Telegram Open Network 虛擬機

Nikolai Durov

November 1, 2025

Abstract

本文旨在描述 Telegram Open Network 虛擬機（TON VM 或 TVM），用於在 TON 區塊鏈中執行智慧合約。

介紹

Telegram Open Network 虛擬機（TON VM 或 TVM）的主要目的是在 TON 區塊鏈中執行智慧合約程式碼。TVM 必須支援解析傳入訊息和持久資料所需的所有操作，以及建立新訊息和修改持久資料。

此外，TVM 必須滿足以下要求：

- 它必須提供未來可能的擴充和改進，同時保持向後相容性和互通性，因為智慧合約的程式碼一旦提交到區塊鏈中，無論未來對 VM 進行任何修改，都必須繼續以可預測的方式工作。
- 它必須努力實現高「（虛擬）機器程式碼」密度，以便典型智慧合約的程式碼佔用盡可能少的持久區塊鏈儲存空間。
- 它必須是完全確定性的。換句話說，使用相同輸入資料多次執行相同程式碼必須產生相同結果，無論使用的特定軟體和硬體如何。[1]

TVM 的設計以這些要求為指導。雖然本文件描述了 TVM 的初步和實驗版本，[2]但系統內建的向後相容性機制讓我們相對不用擔心此初步版本中用於 TVM 程式碼的操作編碼效率。

[1]例如，TVM 中不存在浮點算術操作（可以在大多數現代 CPU 上使用硬體支援的 *double* 型別有效實作），因為執行此類操作的結果取決於特定的底層硬體實作和舍入模式設定。相反，TVM 支援特殊的整數算術操作，如果需要可以用來模擬定點算術。

[2]生產版本在啟動之前可能需要一些調整和修改，這只有在測試環境中使用實驗版本一段時間後才會變得明顯。

TVM 不打算在硬體中實作（例如，在專用微處理器晶片中）；相反，它應該在執行於傳統硬體上的軟體中實作。這種考慮讓我們能夠在 TVM 中納入一些高階概念和操作，這些在硬體實作中需要複雜的微程式碼，但對軟體實作不會造成重大問題。這些操作對於實現高程式碼密度和在 TON 區塊鏈中部署時最小化智慧合約程式碼的位元組（或儲存單元）概況非常有用。

Contents

1 概述

本章概述了 TVM 的主要特性和設計原則。後續章節將提供每個主題的更多細節。

1.0 位元字串的表示法

本文件中對位元字串（或 *bitstring*）——即由二進位數字（位元）0 和 1 組成的有限字串——使用以下表示法。

1.0.1. 位元字串的十六進位表示法. 當位元字串的長度是四的倍數時，我們將其細分為每四個位元一組，並以通常的方式用十六個十六進位數字 0–9、A–F 之一表示每組：$0_{16} \leftrightarrow$ 0000，$1_{16} \leftrightarrow$ 0001，…，$F_{16} \leftrightarrow$ 1111。產生的十六進位字串是我們對原始二進位字串的等效表示。

1.0.2. 長度不能被四整除的位元字串. 如果二進位字串的長度不能被四整除，我們在末尾添加一個 1 和若干個（也許是零個）0，使其長度能被四整除，然後如上所述將其轉換為十六進位數字字串。為了表示已經進行了這樣的轉換，在十六進位字串的末尾添加一個特殊的「完成標記」_。反向轉換（如果存在完成標記則應用）首先將每個十六進位數字替換為四個對應的位元，然後刪除所有尾隨零（如果有的話）和緊接在它們之前的最後一個 1（如果此時產生的位元字串非空）。

請注意，同一個位元字串有多個可接受的十六進位表示。其中，最短的是「正則的」。它可以透過上述程序確定性地獲得。

例如，8A 對應於二進位字串 10001010，而 8A_ 和 8A0_ 都對應於 100010。空位元字串可以由 ‘’、‘8_’、‘0_’、‘_’ 或 ‘00_’ 表示。

1.0.3. 強調字串是位元字串的十六進位表示. 有時我們需要強調十六進位數字字串（末尾有或沒有 _）是位元字串的十六進位表示。在這種情況下，我們要麼在產生的字串前面加上 x（例如，x8A），要麼在前面加上 x{ 並在後面加上 }（例如，x{2D9_}，即 00101101100）。這不應與十六進位數字混淆，通常前面加上 0x（例如，0x2D9 或 0x2d9，即整數 729）。

1.0.4. 將位元字串序列化為八位元組序列. 當需要將位元字串表示為 8 位元位元組（八位元組）序列（其值為 0...255 的整數）時，基本上以與上述相同的方式實現：我們將位元字串分成每八個位元一組，並將每組解釋為整數 0...255 的二進位表示。如果位元字串的長度不是八的倍數，則在拆分成組之前，在末尾添加一個二進位 1 和最多七個（也許是零個）二進位 0，使其長度能被八整除。已經應用了這種補全的事實通常透過「補全標記」位元來反映。

例如，`00101101100` 對應於兩個八位元組的序列 (`0x2d`, `0x90`)（十六進位），或 $(45, 144)$（十進位），以及一個等於 `1` 的補全標記位元（表示已應用補全），該位元必須單獨儲存。

在某些情況下，假設預設啟用補全而不是單獨儲存額外的補全標記位元會更方便。在這種約定下，$8n$ 位元字串由 $n+1$ 個八位元組表示，最後一個八位元組始終等於 `0x80` $= 128$。

1.1 TVM 是堆疊機器

首先，*TVM* 是堆疊機器。這意味著，值不是保存在某些「變數」或「通用暫存器」中，而是保存在（LIFO）堆疊中，至少從「低階」（TVM）的角度來看是這樣。[3]

大多數操作和使用者定義函數從堆疊頂部取得它們的引數，並用它們的結果替換它們。例如，整數加法基本操作（內建操作）`ADD` 不接受任何描述應將哪些暫存器或立即值相加在一起以及結果應儲存在哪裡的引數。相反，從堆疊中取出最上面的兩個值，將它們相加，然後將它們的和推入堆疊以取代它們的位置。

1.1.1. TVM 值. 可以儲存在 TVM 堆疊中的實體將被稱為 *TVM* 值，或簡稱為值。它們屬於幾種預定義的值型別之一。每個值僅屬於一種值型別。值始終與唯一確定其型別的標記一起保存在堆疊中，並且所有內建 TVM 操作（或基本操作）僅接受預定義型別的值。

例如，整數加法基本操作 `ADD` 僅接受兩個整數值，並傳回一個整數值作為結果。不能向 `ADD` 提供兩個字串而不是兩個整數，並期望它連接這些字串或將字串隱式轉換為它們的十進位整數值；任何這樣做的嘗試都將導致執行期型別檢查異常。

1.1.2. 靜態型別、動態型別和執行期型別檢查. 在某些方面，TVM 使用執行期型別檢查執行一種動態型別。然而，這並不使 TVM 程式碼成為像 PHP 或 Javascript 這樣的「動態型別語言」，因為所有基本操作都接受並傳回預定義（值）型別的值，每個值僅屬於嚴格的一種型別，並且值永遠不會從一種型別隱式轉換為另一種型別。另一方面，如果將 TVM 程式碼與傳統的微處理器機器碼進行比較，就會發現 TVM 的值標記機制防止了例如將字串的位址用作數字——或者，可能更災難性的是，將數字用作字串的位址——從而消除了與無效記憶體存取相關的各種錯誤和安全漏洞的可能性，通常導致記憶體損壞和分段錯誤。對於用於在區塊鏈中執行智慧

[3]高階智慧合約語言可能會創建變數的可見性以便於編程；然而，使用變數的高階原始碼將被翻譯成 TVM 機器碼，將所有這些變數的值保存在 TVM 堆疊中。

合約的 VM，這個屬性是非常理想的。在這方面，TVM 堅持用適當的型別標記所有值，而不是根據操作中使用的需求重新解釋暫存器中的位元序列，這只是一個額外的執行期型別安全機制。

另一種方法是在允許執行智慧合約程式碼之前以某種方式分析智慧合約程式碼的型別正確性和型別安全性，甚至在允許將其上傳到區塊鏈作為智慧合約的程式碼之前。對圖靈完備機器的程式碼進行這樣的靜態分析似乎是一個耗時且非平凡的問題（可能等同於圖靈機的停機問題），我們寧願在區塊鏈智慧合約環境中避免這種情況。

應該記住，人們總是可以從靜態型別的高階智慧合約語言實作編譯器到 TVM 程式碼（我們確實期望 TON 的大多數智慧合約將用這樣的語言編寫），就像可以將靜態型別語言編譯成傳統的機器碼（例如，x86 架構）一樣。如果編譯器正常工作，產生的機器碼將永遠不會產生任何執行期型別檢查異常。附加到 TVM 處理的值的所有型別標記將始終具有預期值，並且在分析產生的 TVM 程式碼期間可以安全地忽略，除了 TVM 執行期產生和驗證這些型別標記的事實將稍微減慢 TVM 程式碼的執行速度。

1.1.3. 值型別的初步列表. TVM 支援的值型別初步列表如下：

- *Integer*（整數） — 有號 257 位元整數，表示範圍 $-2^{256} \ldots 2^{256}-1$ 內的整數，以及特殊的「非數字」值 `NaN`。
- *Cell*（單元） — *TVM* 單元由最多 1023 位元的資料和最多四個對其他單元的參照組成。TON 區塊鏈中的所有持久資料（包括 TVM 程式碼）都表示為 TVM 單元的集合（參見 [1, 2.5.14]）。
- *Tuple*（元組） — 最多 255 個元件的有序集合，具有任意值型別，可能不同。可用於表示任意代數資料型別的非持久值。
- *Null*（空） — 僅有一個值 $\perp$ 的型別，用於表示空列表、二元樹的空分支、某些情況下沒有傳回值等。
- *Slice*（切片） — *TVM* 單元切片，或簡稱為切片，是現有單元的連續「子單元」，包含其部分資料位元和部分參照。本質上，切片是單元的子單元的唯讀檢視。切片用於解包先前儲存（或序列化）在單元或單元樹中的資料。
- *Builder*（建構器） — *TVM* 單元建構器，或簡稱為建構器，是一個「不完整」的單元，支援在其末尾附加位元字串和單元參照的快速操作。建構器用於將堆疊頂部的資料打包（或序列化）到新單元中（例如，在將它們傳輸到持久儲存之前）。

- *Continuation*（繼續） — 表示 TVM 的「執行權杖」，可以稍後呼叫（執行）。因此，它概括了函數位址（即函數指標和參照）、子程式傳回位址、指令指標位址、異常處理器位址、閉包、部分應用、匿名函數等。

此值型別列表並不完整，並且可能在 TVM 的未來修訂版中擴展而不會破壞舊的 TVM 程式碼，這主要是因為所有原始定義的基本操作僅接受它們已知型別的值，並且如果在新型別的值上呼叫將失敗（產生型別檢查異常）。此外，現有的值型別本身也可以在將來擴展：例如，257 位元 *Integer* 可能會變成 513 位元 *LongInteger*，如果引數或結果不適合原始子型別 *Integer*，則原始定義的算術基本操作將失敗。關於引入新值型別和擴展現有值型別的向後相容性將在稍後更詳細地討論（參見 **5.1.4**）。

1.2 TVM 指令的類別

TVM 指令，也稱為基本操作，有時稱為（內建）操作，是 TVM 原子執行的可以出現在 TVM 程式碼中的最小操作。它們根據它們處理的值型別（參見 **1.1.3**）分為幾個類別。這些類別中最重要的是：

- 堆疊（操作）基本操作 — 重新排列 TVM 堆疊中的資料，以便稍後可以使用正確的引數呼叫其他基本操作和使用者定義的函數。與大多數其他基本操作不同，它們是多型的，即可以使用任意型別的值。
- 元組（操作）基本操作 — 建構、修改和分解 *Tuple*。與堆疊基本操作類似，它們是多型的。
- 常數或字面量基本操作 — 將嵌入 TVM 程式碼本身的一些「常數」或「字面量」值推入堆疊，從而為其他基本操作提供引數。它們在某種程度上類似於堆疊基本操作，但不太通用，因為它們使用特定型別的值。
- 算術基本操作 — 對 *Integer* 型別的值執行通常的整數算術操作。
- 單元（操作）基本操作 — 建立新單元並在其中儲存資料（單元建立基本操作）或從先前建立的單元中讀取資料（單元解析基本操作）。因為 TVM 的所有記憶體和持久儲存都由單元組成，所以這些單元操作基本操作實際上對應於其他架構的「記憶體存取指令」。單元建立基本操作通常使用 *Builder* 型別的值，而單元解析基本操作使用 *Slice*。
- *Continuation* 和控制流基本操作 — 以不同方式建立和修改 *Continuation*，以及執行現有的 *Continuation*，包括條件和重複執行。

- 自訂或應用程式特定基本操作 — 有效執行應用程式（在我們的情況下是 TON 區塊鏈）所需的特定高階操作，例如計算雜湊函數、執行橢圓曲線密碼學、傳送新的區塊鏈訊息、建立新的智慧合約等。這些基本操作對應於標準函式庫函數而不是微處理器指令。

1.3 控制暫存器

雖然 TVM 是堆疊機器，但在幾乎所有函數中需要的一些很少改變的值最好在某些特殊暫存器中傳遞，而不是在堆疊頂部附近。否則，將需要大量的堆疊重新排序操作來管理所有這些值。

為此，除了堆疊之外，TVM 模型還包括最多 16 個特殊的控制暫存器，由 c0 到 c15 表示，或 c(0) 到 c(15)。TVM 的原始版本僅使用其中一些暫存器；其餘的可能稍後支援。

1.3.1. 控制暫存器中保存的值. 控制暫存器中保存的值與堆疊中保存的值型別相同。但是，某些控制暫存器僅接受特定型別的值，任何載入不同型別的值的嘗試都將導致異常。

1.3.2. 控制暫存器列表. TVM 的原始版本定義並使用以下控制暫存器：

- c0 — 包含下一個繼續或傳回繼續（類似於傳統設計中的子程式傳回位址）。此值必須是 *Continuation*。
- c1 — 包含替代（傳回）繼續；此值必須是 *Continuation*。它用於某些（實驗性）控制流基本操作，允許 TVM 定義和呼叫「具有兩個出口點的子程式」。
- c2 — 包含異常處理器。此值是 *Continuation*，在觸發異常時呼叫。
- c3 — 包含目前字典，本質上是包含程式中使用的所有函數程式碼的雜湊映射。由於稍後在 **4.6** 中解釋的原因，此值也是 *Continuation*，而不是人們可能期望的 *Cell*。
- c4 — 包含持久資料的根，或簡稱為資料。此值是 *Cell*。當呼叫智慧合約的程式碼時，c4 指向其在區塊鏈狀態中保存的持久資料的根單元。如果智慧合約需要修改此資料，它會在傳回之前更改 c4。
- c5 — 包含輸出操作。它也是一個 *Cell*，由對空單元的參照初始化，但其最終值被視為智慧合約輸出之一。例如，特定於 TON 區塊鏈的 SENDMSG 基本操作只是將訊息插入儲存在輸出操作中的列表。

- c7 — 包含暫存資料的根。它是一個 *Tuple*，在呼叫智慧合約之前由對空 *Tuple* 的參照初始化，並在其終止後丟棄。[4]

如有必要，將來可能會為特定的 TON 區塊鏈或高階編程語言目的定義更多控制暫存器。

1.4 TVM 的總狀態（SCCCG）

TVM 的總狀態由以下元件組成：

- 堆疊（參見 **1.1**）— 包含零個或多個值（參見 **1.1.1**），每個都屬於 **1.1.3** 中列出的值型別之一。
- 控制暫存器 *c0*–*c15* — 包含 **1.3.2** 中描述的一些特定值。（目前版本僅使用七個控制暫存器。）
- 目前繼續 *cc* — 包含目前繼續（即，目前基本操作完成後通常會執行的程式碼）。此元件類似於其他架構中的指令指標暫存器（ip）。
- 目前程式碼頁 *cp* — 一個特殊的有號 16 位元整數值，用於選擇下一個 TVM 操作碼的解碼方式。例如，TVM 的未來版本可能使用不同的程式碼頁來添加新的操作碼，同時保持向後相容性。
- *Gas* 限制 *gas* — 包含四個有號 64 位元整數：目前 gas 限制 g_l、最大 gas 限制 g_m、剩餘 gas g_r 和 gas 信用 g_c。始終 $0 \leq g_l \leq g_m$、$g_c \geq 0$ 和 $g_r \leq g_l + g_c$；g_c 通常由零初始化，g_r 由 $g_l + g_c$ 初始化，並隨著 TVM 執行而逐漸減少。當 g_r 變為負數或 g_r 的最終值小於 g_c 時，觸發耗盡 *gas* 異常。

請注意，沒有包含所有先前呼叫但未完成函數的傳回位址的「傳回堆疊」。相反，僅使用控制暫存器 c0。這樣做的原因將在稍後的 **4.1.9** 中解釋。

還請注意，沒有通用暫存器，因為 TVM 是堆疊機器（參見 **1.1**）。因此，上面的列表可以概括為「堆疊、控制、繼續、程式碼頁和 gas」（SCCCG），類似於經典的 SECD 機器狀態（「堆疊、環境、控制、轉存」），確實是 TVM 的總狀態。[5]

[4] 在 TON 區塊鏈環境中，c7 用單例 *Tuple* 初始化，其唯一元件是包含區塊鏈特定資料的 *Tuple*。智慧合約可以自由修改 c7 以儲存其暫存資料，前提是此 *Tuple* 的第一個元件保持不變。

[5] 嚴格來說，還有目前的函式庫上下文，它由具有 256 位元鍵和單元值的字典組成，用於載入 **3.1.7** 的函式庫參照單元。

1.5 整數算術

TVM 的所有算術基本操作都對從堆疊頂部取得的幾個 *Integer* 型別的引數進行操作，並將相同型別的結果傳回堆疊。回想一下，*Integer* 表示範圍 $-2^{256} \leq x < 2^{256}$ 內的所有整數值，並且另外包含特殊值 NaN（「非數字」）。

如果其中一個結果不適合支援的整數範圍——或者如果引數之一是 NaN——那麼此結果或所有結果都將被 NaN 替換，並且（預設情況下）產生整數溢位異常。但是，算術操作的特殊「靜默」版本將簡單地產生 NaN 並繼續執行。如果這些 NaN 最終被用於「非靜默」算術操作或非算術操作中，則會發生整數溢位異常。

1.5.1. 整數的自動轉換缺失. 請注意，TVM *Integer* 是「數學」整數，而不是根據使用的基本操作而不同解釋的 257 位元字串，這對於其他機器碼設計來說是常見的。例如，TVM 只有一個乘法基本操作 MUL，而不是兩個（MUL 用於無號乘法，IMUL 用於有號乘法），例如在流行的 x86 架構中發生的那樣。

1.5.2. 自動溢位檢查. 請注意，所有 TVM 算術基本操作都執行結果的溢位檢查。如果結果不適合 *Integer* 型別，則將其替換為 NaN，並且（通常）發生異常。特別是，結果不會自動對 2^{256} 或 2^{257} 取模，這對於大多數硬體機器碼架構來說是常見的。

