умножения), но оптимизация ROT не будет применима.

С.1.6. Стековая машина со составными примитивами стека. Стековая машина со составными примитивами стека (ср. 2.2.3) существенно не улучшит плотность кода, представленного выше, по крайней мере, с точки зрения используемых байтов. Единственное отличие состоит в том, что, если бы нам не разрешили использовать коммутативность умножения, дополнительный `XCHG s1`, вставленный перед третьим `IMUL`, можно было бы объединить с двумя предыдущими операциями `XCHG s3`, `PUSH s2` в одну составную операцию `PUXC s2,s3`; мы предоставляем

результирующий код ниже. Чтобы сделать это менее избыточным, мы покажем версию кода, которая вычисляет подвыражение af перед ec, как указано в исходном файле. Мы видим, что это заменяет шесть операций (начиная с строки 15) пятью другими операциями и отключает оптимизацию `ROT`:

```
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
```

Мы использовали в общей сложности 27 операций и 28 байт, как и в предыдущей версии (с оптимизацией `ROT`). Однако мы не использовали здесь коммутативность умножения, поэтому можем сказать, что

примитивы обработки составным стеком позволяют нам уменьшить размер кода с 29 до 28 байт.

Опять же, обратите внимание, что приведенный выше код мог быть сгенерирован несложным компилятором. Ручная оптимизация может привести к более компактному коду; например, мы могли бы использовать составные операции, такие как `XCHG3` , чтобы заранее подготовить не только правильные значения `s0` и `s1` для следующей арифметической операции, но и значение `s2` для арифметической операции после нее. В следующем разделе приведен пример такой оптимизации.

С.1.7. Стековая машина с примитивами составного стека и оптимизированным вручную кодом. Предыдущую версию кода для стековой машины со составными примитивами стека можно оптимизировать вручную следующим образом.

Чередуя операции `XCHG` с предыдущими операциями `XCHG`, `PUSH` и арифметическими операциями, когда это возможно, мы получаем фрагмент кода `XCHG s2,s6; XCHG s1,s0; XCHG s0,s5`, который затем может быть заменен составной операцией `XCHG3 s6,s0,s5`. Эта составная операция допускает двухбайтовое кодирование, что приводит к 27-байтовому коду, использующему только 21 операцию:

```
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
```

Интересно отметить, что эта версия машинного кода стека содержит только 9 примитивов манипулирования стеком для 11 арифметических операций. Однако неясно, сможет ли оптимизирующий компилятор реорганизовать код таким образом самостоятельно.

C.2 Сравнение машинного кода для примера функции листа

В таблице 1 обобщены свойства машинного кода, соответствующего одному и тому же исходному файлу, описанному в C.1.1, сгенерированному для гипотетической машины с тремя адресами регистра (см. C.1.2), с «оптимистическими» и «реалистичными» кодировками инструкций; двухадресная машина (см. C.1.3); одноадресная машина (см. C.1.4); и стековая машина, аналогичная TVM, использующая либо только базовые примитивы манипулирования стеком (см. C.1.5), либо как базовый, так и составной примитивы стека (см. C.1.7).

Значение столбцов в таблице 1 заключается в следующем:

- «Операции» — количество используемых инструкций, разделенных на «данные» (т.е. инструкции по перемещению регистров и обмену для регистровых машин и инструкции по манипулированию стеком для стековых машин) и «арифметические» (инструкции по сложению, вычитанию, умножению и делению целых чисел). «Итого» на единицу больше, чем сумма этих двух, потому что в конце машинного кода также есть однобайтовая инструкция RET.
- "Байты кода" — общее количество использованных байтов кода.