1.5.3. 自訂溢位檢查. 除了自動溢位檢查之外，TVM 還包括自訂溢位檢查，由基本操作 FITS n 和 UFITS n 執行，其中 $1 \leq n \leq 256$。這些基本操作檢查堆疊（頂部）上的值是否是範圍 $-2^{n-1} \leq x < 2^{n-1}$ 或 $0 \leq x < 2^n$ 內的整數 x，並在情況不是這樣時將值替換為 NaN 並（可選）產生整數溢位異常。這極大地簡化了任意 n 位元整數型別（有號或無號）的實作：程式設計師或編譯器必須在每個算術操作之後（這更合理，但需要更多檢查）或在儲存計算值並從函數傳回它們之前插入適當的 FITS 或 UFITS 基本操作。這對於智慧合約很重要，其中意外的整數溢位恰好是最常見的錯誤來源之一。

1.5.4. 對 2^n 取模. TVM 還有一個基本操作 MODPOW2 n，它將堆疊頂部的整數對 2^n 取模，結果範圍從 0 到 $2^n - 1$。

1.5.5. *Integer* 是 257 位元，而不是 256 位元. 現在可以理解為什麼 TVM 的 *Integer* 是（有號）257 位元，而不是 256 位元。原因是它是包含有號 256 位元整數和無號 256 位元整數的最小整數型別，不需要根據使用的操作自動重新解釋相同的 256 位元字串（參見 **1.5.1**）。

1.5.6. 除法和舍入. 最重要的除法基本操作是 DIV、MOD 和 DIVMOD。它們都從堆疊中取兩個數字，x 和 y（y 從堆疊頂部取出，x 原來在它下面），計算 x 除以 y 的商 q 和餘數 r（即，兩個整數使得 $x = yq + r$ 且 $|r| < |y|$），並傳回 q、r 或兩者。如果 y 為零，則所有預期結果都將被 NaN 替換，並且（通常）產生整數溢位異常。

TVM 中除法的實作在舍入方面與大多數其他實作有些不同。預設情況下，這些基本操作向 $-\infty$ 舍入，這意味著 $q = \lfloor x/y \rfloor$，並且 r 與 y 具有相同的符號。（大多數除法的傳統實作使用「向零舍入」，這意味著 r 與 x 具有相同的符號。）除了這個「向下舍入」之外，還提供了另外兩種舍入模式，稱為「向上舍入」（其中 $q = \lceil x/y \rceil$，r 和 y 具有相反的符號）和「最近舍入」（其中 $q = \lfloor x/y + 1/2 \rfloor$ 且 $|r| \leq |y|/2$）。透過使用其他除法基本操作選擇這些舍入模式，在它們的助記符中附加字母 C 和 R。例如，DIVMODR 使用向最近整數舍入計算商和餘數。

1.5.7. 組合乘除、乘移和移除操作. 為了簡化定點算術的實作，TVM 支援具有雙長度（即 514 位元）中間積的組合乘除、乘移和移除操作。例如，MULDIVMODR 從堆疊中取三個整數引數 a、b 和 c，首先使用 514 位元中間結果計算 ab，然後使用向最近整數舍入將 ab 除以 c。如果 c 為零或商不適合 *Integer*，則傳回兩個 NaN，或產生整數溢位異常，具體取決於是否使用了操作的靜默版本。否則，商和餘數都將推入堆疊。

2 堆疊

本章包含對暫存器和堆疊機器的一般討論和比較，在附錄 C 中進一步擴展，並描述 TVM 使用的堆疊操作基本操作的兩個主要類別：基本和複合堆疊操作基本操作。還提供了對它們對於正確呼叫其他基本操作和使用者定義函數所需的所有堆疊重新排序的充分性的非正式解釋。最後，在 **2.3** 中討論了有效實作 TVM 堆疊操作基本操作的問題。

2.1 堆疊呼叫約定

堆疊機器（例如 TVM）使用堆疊——尤其是堆疊頂部附近的值——將引數傳遞給呼叫的函數和基本操作（例如內建算術操作）並接收它們的結果。本節討論 TVM 堆疊呼叫約定，介紹一些表示法，並將 TVM 堆疊呼叫約定與某些暫存器機器的呼叫約定進行比較。

2.1.1. 「堆疊暫存器」的表示法. 回想一下，與更傳統的暫存器機器相反，堆疊機器缺少通用暫存器。但是，可以將堆疊頂部附近的值視為一種「堆疊暫存器」。

我們用 `s0` 或 $\mathtt{s}(0)$ 表示堆疊頂部的值，用 `s1` 或 $\mathtt{s}(1)$ 表示緊接在它下面的值，依此類推。堆疊中值的總數稱為其深度。如果堆疊的深度為 n，則 $\mathtt{s}(0)$、$\mathtt{s}(1)$、…、$\mathtt{s}(n-1)$ 是良好定義的，而 $\mathtt{s}(n)$ 和所有後續的 $\mathtt{s}(i)$（其中 $i > n$）則不是。任何使用 $\mathtt{s}(i)$（其中 $i \geq n$）的嘗試都應產生堆疊下溢異常。

編譯器或「TVM 程式碼」中的人類程式設計師將使用這些「堆疊暫存器」來保存所有宣告的變數和中間值，類似於在暫存器機器上使用通用暫存器的方式。

2.1.2. 推入和彈出值. 當值 x 被推入深度為 n 的堆疊時，它成為新的 `s0`；同時，舊的 `s0` 成為新的 `s1`，舊的 `s1`——新的 `s2`，依此類推。產生的堆疊的深度為 $n+1$。

類似地，當從深度為 $n \geq 1$ 的堆疊中彈出值 x 時，它是 `s0` 的舊值（即，堆疊頂部的舊值）。之後，它從堆疊中移除，舊的 `s1` 成為新的 `s0`（堆疊頂部的新值），舊的 `s2` 成為新的 `s1`，依此類推。產生的堆疊的深度為 $n-1$。

如果原來 $n = 0$，則堆疊為空，並且無法從中彈出值。如果基本操作嘗試從空堆疊彈出值，則發生堆疊下溢異常。

2.1.3. 假設的通用暫存器的表示法. 為了將堆疊機器與足夠通用的暫存器機器進行比較，我們將暫存器機器的通用暫存器表示為 `r0`、`r1` 等，或

r(0)、r(1)、…、r($n-1$)，其中 n 是暫存器的總數。當我們需要 n 的特定值時，我們將使用 $n = 16$，對應於非常流行的 x86-64 架構。

2.1.4. 堆疊頂部暫存器 s0 與累加器暫存器 r0. 某些暫存器機器架構要求大多數算術和邏輯操作的引數之一駐留在稱為累加器的特殊暫存器中。在我們的比較中，我們假設累加器是通用暫存器 r0；否則我們可以簡單地重新編號暫存器。在這方面，累加器有點類似於堆疊機器的堆疊頂部「暫存器」s0，因為堆疊機器的幾乎所有操作都使用 s0 作為其引數之一，並將其結果傳回為 s0。

2.1.5. 暫存器呼叫約定. 當為暫存器機器編譯時，高階語言函數通常以預定義順序在某些暫存器中接收其引數。如果引數太多，這些函數從堆疊中取得餘數（是的，暫存器機器通常也有堆疊 !）。但是，某些暫存器呼叫約定根本不在暫存器中傳遞引數，而只使用堆疊（例如，Pascal 和 C 的實作中使用的原始呼叫約定，儘管現代 C 的實作也使用某些暫存器）。

為簡單起見，我們假設最多 $m \leq n$ 個函數引數在暫存器中傳遞，並且這些暫存器依次為 r0、r1、…、r($m-1$)（如果使用其他暫存器，我們可以簡單地重新編號它們）。[6]

2.1.6. 函數引數的順序. 如果函數或基本操作需要 m 個引數 x_1、…、x_m，則呼叫者按相同順序將它們推入堆疊，從 x_1 開始。因此，當呼叫函數或基本操作時，其第一個引數 x_1 在 s($m-1$) 中，其第二個引數 x_2 在 s($m-2$) 中，依此類推。最後一個引數 x_m 在 s0 中（即，在堆疊頂部）。被呼叫的函數或基本操作負責從堆疊中移除其引數。

在這方面，TVM 堆疊呼叫約定——至少由 TVM 基本操作遵守——與 Pascal 和 Forth 的匹配，並且與 C 的相反（在 C 中，引數以相反的順序推入堆疊，並在呼叫者重新獲得控制後由呼叫者移除，而不是被呼叫者）。

當然，TVM 高階語言的實作可能會為其函數選擇一些其他呼叫約定，不同於預設約定。這對於某些函數可能很有用——例如，如果引數總數取決於第一個引數的值，就像「可變參數函數」（如 scanf 和 printf）那樣。在這種情況下，最好將第一個或幾個引數傳遞到堆疊頂部附近，而不是堆疊深處的某個未知位置。

2.1.7. 暫存器機器上算術基本操作的引數. 在堆疊機器上，內建算術基本操作（例如 ADD 或 DIVMOD）遵循與使用者定義函數相同的呼叫約定。在這方面，使用者定義的函數（例如，計算數字平方根的函數）可能被視為堆

[6]我們在這裡包含 r0 與我們假設累加器暫存器（如果存在）也是 r0 產生了一個小衝突；為簡單起見，我們將透過假設函數的第一個引數在累加器中傳遞來解決此問題。

疊機器的「擴充」或「自訂升級」。這是堆疊機器（以及堆疊編程語言如 Forth）相對於暫存器機器的最明顯優勢之一。

相比之下，暫存器機器上的算術指令（內建操作）通常從編碼在完整操作碼中的通用暫存器取得其引數。因此，二元操作（如 SUB）需要兩個引數 $\texttt{r}(i)$ 和 $\texttt{r}(j)$，其中 i 和 j 由指令指定。還必須指定用於儲存結果的暫存器 $\texttt{r}(k)$。算術操作可以採用幾種可能的形式，取決於是否允許 i、j 和 k 取任意值：

- 三位址形式 — 允許程式設計師不僅任意選擇兩個源暫存器 $\texttt{r}(i)$ 和 $\texttt{r}(j)$，還可以選擇單獨的目標暫存器 $\texttt{r}(k)$。這種形式對於大多數 RISC 處理器以及 x86-64 架構中的 XMM 和 AVX SIMD 指令集來說很常見。

- 兩位址形式 — 使用兩個運算元暫存器之一（通常是 $\texttt{r}(i)$）來儲存操作的結果，因此 $k = i$ 永遠不會明確指示。只有 i 和 j 在指令內部編碼。這是暫存器機器上算術操作的最常見形式，在微處理器上非常流行（包括 x86 家族）。

- 一位址形式 — 始終從累加器 r0 取得引數之一，並將結果也儲存在 r0 中；那麼 $i = k = 0$，只需要指令指定 j。這種形式被一些更簡單的微處理器使用（例如 Intel 8080）。

請注意，這種靈活性僅適用於內建操作，而不適用於使用者定義的函數。在這方面，暫存器機器不像堆疊機器那樣容易「可升級」。[7]

2.1.8. 函數的傳回值. 在堆疊機器（如 TVM）中，當函數或基本操作需要傳回結果值時，它只需將其推入堆疊（從中已移除函數的所有引數）。因此，呼叫者將能夠透過堆疊頂部「暫存器」s0 存取結果值。

這完全符合 Forth 呼叫約定，但與 Pascal 和 C 呼叫約定略有不同，後者通常使用累加器暫存器 r0 作為傳回值。

2.1.9. 傳回多個值. 某些函數可能希望傳回多個值 y_1、…、y_k，其中 k 不一定等於一。在這些情況下，k 個傳回值按其自然順序推入堆疊，從 y_1 開始。

例如，「帶餘數除法」基本操作 DIVMOD 需要傳回兩個值，商 q 和餘數 r。因此，DIVMOD 按該順序將 q 和 r 推入堆疊，以便之後商在 s1 中可用，餘數在 s0 中。DIVMOD 的淨效果是將 s1 的原始值除以 s0 的原始值，並在

[7]例如，如果編寫一個提取平方根的函數，此函數將始終在相同的暫存器中接受其引數並傳回其結果，與假設的內建平方根指令相反，後者可以允許程式設計師任意選擇源和目標暫存器。因此，在暫存器機器上，使用者定義的函數遠不如內建指令靈活。

s1 中傳回商，在 s0 中傳回餘數。在這種特殊情況下，堆疊的深度和所有其他「堆疊暫存器」的值保持不變，因為 DIVMOD 接受兩個引數並傳回兩個結果。通常，位於傳遞的引數和傳回的值下面的堆疊中的其他「堆疊暫存器」的值會根據堆疊深度的變化而移位。

原則上，某些基本操作和使用者定義的函數可能傳回可變數量的結果值。在這方面，關於可變參數函數的上述備註（參見 **2.1.6**）適用：結果值的總數及其型別應由堆疊頂部附近的值確定。（例如，可以推入傳回值 y_1、…、y_k，然後將其總數 k 作為整數推入。呼叫者然後透過檢查 s0 來確定傳回值的總數。）

在這方面，TVM 再次忠實地遵守 Forth 呼叫約定。

2.1.10. 堆疊表示法. 當深度為 n 的堆疊按該順序包含值 z_1、…、z_n 時，其中 z_1 是最深的元素，z_n 是堆疊頂部，堆疊的內容通常按該順序表示為列表 $z_1\ z_2\ \ldots z_n$。當基本操作將原始堆疊狀態 S' 轉換為新狀態 S'' 時，這通常寫為 $S' - S''$；這就是所謂的堆疊表示法。例如，除法基本操作 DIV 的動作可以描述為 $S\ x\ y - S\ \lfloor x/y \rfloor$，其中 S 是任何值列表。這通常簡寫為 $x\ y - \lfloor x/y \rfloor$，隱含假設堆疊深處的所有其他值保持不變。

或者，可以將 DIV 描述為在深度為 $n \geq 2$ 的堆疊 S' 上執行的基本操作，將 s1 除以 s0，並將向下舍入的商傳回為深度為 $n-1$ 的新堆疊 S'' 的 s0。$\mathtt{s}(i)$ 的新值等於 $\mathtt{s}(i+1)$ 的舊值，對於 $1 \leq i < n-1$。這些描述是等效的，但說 DIV 將 $x\ y$ 轉換為 $\lfloor x/y \rfloor$，或將 $\ldots x\ y$ 轉換為 $\ldots \lfloor x/y \rfloor$，更簡潔。

堆疊表示法在附錄 **A** 中廣泛使用，其中列出了所有目前定義的 TVM 基本操作。

2.1.11. 明確定義函數的引數數量. 堆疊機器通常將目前堆疊完整地傳遞給呼叫的基本操作或函數。該基本操作或函數僅存取堆疊頂部附近表示其引數的幾個值，並按慣例將傳回值推入它們的位置，保持所有更深的值不變。然後，產生的堆疊再次完整地傳回給呼叫者。

大多數 TVM 基本操作都以這種方式運作，我們期望大多數使用者定義的函數都在這種約定下實作。但是，TVM 提供了機制來指定必須傳遞給被呼叫函數的引數數量（參見 **4.1.10**）。當使用這些機制時，指定數量的值將從呼叫者的堆疊移動到被呼叫函數的（通常最初為空）堆疊中，而更深的值保留在呼叫者的堆疊中，被呼叫者無法存取。呼叫者還可以指定它期望從被呼叫函數接收多少個傳回值。

這樣的引數檢查機制可能很有用，例如，對於呼叫作為引數傳遞給它的使用者提供的函數的函式庫函數。

2.2 堆疊操作基本操作

堆疊機器（如 TVM）使用大量堆疊操作基本操作來重新排列其他基本操作和使用者定義函數的引數，使它們以正確的順序位於堆疊頂部附近。本節討論哪些堆疊操作基本操作對於實現此目標是必要和充分的，以及 TVM 使用哪些基本操作。使用這些基本操作的程式碼示例可以在附錄 **C** 中找到。

2.2.1. 基本堆疊操作基本操作. TVM 使用的最重要的堆疊操作基本操作如下：

- 堆疊頂部交換操作: `XCHG s0,s(`i`)` 或 `XCHG s(`i`)` — 交換 `s0` 和 `s(`i`)` 的值。當 $i = 1$ 時，操作 `XCHG s1` 傳統上表示為 `SWAP`。當 $i = 0$ 時，這是 `NOP`（什麼都不做的操作，至少如果堆疊非空）。
- 任意交換操作: `XCHG s(`i`),s(`j`)` — 交換 `s(`i`)` 和 `s(`j`)` 的值。請注意，此操作並非嚴格必要，因為它可以透過三個堆疊頂部交換來模擬：`XCHG s(`i`); XCHG s(`j`); XCHG s(`i`)`。但是，將任意交換作為基本操作很有用，因為它們經常需要。
- 推入操作: `PUSH s(`i`)` — 將 `s(`i`)` 的（舊）值的副本推入堆疊。傳統上，`PUSH s0` 也表示為 `DUP`（它複製堆疊頂部的值），`PUSH s1` 表示為 `OVER`。
- 彈出操作: `POP` $s(i)$ — 移除堆疊頂部值並將其放入（新的）`s(`$i-1$`)`，或舊的 `s(`i`)`。傳統上，`POP s0` 也表示為 `DROP`（它只是丟棄堆疊頂部值），`POP s1` 表示為 `NIP`。

還可能定義一些其他「非系統化」的堆疊操作（例如，`ROT`，堆疊表示法為 $a\ b\ c - b\ c\ a$）。雖然這樣的操作在 Forth 等堆疊語言中定義（其中也存在 `DUP`、`DROP`、`OVER`、`NIP` 和 `SWAP`），但它們並非嚴格必要，因為上面列出的基本堆疊操作基本操作足以重新排列堆疊暫存器，以允許正確呼叫任何算術基本操作和使用者定義的函數。

2.2.2. 基本堆疊操作基本操作足夠. 編譯器或人類 TVM 程式碼程式設計師可以如下使用基本堆疊基本操作。

假設要呼叫的函數或基本操作要傳遞，比如說，三個引數 x、y 和 z，目前位於堆疊暫存器 `s(`i`)`、`s(`j`)` 和 `s(`k`)` 中。在這種情況下，編譯器（或程式設計師）可能會發出操作 `PUSH s(`i`)`（如果在呼叫此基本操作後需要 x 的副本）或 `XCHG s(`i`)`（如果之後不需要）將第一個引數 x 放入堆疊頂部。

然後，編譯器（或程式設計師）可以使用 PUSH s(j') 或 XCHG s(j')，其中 $j' = j$ 或 $j + 1$，將 y 放入新的堆疊頂部。[8]

以這種方式進行，我們可以看到我們可以將 x、y 和 z 的原始值——或它們的副本（如果需要）——放入位置 s2、s1 和 s0 中，使用一系列推入和交換操作（參見 **2.2.4** 和 **2.2.5** 以獲得更詳細的解釋）。為了產生此序列，編譯器只需要知道三個值 i、j 和 k，描述所討論變數或暫存值的舊位置，以及一些標誌，描述之後是否需要每個值或僅對於此基本操作或函數呼叫需要。其他變數和暫存值的位置將在過程中受到影響，但編譯器（或人類程式設計師）可以輕鬆追蹤它們的新位置。

類似地，如果需要丟棄或移動從函數傳回的結果到其他堆疊暫存器，合適的交換和彈出操作序列將完成工作。在 s0 中有一個傳回值的典型情況下，這透過 XCHG s(i) 或 POP s(i)（在大多數情況下是 DROP）操作實現。[9]

在從函數傳回之前重新排列結果值本質上與為函數呼叫排列引數是相同的問題，並以類似方式實現。

2.2.3. 複合堆疊操作基本操作. 為了提高 TVM 程式碼的密度並簡化編譯器的開發，可以定義複合堆疊操作基本操作，每個組合最多四個交換和推入或交換和彈出基本操作。此類複合堆疊操作可能包括，例如：

- XCHG2 s(i),s(j) — 等效於 XCHG s1,s(i); XCHG s(j)。
- PUSH2 s(i),s(j) — 等效於 PUSH s(i); PUSH s($j + 1$)。
- XCPU s(i),s(j) — 等效於 XCHG s(i); PUSH s(j)。
- PUXC s(i),s(j) — 等效於 PUSH s(i); SWAP; XCHG s($j + 1$)。當 $j \neq i$ 且 $j \neq 0$ 時，它也等效於 XCHG s(j); PUSH s(i); SWAP。
- XCHG3 s(i),s(j),s(k) — 等效於 XCHG s2,s(i); XCHG s1,s(j); XCHG s(k)。
- PUSH3 s(i),s(j),s(k) — 等效於 PUSH s(i); PUSH s($j+1$); PUSH s($k+2$)。

[8]當然，如果使用第二個選項，這將破壞 x 在堆疊頂部的原始排列。在這種情況下，應該在 XCHG s(j') 之前發出 SWAP，或將前一個操作 XCHG s(i) 替換為 XCHG s1, s(i)，以便從一開始就將 x 與 s1 交換。

[9]請注意，如果我們不堅持將相同的暫存值或變數始終保持在相同的堆疊位置，而是追蹤其後續位置，那麼最常見的 XCHG $s(i)$ 操作在這裡並不是真正需要的。我們將在準備下一個基本操作或函數呼叫的引數時將其移動到其他位置。

當然，這些操作只有在它們允許比等效基本操作序列更緊湊的編碼時才有意義。例如，如果所有堆疊頂部交換、XCHG s1,s(i) 交換以及推入和彈出操作都允許單位元組編碼，則上述建議的複合堆疊操作中唯一可能值得包含在堆疊操作基本操作集中的是 PUXC、XCHG3 和 PUSH3。

這些複合堆疊操作本質上為程式碼中的其他基本操作（指令）增加了其運算元的「真實」位置，有點類似於兩位址或三位址暫存器機器程式碼發生的情況。但是，我們不像暫存器機器那樣將這些位置編碼在算術或其他指令的操作碼內部，而是在前面的複合堆疊操作中指示這些位置。如 **2.1.7** 中已經描述的，這種方法的優點是使用者定義的函數（或在 TVM 的未來版本中添加的很少使用的特定基本操作）也可以從中受益（參見 **C.3** 以獲得帶有示例的更詳細討論）。

2.2.4. 複合堆疊操作的助記符. 複合堆疊操作的助記符（其中一些示例已在 **2.2.3** 中提供）按如下方式建立。

這樣一個操作 O 的 $\gamma \geq 2$ 個形式引數 s(i_1)、…、s(i_γ) 表示原始堆疊中的值，這些值將在執行此複合操作後最終位於 s($\gamma-1$)、…、s0 中，至少如果所有 i_ν（$1 \leq \nu \leq \gamma$）都是不同的且至少為 γ。操作 O 本身的助記符是 γ 個兩字母字串 PU 和 XC 的序列，其中 PU 表示相應的引數將被推入（即，要建立副本），XC 表示值將被交換（即，不建立原始值的其他副本）。幾個 PU 或 XC 字串的序列可以縮寫為一個 PU 或 XC，後跟副本數量。（例如，我們寫 PUXC2PU 而不是 PUXCXCPU。）

作為例外，如果助記符僅由 PU 或僅由 XC 字串組成，因此複合操作等效於 m 個推入或交換序列，則使用表示法 PUSHm 或 XCHGm 代替 PUm 或 XCm。

2.2.5. 複合堆疊操作的語意. 每個複合 γ 元操作 O s(i_1),…,s(i_γ) 透過對 γ 的歸納法翻譯為等效的基本堆疊操作序列，如下：

- 作為歸納的基礎，如果 $\gamma = 0$，唯一的零元複合堆疊操作對應於基本堆疊操作的空序列。
- 或者，我們可以從 $\gamma = 1$ 開始歸納。那麼 PU s(i) 對應於由一個基本操作 PUSH s(i) 組成的序列，XC s(i) 對應於由 XCHG s(i) 組成的單元素序列。
- 對於 $\gamma \geq 1$（或對於 $\gamma \geq 2$，如果我們使用 $\gamma = 1$ 作為歸納基礎），有兩個子情況：
    1. Os(i_1),…, s(i_γ)，其中 O = XCO'，其中 O' 是元數為 $\gamma - 1$ 的複合操作（即，O' 的助記符由 $\gamma - 1$ 個字串 XC 和 PU 組成）。

令 α 為 O 中 PUsh 的總數量，β 為 eXChange 的總數量，使得 $\alpha + \beta = \gamma$。那麼原始操作翻譯為 XCHG $\mathtt{s}(\beta - 1)$,$\mathtt{s}(i_1)$，後跟 $O'\mathtt{s}(i_2), \ldots, \mathtt{s}(i_\gamma)$ 的翻譯，由歸納假設定義。

2. $O\mathtt{s}(i_1), \ldots, \mathtt{s}(i_\gamma)$，其中 $O = \mathtt{PU}O'$，其中 O' 是元數為 $\gamma - 1$ 的複合操作。那麼原始操作翻譯為 PUSH $\mathtt{s}(i_1)$; XCHG $\mathtt{s}(\beta)$，後跟 $O'\mathtt{s}(i_2 + 1), \ldots, \mathtt{s}(i_\gamma + 1)$ 的翻譯，由歸納假設定義。[10]

2.2.6. 堆疊操作指令是多型的. 請注意，堆疊操作指令幾乎是 TVM 中唯一的「多型」基本操作——即，它們適用於任意型別的值（包括僅在 TVM 的未來修訂版中才會出現的值型別）。例如，SWAP 始終交換堆疊的兩個頂部值，即使其中一個是整數而另一個是單元。幾乎所有其他指令，尤其是資料處理指令（包括算術指令），都要求其每個引數都是某種固定型別（對於不同引數可能不同）。

2.3 堆疊操作基本操作的效率

堆疊機器（如 TVM）使用的堆疊操作基本操作必須非常有效地實作，因為它們構成典型程式中使用的所有指令的一半以上。實際上，TVM 在（小）常數時間內執行所有這些指令，無論涉及的值如何（即使它們表示非常大的整數或非常大的單元樹）。

2.3.1. 堆疊操作基本操作的實作：對操作使用參照而不是物件. TVM 堆疊操作基本操作實作效率的原因在於，典型的 TVM 實作在堆疊中保存的不是值物件本身，而只是對這些物件的參照（指標）。因此，SWAP 指令只需要交換 s0 和 s1 處的參照，而不是它們參照的實際物件。

2.3.2. 使用寫時複製有效實作 DUP 和 PUSH 指令. 此外，DUP（或更一般地，PUSH $\mathtt{s}(i)$）指令，看起來要複製可能很大的物件，也在小常數時間內工作，因為它使用延遲複製的寫時複製技術：它只複製參照而不是物件本身，但增加物件內部的「參照計數器」，從而在兩個參照之間共享物件。如果檢測到嘗試修改參照計數器大於一的物件，則首先製作所討論物件的單獨副本（對於觸發建立新副本的資料操作指令產生一定的「非唯一性懲罰」或「複製懲罰」）。

2.3.3. 垃圾收集和參照計數. 當 TVM 物件的參照計數器變為零時（例如，因為對這樣的物件的最後一個參照已被 DROP 操作或算術指令消耗），它會立即被釋放。因為 TVM 資料結構中不可能存在迴圈參照，所以這種參照

[10] $\mathtt{PU}O'\mathtt{s}(i_1), \ldots, \mathtt{s}(i_\gamma)$ 的另一種可以說是更好的翻譯由 $O'\mathtt{s}(i_2), \ldots, \mathtt{s}(i_\gamma)$ 的翻譯組成，後跟 PUSH $\mathtt{s}(i_1 + \alpha - 1)$; XCHG $\mathtt{s}(\gamma - 1)$。

計數方法提供了一種快速方便的釋放未使用物件的方法，取代了緩慢且不可預測的垃圾收集器。

2.3.4. 實作的透明性：堆疊值是「值」，而不是「參照」. 無論剛剛討論的實作細節如何，從 TVM 程式設計師的角度來看，所有堆疊值都真正是「值」，而不是「參照」，類似於函數編程語言中所有型別的值。任何從任何其他物件或堆疊位置參照的現有物件的修改嘗試都將導致在實際執行修改之前透明地將此物件替換為其完美副本。

換句話說，程式設計師應該始終表現得好像物件本身由堆疊、算術和其他資料轉換基本操作直接操作，並且僅將前面的討論視為堆疊操作基本操作高效率的解釋。

2.3.5. 迴圈參照的缺失. 可能會嘗試在兩個單元 A 和 B 之間建立迴圈參照，如下：首先建立 A 並將一些資料寫入其中；然後建立 B 並將一些資料寫入其中，以及對先前建構的單元 A 的參照；最後，將對 B 的參照添加到 A 中。雖然在這一系列操作之後似乎我們獲得了一個單元 A，它參照 B，而 B 又參照 A，但情況並非如此。實際上，我們獲得了一個新單元 A'，它包含最初儲存到單元 A 中的資料的副本以及對單元 B 的參照，單元 B 包含對（原始）單元 A 的參照。

這樣，透明的寫時複製機制和「一切都是值」範式使我們能夠僅使用先前建構的單元來建立新單元，從而禁止迴圈參照的出現。此屬性也適用於所有其他資料結構：例如，迴圈參照的缺失使 TVM 能夠使用參照計數立即釋放未使用的記憶體，而不是依賴垃圾收集器。類似地，此屬性對於在 TON 區塊鏈中儲存資料至關重要。

3 單元、記憶體和持久儲存

本章簡要描述 TVM 單元，用於表示 TVM 記憶體及其持久儲存中的所有資料結構，以及用於建立單元、將資料寫入（或序列化）它們以及從它們讀取（或反序列化）資料的基本操作。

3.1 單元的一般性

本節介紹單元型別的分類和一般描述。

3.1.1. TVM 記憶體和持久儲存由單元組成. 回想一下，TVM 記憶體和持久儲存由（*TVM*）單元組成。每個單元包含最多 1023 位元的資料和最多四個對其他單元的參照。[11] 禁止迴圈參照，並且無法透過 TVM 的方式建立（參見 **2.3.5**）。這樣，保存在 TVM 記憶體和持久儲存中的所有單元構成有向無環圖（DAG）。

3.1.2. 普通和特殊單元. 除了資料和參照之外，單元還有一個單元型別，由整數 $-1…255$ 編碼。型別為 -1 的單元稱為普通；這樣的單元不需要任何特殊處理。其他型別的單元稱為特殊，並且可能被載入——當嘗試反序列化它們（即，透過 `CTOS` 指令將它們轉換為 *Slice*）時自動替換為其他單元。在計算它們的雜湊時，它們也可能表現出非平凡的行為。

特殊單元最常見的用途是表示其他單元——例如，存在於外部函式庫中的單元，或在建立 Merkle 證明時從原始單元樹中修剪的單元。

特殊單元的型別儲存為其資料的前八位元。如果特殊單元的資料位元少於八個，則它是無效的。

3.1.3. 單元的層級. 每個單元 c 還有另一個屬性 $\textsc{Lvl}(c)$，稱為其（*de Brujn*）層級，目前在範圍 $0…3$ 內取整數值。普通單元的層級始終等於其所有子單元 c_i 的層級的最大值：

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

對於包含對單元 c_1、…、c_r 的 r 個參照的普通單元 c。如果 $r = 0$，則 $\textsc{Lvl}(c) = 0$。特殊單元可能有不同的設定層級規則。

單元的層級影響它擁有的更高雜湊數量。更準確地說，層級為 l 的單元除了其表示雜湊 $\textsc{Hash}(c) = \textsc{Hash}_\infty(c)$ 之外，還有 l 個更高雜湊 $\textsc{Hash}_1(c)$、

[11] 從低階單元操作的角度來看，這些資料位元和單元參照不是混合的。換句話說，（普通）單元本質上是由最多 1023 位元的列表和最多四個單元參照的列表組成的對，而不規定應該以何種順序反序列化參照和資料位元，即使 TL-B 方案似乎暗示了這樣的順序。

…、$\text{Hash}_l(c)$。非零層級的單元出現在 *Merkle* 證明和 *Merkle* 更新內部，在表示抽象資料型別值的單元樹的某些分支被修剪之後。

3.1.4. 標準單元表示. 當單元需要透過網路協議傳輸或儲存在磁碟檔案中時，它必須被序列化。單元 c 作為八位元組（位元組）序列的標準表示 $\text{CellRepr}(c) = \text{CellRepr}_\infty(c)$ 按如下方式建構：

1. 首先序列化兩個描述符位元組 d_1 和 d_2。位元組 d_1 等於 $r + 8s + 32l$，其中 $0 \le r \le 4$ 是單元中包含的單元參照數量，$0 \le l \le 3$ 是單元的層級，$0 \le s \le 1$ 對於特殊單元為 1，對於普通單元為 0。位元組 d_2 等於 $\lfloor b/8 \rfloor + \lceil b/8 \rceil$，其中 $0 \le b \le 1023$ 是 c 中的資料位元數量。

2. 然後將資料位元序列化為 $\lceil b/8 \rceil$ 個 8 位元八位元組（位元組）。如果 b 不是八的倍數，則在資料位元後附加一個二進位 `1` 和最多六個二進位 `0`。之後，資料被分成 $\lceil b/8 \rceil$ 個八位元組，每組被解釋為無號大端整數 $0 \ldots 255$ 並儲存到八位元組中。

3. 最後，r 個單元參照中的每一個都由包含 256 位元表示雜湊 $\text{Hash}(c_i)$ 的 32 個位元組表示，在下面的 **3.1.5** 中解釋，它是被參照單元 c_i 的表示雜湊。

這樣，獲得 $\text{CellRepr}(c)$ 的 $2 + \lceil b/8 \rceil + 32r$ 個位元組。

3.1.5. 單元的表示雜湊. 單元 c 的 256 位元表示雜湊或簡稱雜湊 $\text{Hash}(c)$ 遞迴定義為單元 c 標準表示的 sha256：

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

請注意，不允許迴圈單元參照，並且無法透過 TVM 的方式建立（參見 **2.3.5**），因此此遞迴始終結束，並且任何單元的表示雜湊都是良好定義的。

3.1.6. 單元的更高雜湊. 回想一下，層級為 l 的單元 c 也有 l 個更高雜湊 $\text{Hash}_i(c)$，$1 \le i \le l$。特殊單元有自己的計算更高雜湊的規則。普通單元 c 的更高雜湊 $\text{Hash}_i(c)$ 的計算方式與其表示雜湊類似，但使用其子單元 c_j 的更高雜湊 $\text{Hash}_i(c_j)$ 而不是它們的表示雜湊 $\text{Hash}(c_j)$。按慣例，我們設定 $\text{Hash}_\infty(c) := \text{Hash}(c)$，並且對於所有 $i > l$，$\text{Hash}_i(c) := \text{Hash}_\infty(c) = \text{Hash}(c)$。[12]

[12]從理論角度來看，我們可能會說單元 c 有一個無限的雜湊序列 $\big(\text{Hash}_i(c)\big)_{i \ge 1}$，它最終穩定：$\text{Hash}_i(c) \to \text{Hash}_\infty(c)$。那麼層級 l 只是最大的索引 i，使得 $\text{Hash}_i(c) \ne \text{Hash}_\infty(c)$。

3.1.7. 特殊單元的型別. TVM 目前支援以下單元型別：

- 型別 -1：普通單元 — 包含最多 1023 位元的資料和最多四個單元參照。

- 型別 1：修剪分支單元 c — 可能有任何層級 $1 \leq l \leq 3$。它精確包含 $8 + 256l$ 個資料位元：首先是等於 1 的 8 位元整數（表示單元的型別），然後是其 l 個更高雜湊 $\text{HASH}_1(c)$、…、$\text{HASH}_l(c)$。修剪分支單元的層級 l 可能稱為其 *de Brujn* 索引，因為它確定在建構期間修剪分支的外部 Merkle 證明或 Merkle 更新。嘗試載入修剪分支單元通常會導致異常。

- 型別 2：函式庫參照單元 — 始終具有層級 0，並包含 $8 + 256$ 個資料位元，包括其 8 位元型別整數 2 和被參照的函式庫單元的表示雜湊 $\text{HASH}(c')$。載入時，如果在目前函式庫上下文中找到，函式庫參照單元可能被其參照的單元透明替換。

- 型別 3：*Merkle* 證明單元 c — 精確有一個參照 c_1 和層級 $0 \leq l \leq 3$，它必須比其唯一子單元 c_1 的層級少一：

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

  Merkle 證明單元的 $8 + 256$ 個資料位元包含其 8 位元型別整數 3，後跟 $\text{HASH}_1(c_1)$（如果 $\text{LVL}(c_1) = 0$，則假設等於 $\text{HASH}(c_1)$）。c 的更高雜湊 $\text{HASH}_i(c)$ 的計算方式與普通單元的更高雜湊類似，但使用 $\text{HASH}_{i+1}(c_1)$ 代替 $\text{HASH}_i(c_1)$。載入時，Merkle 證明單元被 c_1 替換。

- 型別 4：*Merkle* 更新單元 c — 有兩個子單元 c_1 和 c_2。其層級 $0 \leq l \leq 3$ 由以下給出：

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

  Merkle 更新對 c_1 和 c_2 都表現得像 Merkle 證明，並包含 $8+256+256$ 個資料位元，其中包含 $\text{HASH}_1(c_1)$ 和 $\text{HASH}_1(c_2)$。但是，一個額外的要求是所有是 c_2 的後代並由 c 綁定的修剪分支單元 c' 也必須是 c_1 的後代。[13] 載入 Merkle 更新單元時，它被 c_2 替換。

[13]如果從 c 到其後代 c' 的路徑上恰好有 l 個 Merkle 單元，包括 c，則層級為 l 的修剪分支單元 c' 被 Merkle（證明或更新）單元 c 綁定。