«Пространство опкода» — часть «пространства опкода» (т. е. возможного выбора для первого байта кодирования инструкции), используемая данными и арифметическими инструкциями в предполагаемой кодировке инструкций. Например, «оптимистическая» кодировка для трехадресной машины предполагает двухбайтовые кодировки для всех арифметических инструкций *op* `r`(i), `r`(j), `r`(k). Каждая арифметическая инструкция будет потреблять порцию $16/256 = 1/16$ пространства опкода. Обратите внимание, что для стековой машины мы приняли однобайтовые кодировки для `XCHG s`(i), `PUSH s`(i) и `POP s`(i) во всех случаях, дополненные `XCHG s1,s`(i) только для базового случая инструкций стека. Что касается операций с составным стеком, мы предположили двухбайтовые кодировки для `PUSH3`, `XCHG3`, `XCHG2`, `XCPU`, `PUXC`, `PUSH2`, но не для `XCHG s1,s`(i).

	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

Таблица 1: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1-адресных и стековых машин, сгенерированных для образца конечной функции (см. С.1.1). Два наиболее важных столбца, отражающих плотность кода и расширяемость для других операций, выделены жирным шрифтом. Меньшие значения лучше в обоих этих столбцах.

Столбец "байты кода" отражает плотность кода для конкретного примера источника. Однако «пространство опкода» также важно, поскольку оно отражает расширяемость достигнутой плотности на другие классы операций (например, если бы кто-то дополнял арифметические операции операциями манипулирования строками и так далее). Здесь подколонка «арифметика» более важна, чем подколонка «данные», потому что для таких расширений не потребуется никаких дальнейших операций манипулирования данными.

Мы видим, что трехадресная регистровая машина с «оптимистичной» кодировкой, предполагающая двухбайтовые кодировки для всех трехрегистровых арифметических операций, достигает наилучшей плотности кода, требуя всего 23 байта. Однако это имеет свою цену: каждая арифметическая операция занимает 1/16 пространства опкода, поэтому четыре операции уже используют четверть пространства опкода. Не более 11 других операций, арифметических или нет, могут быть добавлены к этой архитектуре при сохранении такой высокой плотности кода. С другой стороны, когда мы рассматриваем «реалистичную» кодировку для трехадресной машины, использующую двухбайтовые кодировки только для наиболее часто используемых операций сложения/вычитания (и более длинные кодировки для менее часто используемых операций умножения/деления, отражающие тот факт, что возможные операции расширения, вероятно, относятся к этому классу), то трехадресная машина перестает предлагать такую привлекательную плотность кода.

Фактически, двухадресная машина становится одинаково привлекательной в этот момент: она способна достичь того же размера кода в 31 байт, что и трехадресная машина с «реалистичной» кодировкой, используя для этого только 6/256 пространства опкода! Однако 31 байт — худший результат в этой таблице.

Машина с одним адресом использует 29 байт, что немного меньше, чем машина с двумя адресами. Тем не менее, он использует четверть пространства опкода для своих арифметических операций, препятствуя его расширяемости. В этом отношении он похож на трехадресную машину с «оптимистичной» кодировкой, но требует 29 байт вместо 23! Таким образом, нет никаких оснований использовать одноадресную машину вообще, с точки зрения расширяемости (отраженной пространством опкода, используемым для арифметических операций) по сравнению с плотностью кода.

Наконец, стековая машина выигрывает конкуренцию по плотности кода (27 или 28 байт), уступая только трехадресной машине с «оптимистичной» кодировкой (что, впрочем, ужасно с точки зрения расширяемости).

Подводя итог: двухадресная машина и стековая машина достигают наилучшей расширяемости по отношению к дополнительным

арифметическим инструкциям или инструкциям по обработке данных (используя только 1/256 кодового пространства для каждой такой инструкции), в то время как стековая машина дополнительно достигает наилучшей плотности кода с небольшим отрывом. Стековая машина использует значительную часть своего кодового пространства (более четверти) для инструкций по манипулированию данными (т.е. стеком); однако это не сильно затрудняет расширяемость, поскольку инструкции по манипулированию стеком занимают постоянную часть opcode stace, независимо от всех остальных инструкций и расширений.

Хотя все еще может возникнуть соблазн использовать машину с двумя адресами, мы кратко объясним (ср. C.3), почему машина регистров с двумя адресами предлагает худшую плотность кода и расширяемость на практике, чем кажется на основе этой таблицы.