3.1.8. 代數資料型別的所有值都是單元樹. 任意代數資料型別的任意值（例如，函數編程語言中使用的所有型別）都可以序列化為（層級 0 的）單元樹，並且這樣的表示用於在 TVM 內表示這樣的值。寫時複製機制（參見 **2.3.2**）允許 TVM 識別包含相同資料和參照的單元，並僅保留這樣單元的一個副本。這實際上將單元樹轉換為有向無環圖（具有所有頂點都可從稱為「根」的標記頂點存取的附加屬性）。但是，這是儲存優化而不是 TVM 的基本屬性。從 TVM 程式碼程式設計師的角度來看，應該將 TVM 資料結構視為單元樹。

3.1.9. TVM 程式碼是單元樹. TVM 程式碼本身也由單元樹表示。實際上，TVM 程式碼只是某個複雜代數資料型別的值，因此它可以序列化為單元樹。

TVM 程式碼（例如，TVM 組合語言程式碼）轉換為單元樹的確切方式將在稍後解釋（參見 **4.1.4** 和 **5.2**），在討論控制流指令、繼續和 TVM 指令編碼的章節中。

3.1.10. 「一切都是單元集合」範式. 如 [1, 2.5.14] 中所述，TON 區塊鏈使用的所有資料，包括區塊本身和區塊鏈狀態，都可以表示——並且被表示——為單元的集合或「袋子」。我們看到 TVM 的資料（參見 **3.1.8**）和程式碼（參見 **3.1.9**）結構很好地符合這個「一切都是單元集合」範式。這樣，TVM 自然可以用於在 TON 區塊鏈中執行智慧合約，並且 TON 區塊鏈可以用於在 TVM 呼叫之間儲存這些智慧合約的程式碼和持久資料。（當然，TVM 和 TON 區塊鏈都已經被設計成使這成為可能。）

3.2 資料操作指令和單元

TVM 指令的下一個大組由資料操作指令組成，也稱為單元操作指令或簡稱單元指令。它們對應於其他架構的記憶體存取指令。

3.2.1. 單元操作指令的類別. TVM 單元指令自然分為兩個主要類別：

- 單元建立指令或序列化指令，用於從先前保存在堆疊中的值和先前建構的單元建構新單元。
- 單元解析指令或反序列化指令，用於提取先前由單元建立指令儲存到單元中的資料。

此外，還有特殊單元指令用於建立和檢查特殊單元（參見 **3.1.2**），特別用於表示 Merkle 證明的修剪分支和 Merkle 證明本身。

3.2.2. *Builder* 和 *Slice* 值. 單元建立指令通常使用 *Builder* 值，這些值只能保存在堆疊中（參見 **1.1.3**）。這樣的值表示部分建構的單元，對於它們可以定義用於在其末尾附加位元字串、整數、其他單元和對其他單元的參照的快速操作。類似地，單元解析指令大量使用 *Slice* 值，這些值表示部分解析的單元的餘數，或駐留在這樣的單元內部並由解析指令從中提取的值（子單元）。

3.2.3. *Builder* 和 *Slice* 值僅作為堆疊值存在. 請注意，*Builder* 和 *Slice* 物件僅作為 TVM 堆疊中的值出現。它們不能儲存在「記憶體」（即，單元樹）或「持久儲存」（也是單元集合）中。從這個意義上說，在 TVM 環境中，*Cell* 物件遠多於 *Builder* 或 *Slice* 物件，但有點矛盾的是，TVM 程式在其堆疊中看到 *Builder* 和 *Slice* 物件的頻率比 *Cell* 更高。實際上，TVM 程式對 *Cell* 值的用處不大，因為它們是不可變和不透明的；所有單元操作基本操作都要求首先將 *Cell* 值轉換為 *Builder* 或 *Slice*，然後才能修改或檢查它。

3.2.4. TVM 沒有單獨的 *Bitstring* 值型別. 請注意，TVM 不提供單獨的位元字串值型別。相反，位元字串由碰巧沒有參照但仍可以包含最多 1023 個資料位元的 *Slice* 表示。

3.2.5. 單元和單元基本操作是位元導向的，而不是位元組導向的. 一個重要的點是 *TVM* 將保存在單元中的資料視為（最多 *1023* 個）位元的序列（字串、流），而不是位元組。換句話說，TVM 是位元導向機器，而不是位元組導向機器。如果需要，應用程式可以自由使用，例如，序列化到 TVM 單元中的記錄內部的 21 位元整數欄位，從而使用更少的持久儲存位元組來表示相同的資料。

3.2.6. 單元建立（序列化）基本操作的分類. 單元建立基本操作通常接受 *Builder* 引數和表示要序列化的值的引數。還可以提供控制序列化過程某些方面的額外引數（例如，應該使用多少位元進行序列化），無論是在堆疊中還是作為指令內部的立即值。單元建立基本操作的結果通常是另一個 *Builder*，表示原始建構器和提供的值的序列化的串接。

因此，可以根據以下問題的答案建議單元序列化基本操作的分類：

- 被序列化的值的型別是什麼？
- 用於序列化的位元數是多少？如果這是可變數字，它來自堆疊還是來自指令本身？
- 如果值不適合規定的位元數會發生什麼？是產生異常，還是在堆疊頂部靜默傳回等於零的成功標誌？

- 如果 *Builder* 中剩餘的空間不足會發生什麼？是產生異常，還是傳回零成功標誌以及未修改的原始 *Builder*？

單元序列化基本操作的助記符通常以 `ST` 開頭。後續字母描述以下屬性：

- 被序列化的值的型別和序列化格式（例如，`I` 表示有號整數，`U` 表示無號整數）。
- 要使用的位元欄位寬度的來源（例如，對於整數序列化指令，`X` 表示位元寬度 n 在堆疊中提供；否則它必須作為立即值嵌入到指令中）。
- 如果操作無法完成要執行的動作（預設情況下產生異常；序列化指令的「靜默」版本在其助記符中用 `Q` 字母標記）。

此分類方案用於建立單元序列化基本操作的更完整分類，可以在 **A.7.1** 中找到。

3.2.7. 整數序列化基本操作. 整數序列化基本操作也可以根據上述分類進行分類。例如：

- 有有號和無號（大端）整數序列化基本操作。
- 要使用的位元欄位的大小 n（對於有號整數為 $1 \leq n \leq 257$，對於無號整數為 $0 \leq n \leq 256$）可以來自堆疊頂部或嵌入到指令本身中。
- 如果要序列化的整數 x 不在範圍 $-2^{n-1} \leq x < 2^{n-1}$（對於有號整數序列化）或 $0 \leq x < 2^n$（對於無號整數序列化）內，通常會產生範圍檢查異常，如果 n 位元無法儲存到提供的 *Builder* 中，則會產生單元溢位異常。
- 序列化指令的靜默版本不會引發異常；相反，它們在成功時在產生的 *Builder* 頂部推入 `-1`，或傳回原始 *Builder* 並在其頂部放置 `0` 以指示失敗。

整數序列化指令的助記符如 `STU 20`（「儲存無號 20 位元整數值」）或 `STIXQ`（「靜默儲存在堆疊中提供的可變長度整數值」）。這些指令的完整列表——包括它們的助記符、描述和操作碼——在 **A.7.1** 中提供。

3.2.8. 單元中的整數預設為大端. 請注意，序列化到 *Cell* 中的 *Integer* 中的位元的預設順序是大端，而不是小端。[14] 在這方面 *TVM* 是大端機器。

[14] 負數使用二的補數表示。例如，整數 -17 由指令 `STI 8` 序列化為位元字串 `xEF`。

但是，這僅影響單元內部整數的序列化。*Integer* 值型別的內部表示取決於實作，與 TVM 的操作無關。此外，還有一些特殊的基本操作，如 STULE，用於（反）序列化小端整數，這些整數必須儲存在整數個位元組中（否則「小端」沒有意義，除非也願意反轉八位元組內部的位元順序）。這樣的基本操作對於與小端世界的介面很有用——例如，用於解析從外部世界到達 TON 區塊鏈智慧合約的自訂格式訊息。

3.2.9. 其他序列化基本操作. 其他單元建立基本操作序列化位元字串（即，沒有參照的單元切片），無論是從堆疊中取出還是作為字面引數提供；單元切片（以明顯的方式串接到單元建構器）；其他 *Builder*（也被串接）；以及單元參照（STREF）。

3.2.10. 其他單元建立基本操作. 除了上述某些內建值型別的單元序列化基本操作之外，還有一些簡單的基本操作，它們建立一個新的空 *Builder* 並將其推入堆疊（NEWC），或將 *Builder* 轉換為 *Cell*（ENDC），從而完成單元建立過程。ENDC 可以與 STREF 組合成單個指令 ENDCST，它完成單元的建立並立即在「外部」*Builder* 中儲存對它的參照。還有一些基本操作可以獲得已儲存在 *Builder* 中的資料位元或參照的數量，並檢查可以儲存多少資料位元或參照。

3.2.11. 單元反序列化基本操作的分類. 單元解析或反序列化基本操作可以如 **3.2.6** 中所述進行分類，並進行以下修改：

- 它們使用 *Slice*（表示正在解析的單元的餘數）而不是 *Builder*。
- 它們傳回反序列化的值而不是接受它們作為引數。
- 它們可能有兩種風格，取決於它們是否從提供的 *Slice* 中移除反序列化的部分（「提取操作」）或保持它不變（「預提取操作」）。
- 它們的助記符通常以 LD（或 PLD 用於預提取操作）開頭，而不是 ST。

例如，先前由 STU 20 指令序列化到單元中的無號大端 20 位元整數可能稍後由匹配的 LDU 20 指令反序列化。

同樣，關於這些指令的更詳細資訊在 **A.7.2** 中提供。

3.2.12. 其他單元切片基本操作. 除了上述單元反序列化基本操作之外，TVM 還提供了一些明顯的基本操作，用於初始化和完成單元反序列化過程。例如，可以將 *Cell* 轉換為 *Slice*（CTOS），以便可以開始其反序列化；或檢查 *Slice* 是否為空，如果不是則產生異常（ENDS）；或反序列化單元參照並立即將其轉換為 *Slice*（LDREFTOS，等效於兩個指令 LDREF 和 CTOS）。

3.2.13. 修改單元中的序列化值. 讀者可能想知道如何修改序列化在單元內部的值。假設一個單元包含三個序列化的 29 位元整數 (x, y, z)，表示空間中點的座標，我們想用 $y' = y + 1$ 替換 y，保持其他座標不變。我們將如何實現這一點？

TVM 不提供任何修改現有值的方法（參見 **2.3.4** 和 **2.3.5**），因此我們的示例只能透過一系列操作來完成，如下：

1. 將原始單元反序列化為堆疊中的三個 *Integer* x、y、z（例如，透過 `CTOS; LDI 29; LDI 29; LDI 29; ENDS`）。
2. 將 y 增加一（例如，透過 `SWAP; INC; SWAP`）。
3. 最後，將產生的 *Integer* 序列化到新單元中（例如，透過 `XCHG s2; NEWC; STI 29; STI 29; STI 29; ENDC`）。

3.2.14. 修改智慧合約的持久儲存. 如果 TVM 程式碼想要修改其持久儲存，由根在 `c4` 的單元樹表示，它只需要用包含其持久儲存新值的單元樹的根重寫控制暫存器 `c4`。（如果只需要修改持久儲存的一部分，參見 **3.2.13**。）

3.3 雜湊映射或字典

雜湊映射或字典是由單元樹表示的特定資料結構。本質上，雜湊映射表示從鍵（固定或可變長度的位元字串）到任意型別 X 的值的映射，以便可以進行快速查找和修改。雖然任何這樣的結構都可以在通用單元序列化和反序列化基本操作的幫助下進行檢查或修改，但 TVM 引入了特殊的基本操作來促進使用這些雜湊映射。

3.3.1. 基本雜湊映射型別. TVM 中預定義的兩種最基本的雜湊映射型別是 *HashmapE* n X 或 $HashmapE(n, X)$，它表示從 n 位元字串（稱為鍵）到某種型別 X 的值的部分定義映射，對於某個固定的 $0 \leq n \leq 1023$，以及 $Hashmap(n, X)$，它類似於 $HashmapE(n, X)$ 但不允許為空（即，它必須包含至少一個鍵值對）。

還可以使用其他雜湊映射型別——例如，具有任意長度鍵（最多某個預定義界限，最多 1023 位元）的雜湊映射型別。

3.3.2. 雜湊映射作為 Patricia 樹. TVM 中雜湊映射的抽象表示是 *Patricia* 樹，或緊湊二元字典樹。它是一個二元樹，邊用位元字串標記，使得從根到葉的路徑上所有邊標籤的串接等於雜湊映射的鍵。對應的值保存在此葉中（對於具有固定長度鍵的雜湊映射），或可選地也保存在中間頂點中（對

於具有可變長度鍵的雜湊映射）。此外，任何中間頂點必須有兩個子節點，並且左子節點的標籤必須以二進位零開頭，而右子節點的標籤必須以二進位一開頭。這使我們不必明確儲存邊標籤的第一個位元。

容易看出，任何鍵值對集合（具有不同的鍵）都由唯一的 Patricia 樹表示。

3.3.3. 雜湊映射的序列化. 雜湊映射序列化為單元樹（或更一般地，序列化為 *Slice*）由以下 TL-B 方案定義：[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;

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

3.3.4. TL-B 方案的簡要解釋. 像上面這樣的 TL-B 方案包括以下元件。

每個「等式」的右側是一個型別，可以是簡單的（如 `Bit` 或 `True`）或參數化的（如 `Hashmap` n X）。型別的參數必須是自然數（即非負整數，實際上需要適合 32 位元），如 `Hashmap` n X 中的 n，或其他型別，如 `Hashmap` n X 中的 X。

[15]較舊版本的 TL 描述可以在 `https://core.telegram.org/mtproto/TL` 找到。

每個等式的左側描述了定義甚至序列化右側指示的型別的值的方法。這樣的描述以建構器的名稱開始，例如 `hm_edge` 或 `hml_long`，緊接著是可選的建構器標記，例如 `#_` 或 `$10`，它描述用於編碼（序列化）所討論建構器的位元字串。這樣的標記可以用二進位（在美元符號之後）或十六進位表示法（在井號之後）給出，使用 **1.0** 中描述的約定。如果未明確提供標記，TL-B 會透過以某種方式對定義此建構器的「等式」文字進行雜湊來計算預設的 32 位元建構器標記。因此，空標記必須由 `#_` 或 `$_` 明確提供。所有建構器名稱必須是不同的，並且相同型別的建構器標記必須構成前綴碼（否則反序列化將不是唯一的）。

建構器及其可選標記後面是欄位定義。每個欄位定義的形式為 *ident* : *type-expr*，其中 *ident* 是帶有欄位名稱的識別字[16]（對於匿名欄位替換為底線），*type-expr* 是欄位的型別。這裡提供的型別是型別表示式，它可以包括簡單型別或具有適當參數的參數化型別。變數——即型別 $\#$（自然數）或 *Type*（型別的型別）的先前定義欄位的（識別字）——可以用作參數化型別的參數。序列化過程根據其型別遞迴序列化每個欄位，值的序列化最終由表示建構器（即建構器標記）和欄位值的位元字串串接組成。

某些欄位可能是隱式的。它們的定義被大括號包圍，這表示欄位實際上不存在於序列化中，但其值必須從其他資料（通常是正在序列化的型別的參數）中推導出來。

「變數」（即已定義的欄位）的某些出現前面有波浪號。這表示變數的出現以與預設行為相反的方式使用：在等式的左側，它表示變數將根據此出現推導（計算），而不是替換其先前計算的值；相反，在右側，它表示變數不會從正在序列化的型別推導，而是在反序列化過程中計算。換句話說，波浪號將「輸入引數」轉換為「輸出引數」，反之亦然。[17]

最後，一些等式也可以包含在大括號中。這些是某些「等式」，必須由包含在其中的「變數」滿足。如果其中一個變數前面有波浪號，則當從左到右處理定義時，其值將由參與等式的所有其他變數的值（此時必須已知）唯一確定。

型別 X 前面的插入符號（`^`）表示，我們不是將型別 X 的值作為位元字串序列化到目前單元內部，而是將此值放入單獨的單元中，並在目前單元中添加對它的參照。因此 $\hat{}X$ 表示「對包含型別 X 的值的單元的參照型別」。

參數化型別 `#<=` p，其中 p : `#`（此表示法表示「p 的型別 `#`」，即自然數）表示自然數型別 $\#$ 的子型別，由整數 $0 \ldots p$ 組成；它被序列化為

[16]欄位的名稱對於以人類可讀形式表示正在定義的型別的值很有用，但它不影響二進位序列化。

[17]這是線性邏輯的「線性否定」操作 $(-)^{\perp}$，因此我們的表示法~。

$\lceil \log_2(p+1) \rceil$ 位元，作為無號大端整數。型別 `#` 本身被序列化為無號 32 位元整數。參數化型別 `##` b，其中 b : `#<=`31 等效於 `#<=` $2^b - 1$（即，它是無號 b 位元整數）。

3.3.5. 應用於雜湊映射的序列化. 讓我們解釋將 **3.3.4** 中描述的一般規則應用於 **3.3.3** 中提出的 TL-B 方案的淨結果。

假設我們希望序列化某個整數 $0 \leq n \leq 1023$ 和某種型別 X 的型別 *HashmapE* n X 的值（即，具有 n 位元鍵和型別 X 的值的字典，允許作為 Patricia 樹的抽象表示（參見 **3.3.2**））。

首先，如果我們的字典為空，則將其序列化為單個二進位 `0`，這是無參數建構器 `hme_empty` 的標記。否則，其序列化由二進位 `1`（`hme_root` 的標記）組成，以及對包含型別 *Hashmap* n X 的值（即必然非空的字典）的序列化的單元的參照。

序列化型別 *Hashmap* n X 的值的唯一方法由 `hm_edge` 建構器給出，它指示我們首先序列化導向所考慮的子樹根的邊的標籤 `label`（即，我們的（子）字典中所有鍵的公共前綴）。此標籤的型別為 `HmLabel` $l^{\perp}$ n，這意味著它是長度最多為 n 的位元字串，以這樣的方式序列化，使得標籤的真實長度 l，$0 \leq l \leq n$，從標籤的序列化中變得已知。（這種特殊的序列化方法在 **3.3.6** 中單獨描述。）

標籤後面必須是型別為 *HashmapNode* m X 的 `node` 的序列化，其中 $m = n - l$。它對應於 Patricia 樹的頂點，表示原始字典的非空子字典，具有 m 位元鍵，透過從原始子字典的所有鍵中移除它們的長度為 l 的公共前綴而獲得。

如果 $m = 0$，則型別 `HashmapNode` 0 X 的值由 `hmn_leaf` 建構器給出，它描述 Patricia 樹的葉——或等效地，具有 0 位元鍵的子字典。葉僅由型別 X 的對應 `value` 組成，並相應地序列化。