Что касается выбора между стековой машиной с базовыми примитивами манипулирования стеком или одной поддерживающей составные примитивные константы стека, то аргументы в пользу более сложной машины стека кажутся слабее: она предлагает только на один или два байта меньше кода за счет использования значительно большего пространства опкода для манипулирования стеком, и оптимизированный код, использующий эти дополнительные инструкции, трудно для программистов написать и для компиляторов автоматически генерировать.

C.2.1. Соглашения о вызовах регистров: некоторые регистры должны быть сохранены функциями. До этого момента мы рассматривали машинный код только одной функции, не принимая во внимание взаимодействие между этой функцией и другими функциями в той же программе.

Обычно программа состоит из более чем одной функции, и когда функция не является «простой» или «листовой» функцией, она должна вызывать другие функции. Поэтому становится важным, сохраняет ли вызываемая функция все или хотя бы некоторые регистры. Если он сохраняет все регистры, кроме тех, которые используются для возврата результатов, вызывающий абонент может безопасно хранить свои локальные и временные переменные в определенных регистрах; однако вызываемому абоненту необходимо сохранить все регистры, которые он будет использовать для своих временных значений где-нибудь (обычно в

стеке, который также существует на регистровых машинах), а затем восстановить исходные значения. С другой стороны, если вызываемой функции разрешено уничтожать все регистры, она может быть записана способом, описанным в С.1.2, С.1.3 и С.1.4, но теперь вызывающий абонент будет отвечать за сохранение всех своих временных значений в стеке до вызова и восстановление этих значений впоследствии.

В большинстве случаев соглашения о вызовах для реестровых машин требуют сохранения некоторых, но не всех регистров. Мы предположим, что $m \leq n$ регистров будут сохранены функциями (если они не используются для возвращаемых значений), и что эти регистры являются $\texttt{r}(n-m)\ldots\ \texttt{r}(n-1)$. Кейс $m = 0$ соответствует рассмотренному до сих пор случаю «звонящий волен уничтожать все регистры»; это довольно болезненно для вызывающего абонента. Случай $m = n$ соответствует случаю «вызываемый должен сохранить все регистры»; это довольно болезненно для вызываемого, как мы увидим через мгновение. Обычно на практике используется *значение* m около $n/2$.

В следующих разделах рассматриваются случаи $m = 0$, $m = 8$ и $m = 16$ для наших регистровых машин с $n = 16$ регистров.

С.2.2. Случай $m = 0$: отсутствие регистров для сохранения. Этот случай был рассмотрен и кратко изложен в С.2 и таблице 1 выше.

С.2.3. Случай $m = n = 16$: все регистры должны быть сохранены. Этот случай является наиболее болезненным для вызываемой функции. Это особенно трудно для листовых функций, подобных той, которую мы рассматривали, которые вообще не выигрывают от того факта, что другие функции сохраняют некоторые регистры при вызове — они не вызывают никаких функций, а вместо этого должны сохранять все регистры сами. Чтобы оценить последствия допущения $m = n = 16$, мы предположим, что все наши регистровые машины оснащены стеком и однобайтовыми инструкциями `PUSH` $\texttt{r}(i)$ и `POP` $\texttt{r}(i)$,которые толкают или вставляют регистр в/из стека. Например, трехадресный машинный код, приведенный в С.1.2, уничтожает значения в регистрах `r2`, `r3`, `r6` и `r7`; это означает, что код этой функции должен быть дополнен четырьмя инструкциями `PUSH r2`; `PUSH r3`; `PUSH r6`; `PUSH r7` в начале и по четырем инструкциям `POP r7`; `POP r6`; `POP r3`; `POP r2` непосредственно перед инструкцией `RET`, чтобы восстановить исходные значения этих регистров

из стека. Эти четыре дополнительные пары `PUSH`/`POP` увеличат количество операций и размер кода в байтах на $4 \times 2 = 8$. Аналогичный анализ может быть сделан для другого регистра2. машины также, ведущие к Таблице 2.