另一方面，如果 $m > 0$，則型別 `HashmapNode` m X 的值對應於 Patricia 樹中的分叉（即中間節點），並由 `hmn_fork` 建構器給出。其序列化由 `left` 和 `right` 組成，兩個對包含型別 `Hashmap` $m-1$ X 的值的單元的參照，它們對應於所討論中間節點的左子節點和右子節點——或等效地，對應於原始字典的兩個子字典，由鍵以二進位 `0` 或二進位 `1` 開頭的鍵值對組成。因為每個這些子字典中所有鍵的第一個位元是已知和固定的，所以它被移除，並且產生的（必然非空）子字典遞迴序列化為型別 `Hashmap` $m-1$ X 的值。

3.3.6. 標籤的序列化. 有幾種方法可以序列化長度最多為 n 的標籤，如果其精確長度為 $l \leq n$（回想一下，精確長度必須從標籤本身的序列化中推導出來，而上限 n 在標籤序列化或反序列化之前是已知的）。這些方法由

型別 `HmLabel` $l^{\perp}$ n 的三個建構器 `hml_short`、`hml_long` 和 `hml_same` 描述：

- `hml_short` — 描述序列化「短」標籤的方法，長度較小 $l \leq n$。這樣的序列化由二進位 `0`（`hml_short` 的建構器標記）組成，後跟 l 個二進位 `1` 和一個二進位 `0`（長度 l 的一元表示），後跟包含標籤本身的 l 個位元。

- `hml_long` — 描述序列化「長」標籤的方法，任意長度 $l \leq n$。這樣的序列化由二進位 `10`（`hml_long` 的建構器標記）組成，後跟長度 $0 \leq l \leq n$ 的大端二進位表示，用 $\lceil \log_2(n+1) \rceil$ 位元表示，後跟包含標籤本身的 l 個位元。

- `hml_same` — 描述序列化「長」標籤的方法，由相同位元 v 的 l 次重複組成。這樣的序列化由 `11`（`hml_same` 的建構器標記）組成，後跟位元 v，後跟如前所述以 $\lceil \log_2(n+1) \rceil$ 位元儲存的長度 l。

每個標籤總是可以以至少兩種不同的方式序列化，使用 `hml_short` 或 `hml_long` 建構器。通常首選最短的序列化（在平局的情況下——最短中字典序最小的），並且由 TVM 雜湊映射基本操作產生，而其他變體仍被視為有效。

此標籤編碼方案被設計為對具有「隨機」鍵（例如，某些資料的雜湊）的字典以及具有「常規」鍵（例如，某個範圍內整數的大端表示）的字典都有效。

3.3.7. 字典序列化的示例. 考慮一個具有三個 16 位元鍵 13、17 和 239（視為大端整數）以及對應的 16 位元值 169、289 和 57121 的字典。

以二進位形式：

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

對應的 Patricia 樹由根 A、兩個中間節點 B 和 C 以及三個葉節點 D、E 和 F 組成，分別對應於 13、17 和 239。根 A 只有一個子節點 B；邊 AB 上的標籤是 $00000000 = 0^8$。節點 B 有兩個子節點：其左子節點是中間節點 C，邊 BC 標記為 (0)00，而其右子節點是葉 F，BF 標記為 (1)1101111。最後，C 有兩個葉子子節點 D 和 E，CD 標記為 (0)1101，CE 標記為 (1)0001。

型別 `HashmapE 16 (## 16)` 的對應值可以以人類可讀形式寫為：

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

此資料結構序列化為單元樹由六個單元組成，其中包含以下二進位資料：

```
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
```

這裡 A 是根單元，$A.0$ 是 A 的第一個參照處的單元，$A.1$ 是 A 的第二個參照處的單元，依此類推。此單元樹可以使用 **1.0** 中描述的十六進位表示法更緊湊地表示，使用縮排來反映單元樹結構：

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

總共使用了 93 個資料位元和 5 個參照在 6 個單元中來序列化此字典。請注意，三個 16 位元鍵及其對應的 16 位元值的簡單表示已經需要 96 位元(儘管沒有任何參照)，因此這種特定的序列化相當有效。

3.3.8. 描述型別 X 的序列化的方法. 請注意，字典操作的內建 TVM 基本操作需要知道關於型別 X 的序列化的一些資訊；否則，它們將無法正確使用 *Hashmap n X*，因為型別 X 的值直接包含在 Patricia 樹葉單元中。有幾種可用選項來描述型別 X 的序列化：

- 最簡單的情況是當 $X = \hat{}Y$ 對於某個其他型別 Y。在這種情況下，X 本身的序列化總是由對單元的一個參照組成，該單元實際上必須包含型別 Y 的值，這與字典操作基本操作無關。
- 另一個簡單的情況是當型別 X 的任何值的序列化總是由 $0 \leq b \leq 1023$ 個資料位元和 $0 \leq r \leq 4$ 個參照組成。然後可以將整數 b 和 r 作為 X 的簡單描述傳遞給字典操作基本操作。(請注意，前一種情況對應於 $b = 0$、$r = 1$。)
- 更複雜的情況可以由四個整數 $1 \leq b_0, b_1 \leq 1023$、$0 \leq r_0, r_1 \leq 4$ 描述，當序列化的第一個位元等於 i 時使用 b_i 和 r_i。當 $b_0 = b_1$ 且 $r_0 = r_1$ 時，此情況簡化為前一種情況。
- 最後，型別 X 的序列化的最一般描述由 X 的分割函數 $split_X$ 給出，它接受一個 *Slice* 參數 s，並傳回兩個 *Slice*，s' 和 s''，其中 s' 是 s 的唯一前綴，它是型別 X 的值的序列化，s'' 是 s 的餘數。如果不存在這樣的前綴，則預期分割函數引發異常。請注意，支援某些或所有代數 TL-B 型別的高階語言的編譯器可能會自動為程式中定義的所有型別產生分割函數。

3.3.9. 關於 X 的序列化的簡化假設. 可以注意到，型別 X 的值總是佔據 $HashmapE\ n\ X$ 的序列化內部 `hm_edge`/`hme_leaf` 單元的剩餘部分。因此，如果我們不堅持對存取的所有字典進行嚴格驗證，我們可以假設在反序列化其 `label` 之後在 `hm_edge`/`hme_leaf` 單元中未解析的所有內容都是型別 X 的值。這極大地簡化了字典操作基本操作的建立，因為在大多數情況下它們根本不需要關於 X 的任何資訊。

3.3.10. 基本字典操作. 讓我們介紹字典（即型別 $HashmapE\ n\ X$ 的值 D）的基本操作分類：

- $\textsc{Get}(D, k)$ — 給定 $D : HashmapE(n, X)$ 和鍵 $k : n \cdot \texttt{bit}$，傳回保存在 D 中的對應值 $D[k] : X^?$。
- $\textsc{Set}(D, k, x)$ — 給定 $D : HashmapE(n, X)$、鍵 $k : n \cdot \texttt{bit}$ 和值 $x : X$，在 D 的副本 D' 中將 $D'[k]$ 設定為 x，並傳回產生的字典 D'（參見 **2.3.4**）。
- $\textsc{Add}(D, k, x)$ — 類似於 $\textsc{Set}$，但僅當鍵 k 在 D 中不存在時才將鍵值對 (k, x) 添加到 D。

- Replace(D, k, x) — 類似於 Set，但僅當鍵 k 已經存在於 D 中時才將 $D'[k]$ 更改為 x。
- GetSet、GetAdd、GetReplace — 分別類似於 Set、Add 和 Replace，但也傳回 $D[k]$ 的舊值。
- Delete(D, k) — 從字典 D 中刪除鍵 k，並傳回產生的字典 D'。
- GetMin(D)、GetMax(D) — 從字典 D 中取得最小或最大鍵 k，以及關聯的值 $x : X$。
- RemoveMin(D)、RemoveMax(D) — 類似於 GetMin 和 GetMax，但也從字典 D 中移除所討論的鍵，並傳回修改後的字典 D'。可用於迭代 D 的所有元素，有效地使用（ D 的副本）本身作為迭代器。
- GetNext(D, k) — 計算最小鍵 $k' > k$（或變體中的 $k' \geq k$）並將其與對應值 $x' : X$ 一起傳回。可用於迭代 D 的所有元素。
- GetPrev(D, k) — 計算最大鍵 $k' < k$（或變體中的 $k' \leq k$）並將其與對應值 $x' : X$ 一起傳回。
- Empty(n) — 建立空字典 $D : \mathit{HashmapE}(n, X)$。
- IsEmpty(D) — 檢查字典是否為空。
- Create$(n, \{(k_i, x_i)\})$ — 給定 n，從在堆疊中傳遞的鍵值對列表 (k_i, x_i) 建立字典。
- GetSubdict(D, l, k_0) — 給定 $D : \mathit{HashmapE}(n, X)$ 和某個 l 位元字串 $k_0 : l \cdot \texttt{bit}$（對於 $0 \leq l \leq n$），傳回 D 的子字典 $D' = D/k_0$，由以 k_0 開頭的鍵組成。結果 D' 可以是型別 $\mathit{HashmapE}(n, X)$ 或型別 $\mathit{HashmapE}(n - l, X)$。
- ReplaceSubdict(D, l, k_0, D') — 給定 $D : \mathit{HashmapE}(n, X)$、$0 \leq l \leq n$、$k_0 : l \cdot \texttt{bit}$ 和 $D' : \mathit{HashmapE}(n - l, X)$，用 D' 替換 D 的由以 k_0 開頭的鍵組成的子字典 D/k_0，並傳回產生的字典 $D'' : \mathit{HashmapE}(n, X)$。ReplaceSubdict 的某些變體也可能傳回所討論子字典 D/k_0 的舊值。
- DeleteSubdict(D, l, k_0) — 等效於 ReplaceSubdict，其中 D' 是空字典。

- SPLIT(D) — 給定 D : *HashmapE*(n, X)，傳回 $D_0 := D/0$ 和 $D_1 := D/1$: *HashmapE*($n-1, X$)，D 的兩個子字典，分別由以 0 和 1 開頭的所有鍵組成。

- MERGE(D_0, D_1) — 給定 D_0 和 D_1 : *HashmapE*($n-1, X$)，計算 D : *HashmapE*(n, X)，使得 $D/0 = D_0$ 且 $D/1 = D_1$。

- FOREACH(D, f) — 執行具有兩個引數 k 和 x 的函數 f，其中 (k, x) 以字典序遍歷字典 D 的所有鍵值對。[18]

- FOREACHREV(D, f) — 類似於 FOREACH，但以相反順序處理所有鍵值對。

- TREEREDUCE(D, o, f, g) — 給定 D : *HashmapE*(n, X)、值 $o : X$ 和兩個函數 $f : X \to Y$ 和 $g : Y \times Y \to Y$，透過首先對所有葉應用 f，然後使用 g 從分配給其子節點的值計算對應於分叉的值來執行 D 的「樹歸約」。[19]

3.3.11. 字典基本操作的分類. 字典基本操作在 **A.10** 中詳細描述，可以根據以下類別進行分類：

- 它們執行哪個字典操作（參見 **3.3.10**）？

- 它們是否專門用於 $X = ^\frown Y$ 的情況？如果是，它們是透過 *Cell* 還是透過 *Slice* 表示型別 Y 的值？（通用版本始終將型別 X 的值表示為 *Slice*。）

- 字典本身是作為 *Cell* 還是作為 *Slice* 傳遞和傳回？（大多數基本操作將字典表示為 *Slice*。）

- 鍵長度 n 是否在基本操作內部固定，還是在堆疊中傳遞？

- 鍵是透過 *Slice* 表示，還是透過有號或無號 *Integer* 表示？

此外，TVM 包括特殊的序列化/反序列化基本操作，例如 `STDICT`、`LDDICT` 和 `PLDDICT`。它們可用於從包含物件的序列化中提取字典，或將字典插入此類序列化中。

[18]實際上，f 可能接收 m 個額外引數並傳回 m 個修改後的值，這些值傳遞給 f 的下一次呼叫。這可以用於實作字典的「映射」和「歸約」操作。

[19]可以引入此操作的版本，其中 f 和 g 接收額外的位元字串引數，等於對應子樹中的鍵（對於葉）或所有鍵的公共前綴（對於分叉）。

3.4 具有可變長度鍵的雜湊映射

TVM 除了支援具有固定長度鍵的字典（如上面 **3.3** 中所述）外，還為具有可變長度鍵的字典或雜湊映射提供了一些支援。

3.4.1. 具有可變長度鍵的字典的序列化. *VarHashmap* 序列化為單元樹（或更一般地，序列化為 *Slice*）由 TL-B 方案定義，類似於 **3.3.3** 中描述的：

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

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

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

3.4.2. 前綴碼的序列化. 具有可變長度鍵的字典的一個特殊情況是前綴碼，其中鍵不能是彼此的前綴。這樣的字典中的值可能僅出現在 Patricia 樹的葉中。

前綴碼的序列化由以下 TL-B 方案定義：

```
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 控制流、繼續和異常

本章描述繼續，它可以表示 TVM 中的執行權杖和異常處理器。繼續與 TVM 程式的控制流深度相關；特別是，子程式呼叫以及條件和迭代執行在 TVM 中使用接受一個或多個繼續作為其引數的特殊基本操作實作。

我們以討論遞迴問題和互遞迴函數家族的問題來結束本章，這個問題因 TVM 資料結構（包括 TVM 程式碼）中不允許迴圈參照而加劇。

4.1 繼續和子程式

回想一下（參見 **1.1.3**）*Continuation* 值表示可以稍後執行的「執行權杖」——例如，透過 EXECUTE=CALLX（「執行」或「間接呼叫」）或 JMPX（「間接跳轉」）基本操作。因此，繼續負責程式的執行，並被控制流基本操作大量使用，使得子程式呼叫、條件表示式、迴圈等成為可能。

4.1.1. 普通繼續. 最常見的繼續是普通繼續，包含以下資料：

- *Slice* code（參見 **1.1.3** 和 **3.2.2**），包含要執行的 TVM 程式碼（的餘數）。
- （可能為空的）*Stack* stack，包含要執行的程式碼的堆疊的原始內容。
- （可能為空的）對 $(\mathtt{c}(i), v_i)$ 的列表 save（也稱為「儲存列表」），包含在執行程式碼之前要恢復的控制暫存器的值。
- 16 位元整數值 cp，選擇用於解釋來自 code 的 TVM 程式碼的 TVM 程式碼頁。
- 可選的非負整數 nargs，指示繼續期望的引數數量。

4.1.2. 簡單普通繼續. 在大多數情況下，普通繼續是最簡單的，具有空的 stack 和 save。它們本質上由對要執行的程式碼（的餘數）的參照 code 和解碼此程式碼中的指令時要使用的程式碼頁 cp 組成。

4.1.3. 目前繼續 cc. 「目前繼續」cc 是 TVM 總狀態的重要組成部分，表示目前正在執行的程式碼（參見 **1.1**）。特別是，在討論所有其他基本操作時我們所說的「目前堆疊」（或簡稱「堆疊」）實際上是目前繼續的堆疊。TVM 總狀態的所有其他元件也可以被認為是目前繼續 cc 的一部分；但是，由於性能原因，它們可以從目前繼續中提取並作為總狀態的一部分單獨保存。這就是為什麼我們在 **1.4** 中將堆疊、控制暫存器和程式碼頁描述為 TVM 狀態的單獨部分。

4.1.4. TVM 的正常工作，或主迴圈. TVM 通常執行以下操作：

如果目前繼續 cc 是普通的，它從 *Slice* code 解碼第一條指令，類似於 TVM LD* 基本操作反序列化其他單元的方式（參見 **3.2** 和 **3.2.11**）：它首先解碼操作碼，然後解碼指令的參數（例如，指示堆疊操作基本操作涉及的「堆疊暫存器」的 4 位元欄位，或「推送常數」或「字面量」基本操作的常數值）。然後將 *Slice* 的餘數放入新 cc 的 code 中，並在目前堆疊上執行解碼的操作。整個過程重複，直到 cc.code 中沒有剩餘操作。

如果 code 為空（即，不包含資料位元和參照），或者遇到（很少需要的）明確子程式傳回（RET）指令，則丟棄目前繼續，並將控制暫存器 c0 中的「傳回繼續」載入 cc 中（此過程在 **4.1.6** 開始更詳細地討論）。[20] 然後透過從新的目前繼續解析操作來繼續執行。

4.1.5. 特殊繼續. 除了到目前為止考慮的普通繼續（參見 **4.1.1**）之外，TVM 還包括一些特殊繼續，表示某些不太常見的狀態。特殊繼續的示例包括：

- 繼續 ec_quit，其參數設定為零，表示 TVM 工作的結束。當 TVM 開始執行智慧合約的程式碼時，此繼續是 c0 的原始值。
- 繼續 ec_until，它包含對表示正在執行的迴圈主體和迴圈後要執行的程式碼的其他兩個繼續（普通或非普通）的參照。

TVM 執行特殊繼續取決於其特定類別，並且與 **4.1.4** 中描述的普通繼續的操作不同。[21]

4.1.6. 切換到另一個繼續：JMP 和 RET. 切換到另一個繼續 c 的過程可以透過諸如 JMPX（從堆疊中取 c）或 RET（使用 c0 作為 c）之類的指令執行。此過程比簡單地將 cc 的值設定為 c 稍微複雜一些：在執行此操作之前，目前堆疊中的所有值或前 n 個值都會移動到繼續 c 的堆疊，然後才丟棄目前堆疊的餘數。

如果需要移動所有值（最常見的情況），並且如果繼續 c 有空堆疊（也是最常見的情況；請注意，假設特殊繼續有空堆疊），那麼 c 的新堆疊等於目前繼續的堆疊，因此我們可以簡單地將目前堆疊完整地傳輸到 c。(如果我們將目前堆疊作為 TVM 總狀態的單獨部分保存，我們根本不需要做任何事情。)

[20] 如果 code 中沒有剩餘資料位元，但仍然恰好有一個參照，則執行對該參照處單元的隱式 JMP，而不是隱式 RET。

[21] 從技術上講，TVM 可能只是呼叫目前在 cc 中的繼續的虛擬方法 run()。

4.1.7. 確定傳遞給下一個繼續 c 的引數數量 n. 預設情況下，n 等於目前堆疊的深度。但是，如果 c 有明確的 nargs 值（要提供的引數數量），則 n 計算為 n'，等於 c.nargs 減去 c 堆疊的目前深度。

此外，還有 JMPX 和 RET 的特殊形式，提供明確值 n''，要從目前堆疊傳遞給繼續 c 的參數數量。如果提供 n''，它必須小於或等於目前堆疊的深度，否則發生堆疊下溢異常。如果同時提供 n' 和 n''，我們必須有 $n' \leq n''$，在這種情況下使用 $n = n'$。如果提供 n'' 而不提供 n'，則使用 $n = n''$。

人們也可以想像 n'' 的預設值等於原始堆疊的深度，並且即使只有其中的 n' 個實際移動到下一個繼續 c 的堆疊，n'' 個值總是從原始堆疊頂部移除。儘管目前堆疊的餘數之後被丟棄，但這種描述稍後會變得有用。

4.1.8. 從新繼續 c 恢復控制暫存器. 計算新堆疊後，相應地恢復 c.save 中存在的控制暫存器的值，並且目前程式碼頁 cp 也設定為 c.cp。只有這樣，TVM 才會將 cc 設定為新的 c 並開始其執行。[22]

4.1.9. 子程式呼叫：CALLX 或 EXECUTE 基本操作. 將繼續作為子程式執行比切換到繼續稍微複雜一些。

考慮 CALLX 或 EXECUTE 基本操作，它從（目前）堆疊中取繼續 c 並將其作為子程式執行。

除了執行 **4.1.6** 和 **4.1.7** 中描述的堆疊操作以及如 **4.1.8** 中所述設定新的控制暫存器和程式碼頁之外，這些基本操作還執行幾個額外的步驟：

1. 從目前堆疊中移除前 n'' 個值後（參見 **4.1.7**），(通常為空的）餘數不會被丟棄，而是儲存在（舊的）目前繼續 cc 中。
2. 特殊暫存器 c0 的舊值儲存到（先前為空的）儲存列表 cc.save 中。
3. 因此修改的繼續 cc 不會被丟棄，而是設定為新的 c0，它為正在呼叫的子程式執行「下一個繼續」或「傳回繼續」的角色。
4. 之後，切換到 c 如前所述繼續。特別是，一些控制暫存器從 c.save 恢復，可能會覆寫在前一步驟中設定的 c0 值。(因此，一個好的優化是從一開始就檢查 $c0$ 是否存在於 c.save 中，並在這種情況下跳過前三個步驟，因為它們沒有用。)

這樣，被呼叫的子程式可以透過將目前繼續切換到儲存在 c0 中的傳回繼續來將控制傳回給呼叫者。嵌套的子程式呼叫正確工作，因為 c0 的前一個值最終儲存到新 c0 的控制暫存器儲存列表 c0.save 中，之後從中恢復。

[22]在執行開始之前，新 cc 的已使用儲存列表 cc.save 被清空。

4.1.10. 確定傳遞給子程式和/或從子程式接受的傳回值的引數數量. 類似於 JMPX 和 RET，CALLX 也有特殊的（很少使用的）形式，允許我們明確指定從目前堆疊傳遞給被呼叫子程式的引數數量 n''（預設情況下，n'' 等於目前堆疊的深度，即完整傳遞）。此外，可以指定第二個數字 n'''，用於在將修改的 cc 繼續儲存到新 c0 之前設定其 nargs；新的 nargs 等於舊堆疊的深度減去 n'' 加上 n'''。這意味著呼叫者願意將恰好 n'' 個引數傳遞給被呼叫的子程式，並願意在其位置接受恰好 n''' 個結果。

這樣的 CALLX 和 RET 形式主要用於接受函數引數並希望安全呼叫它們的函式庫函數。另一個應用與 TVM 的「虛擬化支援」相關，它使 TVM 程式碼能夠在「虛擬 TVM 機器」內部執行其他 TVM 程式碼。這樣的虛擬化技術可能對在 TON 區塊鏈中實作複雜的支付通道很有用（參見 [1, 5]）。

4.1.11. CALLCC：使用目前繼續呼叫. 請注意，TVM 支援一種形式的「使用目前繼續呼叫」基本操作。也就是說，基本操作 CALLCC 類似於 CALLX 或 JMPX，它從堆疊中取一個繼續 c 並切換到它；然而，CALLCC 不會丟棄前一個目前繼續 c'（如 JMPX 所做的那樣）並且不會將 c' 寫入 c0（如 CALLX 所做的那樣），而是將 c' 推入（新的）堆疊作為 c 的額外引數。基本操作 JMPXDATA 做類似的事情，但只將前一個目前繼續的程式碼（餘數）作為 *Slice* 推入。

4.2 控制流基本操作：條件和迭代執行

4.2.1. 條件執行：IF、IFNOT、IFELSE. EXECUTE（或 CALLX）的一個重要修改形式是其條件形式。例如，IF 接受一個整數 x 和一個繼續 c，並僅在 x 非零時執行 c（以 EXECUTE 會做的相同方式）；否則這兩個值僅從堆疊中丟棄。類似地，IFNOT 接受 x 和 c，但僅在 $x = 0$ 時執行 c。最後，IFELSE 接受 x、c 和 c'，從堆疊中移除這些值，並在 $x \neq 0$ 時執行 c 或在 $x = 0$ 時執行 c'。

4.2.2. 迭代執行和迴圈. EXECUTE 的更複雜的修改形式包括：

- REPEAT — 接受一個整數 n 和一個繼續 c，並執行 c n 次。[23]
- WHILE — 接受 c' 和 c''，執行 c'，然後從堆疊中取頂部值 x。如果 x 非零，它執行 c''，然後透過再次執行 c' 開始新的迴圈；如果 x 為零，它停止。

[23] REPEAT 的實作涉及一個非常規繼續，它記住剩餘的迭代次數、迴圈主體 c 和傳回繼續 c'。（後者代表呼叫 REPEAT 的函數主體的餘數，通常儲存在新 cc 的 c0 中。）

- UNTIL — 接受 c，執行它，然後從堆疊中取頂部整數 x。如果 x 為零，開始新的迭代；如果 x 非零，恢復先前執行的程式碼。

4.2.3. 常數或字面繼續. 我們看到我們可以在 TVM 程式碼中建立任意複雜的條件表達式和迴圈，只要我們有辦法將常數繼續推入堆疊。事實上，TVM 包含特殊版本的「字面」或「常數」基本操作，它們從目前程式碼 cc.code 的餘數中截取下 n 個位元組或位元到一個單元切片中，然後將其推入堆疊，不是作為 *Slice*（如 PUSHSLICE 所做的那樣），而是作為一個簡單的普通 *Continuation*（它只有 code 和 cp）。

這些基本操作中最簡單的是 PUSHCONT，它有一個立即引數 n，描述要轉換為簡單繼續的後續位元組（在面向位元組的 TVM 版本中）或位元的數量。另一個基本操作是 PUSHREFCONT，它從目前繼續 cc.code 中移除第一個單元參照，將參照的單元轉換為單元切片，最後將單元切片轉換為簡單繼續。

4.2.4. 常數繼續與條件或迭代執行基本操作組合. 因為常數繼續經常用作條件或迭代執行基本操作的引數，在 TVM 的未來修訂版中可能會定義這些基本操作的組合版本（例如，IFCONT 或 UNTILREFCONT），它們將 PUSHCONT 或 PUSHREFCONT 與另一個基本操作組合。如果檢查產生的程式碼，IFCONT 看起來非常像更傳統的「條件向前分支」指令。

4.3 繼續操作

4.3.1. 繼續是不透明的. 請注意，至少在 TVM 的目前版本中，所有繼續都是不透明的，這意味著沒有辦法修改繼續或檢查其內部資料。繼續的幾乎唯一用途是將其提供給控制流基本操作。

雖然在 TVM 中包含對非不透明繼續的支援（以及虛擬化所需的不透明繼續）有一些論點，但目前的修訂版不提供這樣的支援。

4.3.2. 允許的繼續操作. 然而，對不透明繼續的一些操作仍然是可能的，主要是因為它們等同於「建立一個新繼續，它將做一些特殊的事情，然後呼叫原始繼續」這類操作。允許的繼續操作包括：

- 將一個或多個值推入繼續 c 的堆疊中（從而建立函數的部分應用或閉包）。

- 在繼續 c 的儲存列表 c.save 內部設定控制暫存器 c(i) 的儲存值。如果該控制暫存器已經有一個值，此操作會靜默地不做任何事情。

4.3.3. 範例：控制暫存器操作. TVM 有一些基本操作來設定和檢查控制暫存器的值。其中最重要的是 PUSH c(i)（將 c(i) 的目前值推入堆疊）和 POP c(i)（如果提供的值具有正確的型別，則從堆疊設定 c(i) 的值）。然而，還有後者指令的修改版本，稱為 POPSAVE c(i)，它在設定新值之前將 c(i) 的舊值（對於 $i > 0$）儲存到 c0 的繼續中，如 **4.3.2** 中所述。

4.3.4. 範例：在函數程式碼中設定引數數量. 基本操作 LEAVEARGS n 展示了繼續在操作中的另一個應用：它只保留目前堆疊的頂部 n 個值，並將餘數移動到 c0 中繼續的堆疊。此基本操作使被呼叫的函數能夠將不需要的引數「傳回」給其呼叫者的堆疊，這在某些情況下很有用（例如，與異常處理相關的情況）。

4.3.5. 布林電路. 繼續 c 可以被認為是一段程式碼，具有兩個可選的退出點，保存在 c 的儲存列表中：由 c.c0 := c.save(c0) 給出的主要退出點，以及由 c.c1 := c.save(c1) 給出的輔助退出點。如果執行，繼續執行它被建立時要執行的任何動作，然後（通常）將控制轉移到主要退出點，或者在某些情況下轉移到輔助退出點。我們有時說一個同時定義了退出點 c.c0 和 c.c1 的繼續 c 是一個雙退出繼續，或一個布林電路，特別是如果退出點的選擇取決於某個內部檢查的條件。

4.3.6. 繼續的組合. 可以透過將 c.c0 或 c.c1 設定為 c' 來組合 兩個繼續 c 和 c'。這建立了一個新繼續，表示為 $c \circ_0 c'$ 或 $c \circ_1 c'$，它與 c 在其儲存列表上有所不同。(回想一下，如果 c 的儲存列表已經有與相關控制暫存器對應的條目，如 **4.3.2** 中所解釋的，這樣的操作會靜默地不做任何事情)。

透過組合繼續，可以建立鏈或其他圖，可能帶有迴圈，表示控制流。事實上，產生的圖類似於流程圖，其中布林電路對應於「條件節點」(包含將根據某些條件將控制轉移到 c0 或 c1 的程式碼)，而單退出繼續對應於「動作節點」。

4.3.7. 基本繼續組合基本操作. 用於組合繼續的兩個基本基本操作是 COMPOS (也稱為 SETCONT c0 和 BOOLAND) 和 COMPOSALT (也稱為 SETCONT c1 和 BOOLOR)，它們從堆疊中取 c 和 c'，將 c.c0 或 c.c1 設定為 c'，並傳回產生的繼續 $c'' = c \circ_0 c'$ 或 $c \circ_1 c'$。所有其他繼續組合操作都可以用這兩個基本操作表示。

4.3.8. 進階繼續組合基本操作. 然而，TVM 不僅可以組合從堆疊中取的繼續，還可以從 c0 或 c1 或從目前繼續 cc 取的繼續；同樣，結果可以推入堆疊、儲存到 c0 或 c1 中，或用作新的目前繼續 (即，將控制轉移到它)。此外，TVM 可以定義條件組合基本操作，僅在從堆疊中取的整數值非零時執行上述某些動作。

例如，EXECUTE 可以描述為 $\mathtt{cc} \leftarrow c \circ_0 \mathtt{cc}$，繼續 c 從原始堆疊中取得。類似地，JMPX 是 $\mathtt{cc} \leftarrow c$，而 RET（在布林電路上下文中也稱為 RETTRUE）是 $\mathtt{cc} \leftarrow \mathtt{c0}$。其他有趣的基本操作包括 THENRET（$c' \leftarrow c \circ_0 \mathtt{c0}$）和 ATEXIT（$\mathtt{c0} \leftarrow c \circ_0 \mathtt{c0}$）。

最後，一些「實驗性」基本操作也涉及 c1 和 $\circ_1$。例如：

- RETALT 或 RETFALSE 執行 $\mathtt{cc} \leftarrow \mathtt{c1}$。
- RET 和 RETALT 的條件版本也可能有用：RETBOOL 從堆疊中取一個整數 x，如果 $x \neq 0$ 則執行 RETTRUE，否則執行 RETFALSE。
- INVERT 執行 $\mathtt{c0} \leftrightarrow \mathtt{c1}$；如果 c0 和 c1 中的兩個繼續表示我們應該根據某個布林表達式選擇的兩個分支，INVERT 在外層否定此表達式。
- INVERTCONT 對從堆疊中取的繼續 c 執行 $c.\mathtt{c0} \leftrightarrow c.\mathtt{c1}$。
- ATEXIT 的變體包括 ATEXITALT（$\mathtt{c1} \leftarrow c \circ_1 \mathtt{c1}$）和 SETEXITALT（$\mathtt{c1} \leftarrow (c \circ_0 \mathtt{c0}) \circ_1 \mathtt{c1}$）。
- BOOLEVAL 從堆疊中取一個繼續 c 並執行 $\mathtt{cc} \leftarrow \big((c \circ_0 (\mathtt{PUSH} - 1)) \circ_1 (\mathtt{PUSH0})\big) \circ_0 \mathtt{cc}$。如果 c 表示布林電路，淨效果是評估它並在繼續之前將 -1 或 0 推入堆疊。

4.4 繼續作為物件

4.4.1. 使用繼續表示物件. Smalltalk（或 Objective C）風格的物件導向程式設計可以借助繼續來實作。為此，物件由特殊繼續 o 表示。如果它有任何資料欄位，它們可以保存在 o 的堆疊中，使 o 成為部分應用（即，具有非空堆疊的繼續）。

當某人想要使用引數 x_1、x_2、…、x_n 呼叫 o 的方法 m 時，她將引數推入堆疊，然後推入對應於方法 m 的魔術數字，然後執行 o 傳遞 $n+1$ 個引數（參見 **4.1.10**）。然後 o 使用堆疊頂部整數 m 來選擇具有所需方法的分支，並執行它。如果 o 需要修改其狀態，它只是計算相同類型的新繼續 o'（也許具有與 o 相同的程式碼，但具有不同的初始堆疊）。新繼續 o' 與任何其他需要傳回的傳回值一起傳回給呼叫者。

4.4.2. 可序列化物件. 將 Smalltalk 風格的物件表示為繼續或甚至單元樹的另一種方式包括使用 JMPREFDATA 基本操作（JMPXDATA 的變體，參見 **4.1.11**），它從目前繼續的程式碼中取第一個單元參照，將參照的單元轉換為簡單的普通繼續，並將控制轉移到它，首先將目前繼續的餘數作為

Slice 推入堆疊。這樣，物件可以由單元 $\tilde{o}$ 表示，該單元在其資料的開頭包含 JMPREFDATA，並在第一個參照中包含物件的實際程式碼（可以說單元 $\tilde{o}$ 的第一個參照是物件 $\tilde{o}$ 的類別）。此單元的剩餘資料和參照將用於儲存物件的欄位。

這樣的物件的優點是它們是單元樹，而不僅僅是繼續，這意味著它們可以儲存到 TON 智慧合約的持久儲存中。

4.4.3. 唯一繼續和能力. 在 TVM 的未來修訂版中，將一些繼續標記為唯一的 可能是有意義的，這意味著它們不能被複製，即使以延遲方式，透過將其參照計數器增加到大於 1 的值。如果不透明繼續是唯一的，它實際上變成了能力，可以被其所有者精確使用一次或轉移給其他人。

例如，想像一個表示印表機輸出串流的繼續（這是用作物件的繼續的範例，參見 **4.4.1**）。當使用一個整數引數 n 呼叫時，此繼續將程式碼為 n 的字元輸出到印表機，並傳回反映串流新狀態的相同類型的新繼續。顯然，複製這樣的繼續並並行使用兩個副本會導致一些意外的副作用；將其標記為唯一將禁止這種不利的使用。

4.5 異常處理

TVM 的異常處理非常簡單，包括將控制轉移到保存在控制暫存器 c2 中的繼續。

4.5.1. 異常處理器的兩個引數：異常參數和異常編號. 每個異常由兩個引數表徵：異常編號（一個 *Integer*）和異常參數（任何值，最常見的是零 *Integer*）。異常編號 0–31 保留給 TVM，而所有其他異常編號可用於使用者定義的異常。

4.5.2. 丟擲異常的基本操作. 有幾個特殊的基本操作用於丟擲異常。其中最通用的是 THROWANY，它從堆疊中取兩個引數 v 和 $0 \leq n < 2^{16}$，並丟擲編號為 n 和值為 v 的異常。此基本操作有變體，假設 v 為零整數，將 n 儲存為字面值，和/或在從堆疊中取的整數值上是條件的。如果需要，使用者定義的異常可以使用任意值作為 v（例如，單元樹）。

4.5.3. TVM 產生的異常. 當然，一些異常由正常基本操作產生。例如，每當算術運算的結果不適合有符號 257 位元整數時，會產生算術溢位異常。在這種情況下，異常的引數 v 和 n 由 TVM 本身確定。

4.5.4. 異常處理. 異常處理本身包括控制轉移到異常處理器—即，在控制暫存器 c2 中指定的繼續，將 v 和 n 作為此繼續的兩個引數提供，就好像

已請求帶有 $n'' = 2$ 個引數的 JMP 到 c2（參見 **4.1.7** 和 **4.1.6**）。因此，v 和 n 最終出現在異常處理器的堆疊頂部。舊堆疊的餘數被丟棄。

請注意，如果 c2 中的繼續在其儲存列表中有 c2 的值，它將用於在執行異常處理器之前設定 c2 的新值。特別是，如果異常處理器呼叫 THROWANY，它將使用 c2 的恢復值重新丟擲原始異常。此技巧使異常處理器僅處理某些異常，並將其餘的傳遞給外部異常處理器。

4.5.5. 預設異常處理器. 當建立 TVM 實例時，c2 包含對「預設異常處理器繼續」的參照，這是一個 ec_fatal 非常規繼續（參見 **4.1.5**）。它的執行導致 TVM 執行終止，異常的引數 v 和 n 傳回給外部呼叫者。在 TON 區塊鏈的上下文中，n 將作為交易結果的一部分儲存。

4.5.6. TRY 基本操作. TRY 基本操作可用於實作類似 C++ 的異常處理。此基本操作接受兩個繼續 c 和 c'。它將 c2 的舊值儲存到 c' 的儲存列表中，將 c2 設定為 c'，並像 EXECUTE 一樣執行 c，但還將 c2 的舊值儲存到新 c0 的儲存列表中。通常使用帶有明確引數數量 n'' 的 TRY 基本操作版本，傳遞給繼續 c。

淨結果大致等同於 C++ 的 try { c } catch(...) { c' } 運算子。

4.5.7. 預定義異常列表. TVM 的預定義異常對應於範圍 0–31 中的異常編號 n。它們包括：

- 正常終止（$n = 0$）— 不應該被產生，但它對某些技巧很有用。
- 替代終止（$n = 1$）— 同樣，不應該被產生。
- 堆疊下溢（$n = 2$）— 基本操作的堆疊中沒有足夠的引數。
- 堆疊溢位（$n = 3$）— 堆疊上儲存的值比此版本的 TVM 允許的值更多。
- 整數溢位（$n = 4$）— 整數不適合 $-2^{256} \le x < 2^{256}$，或發生了除以零。
- 範圍檢查錯誤（$n = 5$）— 整數超出預期範圍。
- 無效操作碼（$n = 6$）— 指令或其立即引數無法解碼。
- 型別檢查錯誤（$n = 7$）— 基本操作的引數具有不正確的值型別。
- 單元溢位（$n = 8$）— 序列化基本操作之一中的錯誤。

- 單元下溢（$n = 9$）— 反序列化錯誤。
- 字典錯誤（$n = 10$）— 反序列化字典物件時的錯誤。
- 未知錯誤（$n = 11$）— 未知錯誤，可能由使用者程式丟擲。
- 致命錯誤（$n = 12$）— 在被認為不可能的情況下由 TVM 丟擲。
- *Gas* 不足（$n = 13$）— 當剩餘 gas（g_r）變為負數時由 TVM 丟擲。此異常通常無法被捕獲，並導致 TVM 立即終止。

這些異常中的大多數沒有參數（即，使用零整數代替）。檢查這些異常的順序在下面的 **4.5.8** 中概述。

4.5.8. 堆疊下溢、型別檢查和範圍檢查異常的順序. 所有 TVM 基本操作首先檢查堆疊是否包含所需數量的引數，如果不是這種情況，則產生堆疊下溢異常。只有這樣，才檢查引數的型別標記及其範圍（例如，如果基本操作期望引數不僅是 *Integer*，而且在 0 到 256 的範圍內），從堆疊頂部的值（最後一個引數）開始並繼續深入堆疊。如果引數的型別不正確，則產生型別檢查異常；如果型別正確，但值不落入預期範圍，則產生範圍檢查異常。

一些基本操作接受可變數量的引數，這取決於位於堆疊頂部附近的某個小的固定引數子集的值。在這種情況下，上述過程首先針對此小子集中的所有引數執行。然後，一旦從已處理的引數中確定了剩餘引數的數量和型別，就對剩餘引數重複該過程。

4.6 函數、遞迴和字典

4.6.1. 遞迴問題. **4.2** 中描述的條件和迭代執行基本操作—以及 **4.1** 中描述的無條件分支、呼叫和傳回基本操作—使人們能夠實作或多或少任意的程式碼，具有嵌套迴圈和條件表達式，但有一個值得注意的例外：只能從目前繼續的部分建立新的常數繼續。（特別是，不能以這種方式從自身呼叫子程式。）因此，正在執行的程式碼—即目前繼續—逐漸變得越來越小。[24]

4.6.2. Y-組合子解決方案：將繼續作為引數傳遞給自身. 處理遞迴問題的一種方法是將表示遞迴函數主體的繼續的副本作為額外引數傳遞給自身。例如，考慮以下階乘函數的程式碼：

[24]這裡的重要點是表示 TVM 程式的單元樹不能有循環參照，因此使用 `CALLREF` 以及對樹中較高單元的參照將不起作用。

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

這大致對應於定義一個輔助函數 $body$，它有三個引數 n、x 和 f，使得 $body(n, x, f)$ 在 $n < 2$ 時等於 x，否則等於 $f(n-1, nx, f)$，然後呼叫 $body(n, 1, body)$ 來計算 n 的階乘。然後使用 DUP; EXECUTE 構造來實作遞迴，或者在尾遞迴的情況下使用 DUP; JMPX。此技巧等同於將 Y-組合子應用於函數 $body$。

4.6.3. Y-組合子解決方案的變體. 另一種遞迴計算階乘的方法，更接近經典的遞迴定義

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

如下：