		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

Таблица 2: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1адресных и стековых машин, сгенерированных для образца конечной функции (ср. C.1.1), предполагающая, что все 16 регистров должны быть сохранены вызываемыми функциями ($m = n = 16$). Новый столбец с меткой r обозначает количество регистров, которые необходимо сохранить и восстановить, что приводит к увеличению количества операций и байтов кода на 2r по сравнению с таблицей 1. Недавно добавленные `инструкции PUSH` и `POP` для регистровых машин также используют 32/256 пространства опкода. Две строки, соответствующие стековым машинам, остаются неизменными.

Мы видим, что при этих предположениях стековые машины являются очевидными победителями с точки зрения плотности кода и находятся в группе победителей в отношении расширяемости.

C.2.4. Случай $m = 8$, $n = 16$: регистры `r8`... `r15` должен быть сохранен. Анализ этого случая аналогичен предыдущему. Результаты обобщены в таблице 3.

Обратите внимание, что результирующая таблица очень похожа на таблицу 1, за исключением столбцов «Пространство опкода» и строки для одноадресной машины. Поэтому выводы C.2 по-прежнему применимы в данном случае с некоторыми незначительными изменениями. Однако мы должны подчеркнуть, что *эти выводы справедливы только для листовых функций, т. е. функций, которые не вызывают другие функции*. Любая программа, кроме самой простой, будет иметь много невневых функций, особенно если мы минимизируем результирующий размер машинного кода (что предотвращает встраивание функций в большинстве случаев).

C.2.5. Более справедливое сравнение с использованием двоичного кода вместо байтового кода. Читатель, возможно, заметил, что наше предыдущее обсуждение k-адресной машины регистра и машины стека очень сильно зависели от нашего настойчивого требования о том, чтобы полные инструкции кодировались целым числом байтов. Если бы нам разрешили использовать «бит» или «двоичный код» вместо байтового кода для кодирования инструкций, мы могли бы более равномерно сбалансировать пространство опкода, используемое различными машинами. Например, опкод `SUB` для машины с тремя адресами должен был быть либо 4-битным (хорошо для плотности кода, плохо для пространства опкода), либо 12-битным (очень плохо для плотности кода), потому что полная инструкция должна занимать кратное восьми битам (например, 16 или 24 бита), и $3 \cdot 4 = 12$ из этих битов должны использоваться для трех имен регистров.

		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

Таблица 3: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1-адресных и стековых машин, сгенерированных для образца конечной функции (ср. C.1.1), предполагая, что только последние 8 из 16 регистров должны быть сохранены вызываемыми функциями ($m = 8$, $n = 16$). Эта таблица аналогична таблице 2, но имеет меньшие значения r.

Поэтому давайте избавимся от этого ограничения.

Теперь, когда мы можем использовать любое количество битов для кодирования инструкции, мы можем выбрать все опкоды одинаковой длины для всех рассматриваемых машин. Например, все арифметические инструкции могут иметь 8-битные опкоды, как это делает стековая машина, используя $1/256$ пространства опкода каждая; тогда машина с тремя адресными регистрами будет использовать 20 бит для кодирования каждой полной арифметической инструкции. Можно предположить, что все `MOV`, `XCG`, `PUSHes и POP` на регистровых машинах имеют 4-

битные опкоды, потому что это то, что мы делаем для наиболее распространенных примитивов манипулирования стеком на стековой машине. Результаты этих изменений приведены в таблице 4.

Мы видим, что производительность различных машин гораздо более сбалансирована, причем машина стека по-прежнему выигрывает с точки зрения плотности кода, но с машиной с тремя адресами, занимающей второе место, она действительно заслуживает. Если бы мы рассмотрели скорость декодирования и возможность параллельного выполнения инструкций, нам пришлось бы выбрать трехадресную машину, потому что она использует только 12 инструкций вместо 21.