```
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
```

階乘函數的這個定義比前一個定義短兩個位元組，但它使用一般遞迴而不是尾遞迴，因此不能輕易轉換為迴圈。

4.6.4. 比較：階乘函數的非遞迴定義. 順便說一下，借助 REPEAT 迴圈的階乘非遞迴定義也是可能的，而且它比兩個遞迴定義都短得多：

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

4.6.5. 幾個相互遞迴的函數. 如果有一個相互遞迴函數的集合 f_1、…、f_n，可以使用相同的技巧，將整個繼續集合 $\{f_i\}$ 作為額外的 n 個引數傳遞給這些函數中的每一個。然而，隨著 n 增長，這變得越來越麻煩，因為必須重新排列堆疊中的這些額外引數以處理「真正的」引數，然後在任何遞迴呼叫之前將它們的副本推入堆疊頂部。

4.6.6. 將多個函數組合成一個元組. 也可以將表示函數 f_1、…、f_n 的繼續集合組合成一個「元組」$\mathbf{f} := (f_1, \ldots, f_n)$，並將此元組作為一個堆疊元素 **f** 傳遞。例如，當 $n \leq 4$ 時，每個函數可以由單元 $\tilde{f}_i$（以及根植於此單元的

單元樹）表示，元組可以由單元 $\tilde{\mathbf{f}}$ 表示，它有對其元件單元 $\tilde{f_i}$ 的參照。然而，這將導致在每次遞迴呼叫之前需要從此元組「解包」所需的元件。

4.6.7. 將多個函數組合成選擇器函數. 另一種方法是將多個函數 f_1、…、f_n 組合成一個「選擇器函數」f，它從堆疊頂部取一個額外的引數 i，$1 \leq i \leq n$，並呼叫適當的函數 f_i。像 TVM 這樣的堆疊機器非常適合這種方法，因為它們不要求函數 f_i 具有相同數量和型別的引數。使用這種方法，只需要向這些函數中的每一個傳遞一個額外的引數 f，並在每次遞迴呼叫 f 之前向堆疊推入一個額外的引數 i 來選擇要呼叫的正確函數。

4.6.8. 使用專用暫存器來保存選擇器函數. 然而，即使我們使用前兩種方法之一將所有函數組合成一個額外的引數，將此引數傳遞給所有相互遞迴的函數仍然非常麻煩，並且需要大量額外的堆疊操作操作。因為此引數很少更改，可以使用專用暫存器來保存它並透明地將其傳遞給所有呼叫的函數。這是 TVM 預設使用的方法。

4.6.9. 選擇器函數的特殊暫存器 c3. 事實上，TVM 使用專用暫存器 c3 來保存表示目前或全域「選擇器函數」的繼續，它可以用於呼叫相互遞迴函數家族中的任何一個。特殊基本操作 CALL nn 或 CALLDICT nn（參見 **A.8.7**）等同於 PUSHINT nn; PUSH c3; EXECUTE，類似地 JMP nn 或 JMPDICT nn 等同於 PUSHINT nn; PUSH c3; JMPX。這樣，TVM 程式（最終是相互遞迴函數的大集合）可以使用表示程式中所有函數家族的正確選擇器函數初始化 c3，然後使用 CALL nn 透過其索引（有時也稱為函數的選擇器）呼叫這些函數中的任何一個。

4.6.10. c3 的初始化. TVM 程式可能透過 POP c3 指令初始化 c3。然而，因為這通常是程式（例如，智慧合約）執行的第一個動作，TVM 為 c3 的自動初始化做了一些規定。也就是說，c3 由程式本身的程式碼（cc 的初始值）初始化，並且在程式執行之前將額外的零（或者在某些情況下，某個其他預定義的數字 s）推入堆疊。這大致等同於在程式的最開始呼叫 JMPDICT 0（或 JMPDICT s）—即，索引為零的函數實際上是程式的 main() 函數。

4.6.11. 建立選擇器函數和 switch 語句. TVM 為選擇器函數（通常構成 TVM 程式的頂層）的簡單和簡潔實作做了特殊規定，或者更一般地說，任意 switch 或 case 語句（在 TVM 程式中也很有用）。為此目的包含的最重要的基本操作是 IFBITJMP、IFNBITJMP、IFBITJMPREF 和 IFNBITJMPREF（參見 **A.8.2**）。它們有效地使人們能夠將子程式（保存在單獨的單元中或作為某些單元的子切片）組合成二元決策樹，根據在堆疊頂部傳遞的整數的指定位元做出決策。

另一個對實作和-積型別有用的指令是 PLDUZ（參見 **A.7.2**）。此指令將 *Slice* 的前幾個位元預載入到 *Integer* 中，稍後可以透過 IFBITJMP 和其他類似指令檢查。

4.6.12. 替代方法：使用雜湊映射選擇正確的函數. 另一個替代方法是使用 *Hashmap*（參見 **3.3**）來保存程式中所有函數的程式碼「集合」或「字典」，並使用雜湊映射查找基本操作（參見 **A.10**）來選擇所需函數的程式碼，然後可以將其 BLESS 成繼續（參見 **A.8.5**）並執行。特殊的組合「查找、祝福和執行」基本操作，如 DICTIGETJMP 和 DICTIGETEXEC，也可用（參見 **A.10.11**）。這種方法對於較大的程式和 switch 語句可能更有效率。

5 程式碼頁和指令編碼

本章描述程式碼頁機制，它允許 TVM 靈活且可擴展，同時保留對先前產生的程式碼的向後相容性。

我們還討論了一些關於指令編碼的一般考慮（適用於任意機器碼，不僅僅是 TVM），以及這些考慮對 TVM 的影響以及在設計 TVM 的（實驗性）程式碼頁零時所做的選擇。指令編碼本身稍後在附錄 **A** 中呈現。

5.1 程式碼頁和不同 TVM 版本的互操作性

程式碼頁 是向後相容性和 TVM 未來擴展的基本機制。它們使為不同修訂版 TVM 編寫的程式碼能夠透明執行，並在這些程式碼的實例之間透明互動。然而，程式碼頁的機制足夠通用和強大，可以啟用一些最初未預期的其他應用程式。

5.1.1. 繼續中的程式碼頁. 每個普通繼續都包含一個 16 位元程式碼頁 欄位 `cp`（參見 **4.1.1**），它決定將用於執行其程式碼的程式碼頁。如果繼續由 `PUSHCONT`（參見 **4.2.3**）或類似的基本操作建立，它通常繼承目前程式碼頁（即，`cc` 的程式碼頁）。[25]

5.1.2. 目前程式碼頁. 目前程式碼頁 `cp`（參見 **1.4**）是目前繼續 `cc` 的程式碼頁。它決定下一條指令將如何從 `cc.code`（目前繼續的程式碼的餘數）解碼。一旦指令被解碼和執行，它決定目前程式碼頁的下一個值。在大多數情況下，目前程式碼頁保持不變。

另一方面，所有切換目前繼續的基本操作從新的目前繼續載入 `cp` 的新值。這樣，繼續中的所有程式碼總是按照其預期的方式被解釋。

5.1.3. 不同版本的 TVM 可能使用不同的程式碼頁. 不同版本的 TVM 可能為其程式碼使用不同的程式碼頁。例如，TVM 的原始版本可能使用程式碼頁零。較新的版本可能使用程式碼頁一，它包含所有先前定義的操作碼，以及一些新定義的操作碼，使用一些先前未使用的操作碼空間。後續版本可能使用另一個程式碼頁，等等。

然而，較新版本的 TVM 將像以前一樣執行程式碼頁零的舊程式碼。如果舊程式碼包含一個操作碼，用於在原始版本的 TVM 中未定義的某些新操作，它仍然會產生無效操作碼異常，因為新操作在程式碼頁零中不存在。

[25]這不完全正確。更準確的說法是通常新建立的繼續的程式碼頁是目前程式碼頁的已知函數。

5.1.4. 更改舊操作的行為. 新程式碼頁也可以更改舊程式碼頁中存在的某些操作的效果，同時保留它們的操作碼和助記符。

例如，想像 TVM 的未來 513 位元升級（取代目前的 257 位元設計）。它可能在與以前相同的算術基本操作中使用 513 位元 *Integer* 型別。然而，雖然新程式碼頁中的操作碼和指令看起來與舊的完全一樣，但它們會以不同的方式工作，接受 513 位元整數引數和結果。另一方面，在程式碼頁零中執行相同程式碼期間，每當算術和其他基本操作中使用的整數不適合 257 位元時，新機器將產生異常。[26] 這樣，升級不會更改舊程式碼的行為。

5.1.5. 改進指令編碼. 程式碼頁的另一個應用是更改指令編碼，反映對程式碼庫中這些指令的實際頻率的改進知識。在這種情況下，新程式碼頁將具有與舊程式碼頁完全相同的指令，但具有不同的編碼，可能具有不同的長度。例如，可以建立第一個版本的 TVM 的實驗版本，使用（前綴）位元碼而不是原始位元組碼，目的是實作更高的程式碼密度。

5.1.6. 使指令編碼與上下文相關. 使用程式碼頁改進程式碼密度的另一種方法是使用多個程式碼頁，在每個程式碼頁中定義整個指令集的不同子集，或者定義整個指令集，但在不同程式碼頁中為相同指令使用不同長度的編碼。

例如，想像一個「堆疊操作」程式碼頁，其中堆疊操作基本操作以犧牲所有其他操作為代價具有短編碼，以及一個「資料處理」程式碼頁，其中所有其他操作以犧牲堆疊操作操作為代價更短。如果堆疊操作操作傾向於一個接一個地出現，我們可以在執行任何這樣的指令後自動切換到「堆疊操作」程式碼頁。當資料處理指令出現時，我們切換回「資料處理」程式碼頁。如果下一個指令的類別取決於前一個指令的類別的條件機率與相應的無條件機率有很大不同，這種技術—自動切換到堆疊操作模式以使用較短的指令重新排列堆疊，然後切換回來—可能會大大改進程式碼密度。

5.1.7. 使用程式碼頁作為狀態和控制旗標. 在同一 TVM 修訂版內部使用多個程式碼頁的另一個潛在應用包括根據某些指令的執行結果在多個程式碼頁之間切換。

例如，想像一個使用兩個新程式碼頁 2 和 3 的 TVM 版本。大多數操作不會更改目前程式碼頁。然而，整數比較操作將在條件為假時切換到程式碼頁 2，在條件為真時切換到程式碼頁 3。此外，類似於 `EXECUTE` 的新操

[26]這是向後相容性的另一個重要機制。所有新增型別的值，以及屬於不屬於原始型別的擴展原始型別的值（例如，在上面的範例中不適合 257 位元的 513 位元整數），在舊程式碼頁中的所有指令（堆疊操作指令除外，它們自然是多態的，參見 **2.2.6**）都被視為「型別不正確的值」，並相應地產生型別檢查異常。

作?EXECUTE 在程式碼頁 3 中確實等同於 EXECUTE，但在程式碼頁 2 中將改為 DROP。這樣的技巧有效地使用目前程式碼頁的位元 0 作為狀態旗標。

或者，可以建立幾個程式碼頁—例如，4 和 5—它們僅在其單元反序列化基本操作中有所不同。例如，在程式碼頁 4 中它們可能像以前一樣工作，而在程式碼頁 5 中它們可能不是從 *Slice* 的開頭反序列化資料，而是從其末尾反序列化資料。兩個新指令—例如，CLD 和 STD—可能用於切換到程式碼頁 4 或程式碼頁 5。顯然，我們現在已經描述了一個狀態旗標，以某種新方式影響某些指令的執行。

5.1.8. 在程式碼本身中設定程式碼頁. 為了方便起見，我們在所有程式碼頁中保留一些操作碼—例如，FF n—用於指令 SETCP n，n 從 0 到 255（參見 **A.13**）。然後透過將這樣的指令插入程式（例如，TON 區塊鏈智慧合約）或函式庫函數的（主函數）的最開始，我們可以確保程式碼將始終在預期的程式碼頁中執行。

5.2 指令編碼

本節討論對所有程式碼頁和所有版本的 TVM 有效的指令編碼的一般原則。稍後，**5.3** 討論為實驗性「程式碼頁零」所做的選擇。

5.2.1. 指令由二元前綴碼編碼. TVM 程式碼頁的所有完整指令（即，指令及其所有參數，例如堆疊暫存器 $s(i)$ 的名稱或其他嵌入的常數）都由二元前綴碼 編碼。這意味著（有限）二元字串（即，位元字串）對應於每個完整指令，使得對應於不同完整指令的二元字串不重合，並且所選子集中的任何二元字串都不是此子集中另一個二元字串的前綴。

5.2.2. 從程式碼串流確定第一條指令. 作為此編碼方法的結果，任何二元字串最多允許一個前綴，它是某個完整指令的編碼。特別是，目前繼續的程式碼 cc.code（它是 *Slice*，因此是位元字串以及一些單元參照）最多允許一個這樣的前綴，它對應於 TVM 將首先執行的（唯一確定的）指令。執行後，此前綴從目前繼續的程式碼中移除，並且可以解碼下一條指令。

5.2.3. 無效操作碼. 如果 cc.code 的任何前綴都沒有在目前程式碼頁中編碼有效指令，則產生無效操作碼異常（參見 **4.5.7**）。然而，空 cc.code 的情況被單獨處理，如 **4.1.4** 中所解釋的（確切的行為可能取決於目前程式碼頁）。

5.2.4. 特殊情況：程式碼末尾填充. 作為上述規則的例外，某些程式碼頁可能接受某些 cc.code 的值，這些值太短而無法成為有效的指令編碼，作

為 NOP 的額外變體，從而有效地對它們使用與空 cc.code 相同的程序。這樣的位元字串可用於在程式碼末尾附近填充程式碼。

例如，如果在程式碼頁中使用二元字串 00000000(即，x00，參見 **1.0.3**）來編碼 NOP，則其適當的前綴不能編碼任何指令。因此，如果這是 cc.code 中剩餘的全部內容，此程式碼頁可能接受 0、00、000、…、0000000 作為 NOP 的變體，而不是產生無效操作碼異常。

例如，如果 PUSHCONT 基本操作（參見 **4.2.3**）僅建立由整數個位元組組成的程式碼的繼續，但並非所有指令都由整數個位元組編碼，則這樣的填充可能很有用。

5.2.5. TVM 程式碼是位元碼，而非位元組碼. 回想一下，TVM 是一個位元導向的機器，從某種意義上說，其 *Cell*（和 *Slice*）自然被視為位元序列，而不僅僅是八位元組（位元組）序列，參見 **3.2.5**。因為 TVM 程式碼也保存在單元中（參見 **3.1.9** 和 **4.1.4**），所以沒有理由僅使用長度可被 8 整除的位元字串作為完整指令的編碼。換句話說，一般來說，*TVM* 程式碼是位元碼，而非位元組碼。

話雖如此，某些程式碼頁（例如我們的實驗性程式碼頁零）可能選擇使用位元組碼（即，僅使用由整數個位元組組成的編碼）—要麼是為了簡單性，要麼是為了便於除錯和研究記憶體（即，單元）轉儲。[27]

5.2.6. 完整指令使用的操作碼空間. 回想一下編碼理論，二元前綴碼中使用的位元字串長度 l_i 滿足 Kraft–McMillan 不等式 $\sum_i 2^{-l_i} \leq 1$。這特別適用於 TVM 程式碼頁使用的（完整）指令編碼。我們說特定完整指令（或者更準確地說，完整指令的編碼）利用操作碼空間的 2^{-l} 部分，如果它由 l 位元字串編碼。可以看出，所有完整指令一起最多利用 1（即，「最多整個操作碼空間」）。

5.2.7. 指令或指令類別使用的操作碼空間. 上述術語擴展到指令（與其參數的所有可接受值一起考慮），甚至指令類別（例如，所有算術指令）。我們說（不完整）指令或指令類別佔用操作碼空間的 α 部分，如果 α 是屬於該類別的所有完整指令佔用的操作碼空間部分的總和。

5.2.8. 位元組碼的操作碼空間. 上述定義的有用近似如下：考慮指令編碼的第一個位元組的所有 256 個可能值。假設這些值中的 k 個對應於我們正在考慮的特定指令或指令類別。那麼此指令或指令類別佔用操作碼空間的大約 $k/256$ 部分。

這個近似顯示了為什麼所有指令不能一起佔用操作碼空間的 $256/256 = 1$ 以上的部分，至少在不損害指令解碼的唯一性的情況下。

[27]如果單元轉儲是十六進位的，則由整數個十六進位數字組成的編碼（即，長度可被四位元整除）可能同樣方便。

5.2.9. 幾乎最優編碼. 編碼理論告訴我們，在最優密集編碼中，完整指令使用的操作碼空間部分（如果完整指令以 l 位元編碼，則為 2^{-l}）應該大致等於其在實際程式中出現的機率或頻率。[28]同樣的原則應該適用於（不完整）指令或基本操作（即，沒有指定參數值的通用指令）和指令類別。

5.2.10. 範例：堆疊操作基本操作. 例如，如果堆疊操作指令在典型的 TVM 程式中大約構成所有指令的一半，則應該為編碼堆疊操作指令分配大約一半的操作碼空間。可以為這些指令保留第一個位元組（「操作碼」）`0x00`–`0x7f`。如果這些指令中的四分之一是 `XCHG`，則為 `XCHG` 保留 `0x00`–`0x1f` 是有意義的。類似地，如果所有 `XCHG` 中的一半涉及堆疊頂部 `s0`，則使用 `0x00`–`0x0f` 來編碼 `XCHG s0,s`(i) 是有意義的。

5.2.11. 指令的簡單編碼. 在大多數情況下，使用完整指令的簡單 編碼。簡單編碼以稱為指令的操作碼 的固定位元字串開始，後跟，例如，包含在指令中指定的堆疊暫存器 `s`(i) 的索引 i 的 4 位元欄位，後跟完整指令中包含的所有其他常數（字面、立即）參數。雖然簡單編碼可能不是完全最優的，但它們允許簡短的描述，並且它們的解碼和編碼可以輕鬆實作。

如果（通用）指令使用具有 l 位元操作碼的簡單編碼，則該指令將利用 2^{-l} 部分的操作碼空間。此觀察可能對 **5.2.9** 和 **5.2.10** 中描述的考慮有用。