		Operations			Code bytes			Opcode space		
Machine	r	data	arith	total	data	arith	**total**	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

Таблица 4: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1-адресных и стековых машин, сгенерированных для образца конечной функции (ср. C.1.1), предполагая, что только 8 из 16 регистров должны быть сохранены функциями (m = 8, n = 16). На этот раз мы можем использовать доли байтов для кодирования инструкций, чтобы соответствовать пространству опкода, используемому различными машинами. Все арифметические инструкции имеют 8-битные опкоды, все инструкции манипулирования данными/стеком имеют 4-битные опкоды. В остальном эта таблица аналогична таблице 3.

C.3 Образец нелистовой функции

В этом разделе сравнивается машинный код для различных регистровых машин для примера нелистовой функции. Опять же, мы предполагаем, что либо $m = 0$, $m = 8$, либо $m = 16$ регистров сохраняются вызываемыми функциями, причем $m = 8$ представляет собой компромисс, достигнутый большинством современных компиляторов и операционных систем.

C.3.1. Пример исходного кода для нелистовой функции. Образец исходного файла может быть получен путем замены встроенного целочисленного типа пользовательским типом *Rational* , представленным указателем на объект в памяти, в нашей функции для решения систем двух линейных уравнений (ср. 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
}
```

Мы проигнорируем все вопросы, связанные с выделением новых объектов типа *Rational* в памяти (например, в куче) и предотвращением утечек памяти. Мы можем предположить, что называемые подпрограммы `r_sub`, `r_mul` и т. д. выделяют новые объекты, просто продвигая некоторый указатель в предварительно выделенном буфере, и что неиспользуемые объекты позже освобождаются сборщиком мусора, внешним по отношению к анализируемому коду.

Рациональные числа теперь будут представлены указателями, адресами или ссылками, которые будут вписываться в регистры наших гипотетических регистровых машин или в стек наших стековых машин. Если мы хотим использовать TVM в качестве экземпляра этих стековых машин, мы должны использовать значения типа *Cell* для представления таких ссылок на объекты типа *Rational* в памяти.

Мы предполагаем, что подпрограммы (или функции) вызываются специальной инструкцией `CALL` , которая кодируется тремя байтами, включая спецификацию вызываемой функции (например, индекс в «глобальной таблице функций»).

С.3.2. Трехадресные и двухадресные реестровые машины, $m = 0$ сохранившихся регистров. Поскольку наш пример функции вообще не использует встроенные арифметические инструкции, компиляторы для наших гипотетических машин с тремя адресами и двумя адресами будут производить один и тот же машинный код. Помимо ранее представленных однобайтовых инструкций `PUSH r(`i`)` и `POP r(`i`)`, мы предполагаем, что наши двух- и трехадресные машины поддерживают следующие двухбайтовые инструкции: `MOV r(`i`),s(`j`)`, `MOV s(`j`),r(` i`)`, и `XCHG r(`i`),s(`j`)`, для $0 \le i,j \le 15$. Такие инструкции занимают всего 3/256 пространства опкода, поэтому их добавление кажется вполне естественным.

Сначала мы предполагаем, что $m = 0$ (т. е. что все подпрограммы могут свободно уничтожать значения всех регистров). В этом случае наш машинный код для `r_f` не должен сохранять какие-либо регистры, но должен сохранять все регистры, содержащие полезные значения, в стек перед вызовом каких-либо подпрограмм. Компилятор, оптимизирующий размер, может выдавать следующий код:

```
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 p1,c0  // 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
```

Мы использовали 41 инструкцию: 17 однобайтовых (восемь пар `PUSH`/`POP` и одна `RET`), 13 двухбайтовых (`MOV` и `XCHG`; из них 11 «новых», включающих стек) и 11 трехбайтовых (`CALL`), в общей сложности $17 \cdot 1 + 13 \cdot 2 + 11 \cdot 3 = 76$ байт.[32]

С.3.3. Трехадресные и двухадресные реестровые машины, $m = 8$ сохранившихся регистров. Теперь у нас есть восемь регистров, от `r8` до `r15`, которые сохраняются вызовами подпрограмм. Мы могли бы сохранить некоторые промежуточные значения там, вместо того, чтобы выталкивать их в стек. Однако штраф за это заключается в паре `PUSH`/`POP` для каждого такого регистра, который мы выбираем для использования,

[32] Код, созданный для этой функции оптимизирующим компилятором для архитектуры x86-64

потому что наша функция также должна сохранять свое исходное значение. Похоже, что использование этих регистров под таким штрафом не улучшает плотность кода, поэтому оптимальный код для трех- и двухадресных машин для $m = 8$ сохраненных регистров такой же, как и в С.3.2, с общим количеством 42 инструкций и 74 кодовых байтов.

С.3.4. Трехадресные и двухадресные регистровые машины, $m = 16$ сохранившихся регистров. На этот раз *все* регистры должны быть сохранены подпрограммами, за исключением тех, которые используются для возврата результатов. Это означает, что наш код должен сохранять исходные значения от `r2` до `r5`, а также любые другие регистры, которые он использует для временных значений. Простым способом написания кода нашей подпрограммы было бы протолкнуть регистры `r2` , скажем, до `r8` в стек, затем выполнить все необходимые операции, используя `r6–r8` для промежуточных значений, и, наконец, восстановить регистры из стека. Однако это не оптимизирует размер кода. Мы выбираем другой подход:

```
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 Д
MOV r1,r5  // f
CALL r_mul // bf
PUSH r0    //  .. a D bf
```

с включенной оптимизацией размера фактически занято 150 байт, в основном из-за того, что фактические кодировки инструкций примерно в два раза длиннее, чем мы оптимистично предполагали.

```
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
```

Мы использовали 39 инструкций: 11 однобайтовых, 17 двухбайтовых (среди них 5 «новых» инструкций) и 11 трехбайтовых, всего $11 \cdot 1 + 17 \cdot 2 + 11 \cdot 3 = 78$ байт. Несколько парадоксально, что размер кода в байтах немного больше, чем в предыдущем случае (см. С.3.2), вопреки тому, что можно было ожидать. Отчасти это связано с тем, что мы приняли двухбайтовые кодировки для «новых» инструкций `MOV` и `XCHG` с участием стека, аналогично «старым» инструкциям. Большинство существующих архитектур (таких как x86-64) используют более длинные кодировки (возможно, даже в два раза длиннее) для своих аналогов наших «новых» инструкций перемещения и обмена по сравнению с «обычными» регистр-регистрами. Принимая это во внимание, мы видим,

что получили бы здесь 83 байта (против 87 для кода в С.3.2), предполагая трехбайтовые кодировки новых операций, и 88 байт (против 98) при четырехбайтовых кодировках. Это показывает, что для архитектур с двумя адресами без оптимизированных кодировок для операций перемещения и обмена стеком регистров $m = 16$ сохраненных регистров может привести к несколько более короткому коду для некоторых невых функций за счет конечных функций (см. С.2.3 и С.2.4), которые станут значительно длиннее.

С.3.5. Одноадресная регистровая машина, $m = 0$ сохранившихся регистров. Для нашей одноадресной регистровой машины мы предполагаем, что новые инструкции стека регистров работают только через аккумулятор. Таким образом, у нас есть три новые инструкции: `LD s(`*j*`)` (эквивалент `MOV r0,s(`*j*`)` машин с двумя адресами), `ST s(`*j*`)` (эквивалент `MOV s(`*j*`),r0`) и `XCHG s(`*j*`)` (эквивалент `XCHG r0,s(` *j*`)`). Чтобы сделать сравнение с машинами с двумя адресами более интересным, мы предполагаем однобайтовые кодировки для этих новых инструкций, хотя это будет занимать $48/256 = 3/16$ пространства опкода.

Адаптируя код, приведенный в С.3.2, к одноадресной машине, мы получаем следующее:

```
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 c1       //  r0:=c
XCHG r1     //  r0:=b, r1:=c
CALLr_mul  // bc
PUSH r0    //   .. e b a f c d bc
LD c1       //  d
XCHG r1     //  r1:=d
LD c4       //  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 c2       //  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 c5      // 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
```

Мы использовали 45 инструкций: 34 однобайтовых и 11 трехбайтовых, в общей сложности 67 байт. По сравнению с 76 байтами, используемыми машинами с двумя и тремя адресами в С.3.2, мы видим, что, опять же, машинный код регистра с одним адресом может быть плотнее, чем у машин с двумя регистрами, за счет использования большего пространства опкода (как показано в С.2). Однако на этот раз дополнительные 3/16 пространства опкода были использованы для инструкций по манипулированию данными, которые не зависят от конкретных арифметических операций или вызываемых пользовательских функций.

С.3.6. Одноадресная регистровая машина, $m = 8$ сохранившихся регистров. Как объясняется в С.3.3, сохранение `r8–r15` между вызовами подпрограмм не улучшает размер нашего ранее написанного кода, поэтому одноадресная машина будет использовать для $m = 8$ тот же код, что и в С.3.5.

С.3.7. Одноадресная регистровая машина, $m = 16$ сохранившихся регистров. Мы просто адаптируем код, приведенный в С.3.4, к одноадресному регистру:

```
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
```

Мы использовали 40 инструкций: 18 однобайтовых, 11 двухбайтовых и 11 трехбайтовых, в общей сложности $18 \cdot 1 + 11 \cdot 2 + 11 \cdot 3 = 73$ байта.

С.3.8. Стековая машина с базовыми примитивами стека. Мы повторно используем код, приведенный в С.1.5, просто заменяя арифметические примитивы (инструкции VM) вызовами подпрограмм. Единственным существенным изменением является вставка ранее необязательного `XCHG s1` перед третьим умножением, потому что даже оптимизирующий компилятор теперь не может знать, является ли `CALL r_mul` коммутативной операцией. Мы также использовали «оптимизацию хвостовой рекурсии», заменив окончательную `r_div CALL`, за которой следует `RET` на `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
r_div JMP  // x y
```

Мы использовали 29 инструкций; предполагая однобайтовые кодировки для всех операций стека и трехбайтовые кодировки для инструкций CALL и JMP, мы получаем 51 байт.

С.3.9. Стековая машина со составными примитивами стека. Мы снова повторно используем код, представленный в С.1.7, заменяя арифметические примитивы вызовами подпрограмм и оптимизируя хвостовую рекурсию:

```
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 Ди Д
r_div JMP      // x y
```

Этот код использует только 20 инструкций, 9 связанных со стеком и 11 связанных с потоком управления (`CALL` и `JMP`), в общей сложности 48 байт.

		Operations			Code bytes			Opcode space		
Machine	m	data	cont.	total	data	cont.	**total**	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

Таблица 5: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1-адресных и стековых машин, сгенерированных для образца невневой функции (см. С.3.1), предполагая, что m из 16 регистров должны быть сохранены с помощью подпрограмм, называемых подпрограммами.

C.4 Сравнение машинного кода для примера нелистовой функции

В таблице 5 обобщены свойства машинного кода, соответствующие тому же исходному файлу, приведенному в C.3.1. Мы рассматриваем только «реалистично» закодированные трехадресные машины. Машины с тремя и двумя адресами имеют одинаковые свойства плотности кода, но отличаются использованием пространства опкода. Одноадресная машина, что несколько удивительно, сумела создать более короткий код, чем машины с двумя адресами и тремя адресами, за счет использования более половины всего пространства опкода. Стековая машина является очевидным победителем в этом конкурсе плотности кода, не компрометируя ее отличную расширяемость (измеряется в пространстве опкода, используемом для арифметических и других инструкций преобразования данных).

C.4.1. Объединение с результатами для листовых функций. Поучительно сравнить эту таблицу с результатами B.2 для выборочной листовой функции, обобщенными в Таблице 1 (для $m = 0$ сохранившихся регистров) и очень похожей Таблице 3 (для $m = 8$ сохраненных регистров), и, если кого-то все еще интересует случай $m = 16$ (который оказался хуже $m = 8$ почти во всех ситуациях), также к таблице 2.

Мы видим, что стековая машина превосходит все регистровые машины по нелистовым функциям. Что касается листовых функций, то только трехадресная машина с «оптимистичным» кодированием арифметических инструкций смогла превзойти стековую машину, выиграв на 15%, поставив под угрозу ее расширяемость. Однако та же машина с тремя адресами производит на 25% более длинный код для нелистовых функций.

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

Таблица 6: Сводка свойств машинного кода для гипотетических 3-адресных, 2-адресных, 1-адресных и стековых машин, сгенерированных для примера нелистовой функции (ср. С.3.1), предполагающая, что m из 16 регистров должны быть сохранены с помощью так называемых подпрограмм. На этот раз мы используем доли байтов для кодирования инструкций, что позволяет проводить более справедливое сравнение. В остальном эта таблица аналогична таблице 5.

Если типичная программа состоит из смеси листовых и нелистовых функций примерно в равной пропорции, то машина штабеля все равно выиграет.

С.4.2. Более справедливое сравнение с использованием двоичного кода вместо байтового кода. Подобно С.2.5, мы можем предложить более справедливое сравнение различных регистровых машин и стековой машины, используя произвольные двоичные коды вместо байтовых кодов для кодирования инструкций и сопоставляя пространство опкода, используемое для обработки данных и арифметических инструкций различными машинами. Результаты этого модифицированного сопоставления обобщены в таблице 6. Мы видим, что стековые машины по-прежнему выигрывают с большим отрывом, используя при этом меньше пространства опкода для манипулирования стеком / данными.

С.4.3. Сравнение с реальными машинами. Обратите внимание, что наши гипотетические регистровые машины были значительно оптимизированы для получения более короткого кода, чем фактически существующие регистровые машины; последние зависят от других конструктивных соображений, помимо плотности и расширяемости кода, таких как обратная совместимость, более быстрое декодирование инструкций, параллельное выполнение соседних инструкций, простота

автоматического создания оптимизированного кода компиляторами и так далее.

Например, очень популярная архитектура регистров с двумя адресами x86-64 создает код, который примерно в два раза длиннее наших «идеальных» результатов для машин с двумя адресами. С другой стороны, наши результаты для стековых машин непосредственно применимы к TVM, который был явно разработан с учетом соображений, представленных в этом приложении. Кроме того, фактический код TVM еще *короче* (в байтах), чем показано в таблице 5, из-за наличия двухбайтовой инструкции `CALL` , что позволяет TVM вызывать до 256 пользовательских функций из словаря по адресу `c3`. Это означает, что следует вычесть 10 байт из результатов для стековых машин в таблице 5, если вы хотите конкретно рассмотреть TVM, а не абстрактную машину стека; это приводит к размеру кода примерно 40 байт (или короче), что почти вдвое меньше, чем у абстрактной машины с двумя или тремя адресами.

С.4.4. Автоматическая генерация оптимизированного кода. Интересным моментом является то, что машинный код стека в наших примерах мог быть сгенерирован автоматически очень простым оптимизирующим компилятором, который соответствующим образом переупорядочивает значения в верхней части стека перед вызовом каждого примитива или вызовом функции, как описано в 2.2.2 и 2.2.5. Единственным исключением является неважная «ручная» `оптимизация XCHG3`, описанная в С.1.7, которая позволила нам сократить код еще на один байт.

Напротив, сильно оптимизированный (по отношению к размеру) код для регистровых машин, показанный в С.3.2 и С.3.3, вряд ли будет автоматически создаваться оптимизирующим компилятором. Поэтому, если бы мы сравнили код, созданный компилятором, вместо кода, сгенерированного вручную, преимущества стековых машин в отношении плотности кода были бы еще более поразительными.