5.2.12. 進一步優化程式碼密度：Huffman 碼. 可以為所有完整指令的集合構造最優密集二元碼，前提是已知它們在實際程式碼中的機率或頻率。這是著名的 Huffman 碼（對於給定的機率分布）。然而，這樣的碼將是高度無系統的並且難以解碼。

5.2.13. 實用指令編碼. 實際上，TVM 和其他虛擬機器中使用的指令編碼在程式碼密度和編碼和解碼的容易性之間提供了折衷。這樣的折衷可以透過為所有指令選擇簡單編碼（參見 **5.2.11**）（也許為一些經常使用的變體使用單獨的簡單編碼，例如所有 `XCHG s`(i)`,s`(j) 中的 `XCHG s0,s`(i)），並使用 **5.2.9** 和 **5.2.10** 中概述的啟發式方法為這些簡單編碼分配操作碼空間來實作；這是 TVM 目前使用的方法。

5.3 程式碼頁零中的指令編碼

本節提供關於程式碼頁零的實驗性指令編碼的詳細資訊，如本文件其他地方所述（參見附錄 **A**）並在 TVM 的初步測試版本中使用。

[28]請注意，計算的是在程式碼中出現的機率，而不是被執行的機率。在執行一百萬次的迴圈主體中出現的指令仍然只計數一次。

5.3.1. 可升級性. 首先，即使這個初步版本以某種方式進入 TON 區塊鏈的生產版本，程式碼頁機制（參見 **5.1**）使我們能夠稍後引入更好的版本，而不會損害向後相容性。[29]所以在此期間，我們可以自由實驗。

5.3.2. 指令選擇. 我們選擇在程式碼頁零中包含許多「實驗性」和並非嚴格必要的指令，只是為了看看它們在實際程式碼中可能如何使用。例如，我們既有基本（參見 **2.2.1**）又有複合（參見 **2.2.3**）堆疊操作基本操作，以及一些「無系統的」操作，如 `ROT`（主要從 Forth 借用）。如果這些基本操作很少使用，它們的包含只是浪費了操作碼空間的一部分，並使其他指令的編碼稍微不那麼有效，這是我們在 TVM 發展的這個階段可以承受的。

5.3.3. 使用實驗性指令. 這些實驗性指令中的一些已被分配了相當長的操作碼，只是為了將更多的指令放入操作碼空間。不應該僅僅因為它們很長就害怕使用它們；如果這些指令被證明是有用的，它們將在未來的修訂版中獲得更短的操作碼。程式碼頁零在這方面並不打算進行微調。

5.3.4. 選擇位元組碼. 我們選擇使用位元組碼（即，使用長度可被 8 整除的完整指令的編碼）。雖然這可能不會產生最優的程式碼密度，因為這樣的長度限制使得將用於指令編碼的操作碼空間部分與 TVM 程式碼中這些指令的估計頻率相匹配變得更加困難（參見 **5.2.11** 和 **5.2.9**），但這種方法有其優點：它允許更簡單的指令解碼器並簡化除錯（參見 **5.2.5**）。

畢竟，我們現在沒有足夠的關於不同指令的相對頻率的資料，所以我們的程式碼密度優化在這個階段可能非常近似。除錯和實驗的容易性以及實作的簡單性在這一點上更重要。

5.3.5. 所有指令的簡單編碼. 出於類似的原因，我們選擇對所有指令使用簡單編碼（參見 **5.2.11** 和 **5.2.13**），對一些非常頻繁使用的子情況使用單獨的簡單編碼，如 **5.2.13** 中所概述的。話雖如此，我們嘗試使用 **5.2.9** 和 **5.2.10** 中描述的啟發式方法分配操作碼空間。

5.3.6. 缺乏與上下文相關的編碼. 此版本的 TVM 也不使用與上下文相關的編碼（參見 **5.1.6**）。如果認為有用，它們可能會在稍後階段添加。

5.3.7. 所有指令的列表. 程式碼頁零中可用的所有指令的列表，以及它們的編碼和（在某些情況下）簡短描述，可以在附錄 **A** 中找到。

[29]請注意，啟動後的任何修改都不能單方面進行；相反，它們需要至少三分之二的驗證者的支援。

References

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

A 指令和操作碼

本附錄列出了 TVM 的（實驗性）程式碼頁零中可用的所有指令，如 **5.3** 中所解釋的。

我們按照字典序操作碼順序列出指令。然而，操作碼空間的分配方式使得每個類別中的所有指令（例如，算術基本操作）都有相鄰的操作碼。因此，我們首先列出許多堆疊操作基本操作，然後是常數基本操作、算術基本操作、比較基本操作、單元基本操作、繼續基本操作、字典基本操作，最後是應用程式特定的基本操作。

我們對位元字串使用十六進位記法（參見 **1.0**）。堆疊暫存器 $\mathtt{s}(i)$ 通常有 $0 \leq i \leq 15$，並且 i 在 4 位元欄位中編碼（或者，在一些罕見的情況下，在 8 位元欄位中）。其他立即參數通常是 4 位元、8 位元或可變長度。

在整個附錄中廣泛使用 **2.1.10** 中描述的堆疊記法。

A.1 Gas 價格

大多數基本操作的 gas 價格等於基本 *gas* 價格，計算為 $P_b := 10 + b + 5r$，其中 b 是以位元為單位的指令長度，r 是指令中包含的單元參照的數量。當指令的 gas 價格與此基本價格不同時，它在其助記符後面的括號中指示，要麼作為 *(x)*，表示總 gas 價格等於 x，要麼作為 *(+x)*，表示 $P_b + x$。除了整數常數外，可能出現以下表達式：

- C_r — 「讀取」單元（即，將單元參照轉換為單元切片）的總價格。目前等於每個單元 100 或 25 gas 單位，這取決於在 VM 的目前執行期間是否第一次「讀取」具有此雜湊的單元。
- L — 載入單元的總價格。取決於所需的載入動作。
- B_w — 建立新 *Builder* 的總價格。目前等於每個建構器 0 gas 單位。
- C_w — 從 *Builder* 建立新 *Cell* 的總價格。目前等於每個單元 500 gas 單位。

預設情況下，指令的 gas 價格等於 $P := P_b + C_r + L + B_w + C_w$。

A.2 堆疊操作基本操作

本節包括基本（參見 **2.2.1**）和複合（參見 **2.2.3**）堆疊操作基本操作，以及一些「無系統的」操作。一些複合堆疊操作基本操作，如 `XCPU` 或 `XCHG2`，被證明與等效的簡單操作序列具有相同的長度。我們仍然包含了這些基本

操作，以便它們可以在 TVM 的未來修訂版中輕鬆分配更短的操作碼—或永久刪除。

一些堆疊操作指令有兩個助記符：一個是 Forth 風格的（例如，-ROT），另一個符合識別符的通常規則（例如，ROTREV）。每當堆疊操作基本操作（例如，PICK）從堆疊接受整數參數 n 時，它必須在範圍 $0 \ldots 255$ 內；否則在任何進一步檢查之前會發生範圍檢查異常。

A.2.1. 基本堆疊操作基本操作.

- 00 — NOP，不做任何事情。
- 01 — XCHG s1，也稱為 SWAP。
- 0i — XCHG s(i) 或 XCHG s0,s(i)，交換堆疊頂部與 s(i)，$1 \leq i \leq 15$。
- 10ij — XCHG s(i),s(j)，$1 \leq i < j \leq 15$，交換 s(i) 與 s(j)。
- 11ii — XCHG s0,s(ii)，$0 \leq ii \leq 255$。
- 1i — XCHG s1,s(i)，$2 \leq i \leq 15$。
- 2i — PUSH s(i)，$0 \leq i \leq 15$，將舊 s(i) 的副本推入堆疊。
- 20 — PUSH s0，也稱為 DUP。
- 21 — PUSH s1，也稱為 OVER。
- 3i — POP $s(i)$，$0 \leq i \leq 15$，將舊堆疊頂部值彈出到舊 s(i)。
- 30 — POP s0，也稱為 DROP，丟棄堆疊頂部值。
- 31 — POP s1，也稱為 NIP。

A.2.2. 複合堆疊操作基本操作.

以下基本操作的參數 i、j 和 k 都是範圍 $0 \ldots 15$ 中的 4 位元整數。

- 4ijk — XCHG3 s(i),s(j),s(k)，等同於 XCHG s2,s(i); XCHG s1, s(j); XCHG s0,s(k)，$0 \leq i, j, k \leq 15$。
- 50ij — XCHG2 s(i),s(j)，等同於 XCHG s1,s(i); XCHG s(j)。
- 51ij — XCPU s(i),s(j)，等同於 XCHG s(i); PUSH s(j)。

- 52ij — PUXC s(i),s($j-1$)，等同於 PUSH s(i); SWAP; XCHG s(j)。
- 53ij — PUSH2 s(i),s(j)，等同於 PUSH s(i); PUSH s($j+1$)。
- 540ijk — XCHG3 s(i),s(j),s(k)（長形式）。
- 541ijk — XC2PU s(i),s(j),s(k)，等同於 XCHG2 s(i),s(j); PUSH s(k)。
- 542ijk — XCPUXC s(i),s(j),s($k-1$)，等同於 XCHG s1,s(i); PUXC s(j),s($k-1$)。
- 543ijk — XCPU2 s(i),s(j),s(k)，等同於 XCHG s(i); PUSH2 s(j), s(k)。
- 544ijk — PUXC2 s(i),s($j-1$),s($k-1$)，等同於 PUSH s(i); XCHG s2; XCHG2 s(j),s(k)。
- 545ijk — PUXCPU s(i),s($j-1$),s($k-1$)，等同於 PUXC s(i),s($j-1$); PUSH s(k)。
- 546ijk — PU2XC s(i),s($j-1$),s($k-2$)，等同於 PUSH s(i); SWAP; PUXC s(j),s($k-1$)。
- 547ijk — PUSH3 s(i),s(j),s(k)，等同於 PUSH s(i); PUSH2 s(j+1),s(k+1)。
- 54C_ — 未使用。

A.2.3. 特殊堆疊操作基本操作.

- 55ij — BLKSWAP $i+1$,$j+1$，排列兩個區塊 s($j+i+1$)…s($j+1$) 和 s(j)…s0，$0 \le i,j \le 15$。等同於 REVERSE $i+1$,$j+1$; REVERSE $j+1$,0; REVERSE $i+j+2$,0。
- 5513 — ROT2 或 2ROT（$a\ b\ c\ d\ e\ f - c\ d\ e\ f\ a\ b$），旋轉頂部三對堆疊條目。
- 550i — ROLL $i+1$，旋轉頂部 $i+1$ 個堆疊條目。等同於 BLKSWAP 1,$i+1$。
- 55i0 — ROLLREV $i+1$ 或 -ROLL $i+1$，以另一個方向旋轉頂部 $i+1$ 個堆疊條目。等同於 BLKSWAP $i+1$,1。
- 56ii — PUSH s(ii)，$0 \le ii \le 255$。

- 57ii — POP s(ii)，$0 \leq ii \leq 255$。
- 58 — ROT（$a\ b\ c - b\ c\ a$），等同於 BLKSWAP 1,2 或 XCHG2 s2,s1。
- 59 — ROTREV 或 -ROT（$a\ b\ c - c\ a\ b$），等同於 BLKSWAP 2,1 或 XCHG2 s2,s2。
- 5A — SWAP2 或 2SWAP（$a\ b\ c\ d - c\ d\ a\ b$），等同於 BLKSWAP 2,2 或 XCHG2 s3,s2。
- 5B — DROP2 或 2DROP（$a\ b -$ ），等同於 DROP; DROP。
- 5C — DUP2 或 2DUP（$a\ b - a\ b\ a\ b$），等同於 PUSH2 s1,s0。
- 5D — OVER2 或 2OVER（$a\ b\ c\ d - a\ b\ c\ d\ a\ b$），等同於 PUSH2 s3,s2。
- 5Eij — REVERSE i+2,j，反轉 s(j+i+1)…s(j) 的順序，$0 \leq i, j \leq 15$；等同於 $\lfloor i/2 \rfloor + 1$ 個 XCHG 的序列。
- 5F0i — BLKDROP i，等同於執行 i 次 DROP。
- 5Fij — BLKPUSH i,j，等同於執行 i 次 PUSH s(j)，$1 \leq i \leq 15$，$0 \leq j \leq 15$。
- 60 — PICK 或 PUSHX，從堆疊彈出整數 i，然後執行 PUSH s(i)。
- 61 — ROLLX，從堆疊彈出整數 i，然後執行 BLKSWAP 1,i。
- 62 — -ROLLX 或 ROLLREVX，從堆疊彈出整數 i，然後執行 BLKSWAP i,1。
- 63 — BLKSWX，從堆疊彈出整數 i、j，然後執行 BLKSWAP i,j。
- 64 — REVX，從堆疊彈出整數 i、j，然後執行 REVERSE i,j。
- 65 — DROPX，從堆疊彈出整數 i，然後執行 BLKDROP i。
- 66 — TUCK（$a\ b$ - $b\ a\ b$），等同於 SWAP; OVER 或 XCPU s1,s1。
- 67 — XCHGX，從堆疊彈出整數 i，然後執行 XCHG s(i)。
- 68 — DEPTH，推入堆疊的目前深度。

- 69 — CHKDEPTH，從堆疊彈出整數 i，然後檢查是否至少有 i 個元素，否則產生堆疊下溢異常。
- 6A — ONLYTOPX，從堆疊彈出整數 i，然後移除除頂部 i 個元素之外的所有元素。
- 6B — ONLYX，從堆疊彈出整數 i，然後只保留底部 i 個元素。大致等同於 DEPTH; SWAP; SUB; DROPX。
- 6C00–6C0F — 保留用於堆疊操作。
- 6Cij — BLKDROP2 i,j，丟棄頂部 j 個元素下的 i 個堆疊元素，其中 $1 \leq i \leq 15$ 和 $0 \leq j \leq 15$。等同於 REVERSE $i+j$,0; BLKDROP i; REVERSE j,0。

A.3 元組、列表和 Null 基本操作

Tuple 是由最多 255 個任意型別（不一定相同）的 TVM 堆疊值組成的有序集合。元組基本操作建立、修改和解包 *Tuple*；它們在此過程中操作任意型別的值，類似於堆疊基本操作。我們不建議使用超過 15 個元素的 *Tuple*。

當 *Tuple* t 包含元素 x_1、…、x_n(按此順序)時，我們寫作 $t = (x_1, \ldots, x_n)$；數字 $n \geq 0$ 是 *Tuple* t 的長度。它也表示為 $|t|$。長度為 2 的 *Tuple* 稱為對，長度為 3 的 *Tuple* 稱為三元組。

Lisp 風格的列表借助對（即，恰好由兩個元素組成的元組）來表示。空列表由 *Null* 值表示，非空列表由對 (h, t) 表示，其中 h 是列表的第一個元素，t 是其尾部。

A.3.1. *Null* 基本操作. 以下基本操作使用型別 *Null* 的（唯一）值 $\perp$，對於表示空列表、二元樹的空分支以及 *Maybe X* 型別中值的缺失很有用。由 NIL 建立的空 *Tuple* 本來可以用於相同的目的；然而，*Null* 更有效率且消耗更少的 gas。

- 6D — NULL 或 PUSHNULL（– $\perp$），推入型別 *Null* 的唯一值。
- 6E — ISNULL（x – ?），檢查 x 是否為 *Null*，並相應地傳回 -1 或 0。

A.3.2. 元組基本操作.

- 6F0n — TUPLE n（x_1 …x_n – t），建立包含 n 個值 x_1、…、x_n 的新 *Tuple* $t = (x_1, \ldots, x_n)$，其中 $0 \leq n \leq 15$。

- 6F00 — NIL（– t），推入長度為零的唯一 *Tuple* $t = ()$。
- 6F01 — SINGLE（x – t），建立單例 $t := (x)$，即，長度為 1 的 *Tuple*。
- 6F02 — PAIR 或 CONS（x y – t），建立對 $t := (x, y)$。
- 6F03 — TRIPLE（x y z – t），建立三元組 $t := (x, y, z)$。
- 6F1k — INDEX k（t – x），傳回 *Tuple* t 的第 k 個元素，其中 $0 \leq k \leq 15$。換句話說，如果 $t = (x_1, \ldots, x_n)$，則傳回 x_{k+1}。如果 $k \geq n$，丟擲範圍檢查異常。
- 6F10 — FIRST 或 CAR（t – x），傳回 *Tuple* 的第一個元素。
- 6F11 — SECOND 或 CDR（t – y），傳回 *Tuple* 的第二個元素。
- 6F12 — THIRD（t – z），傳回 *Tuple* 的第三個元素。
- 6F2n — UNTUPLE n（t – x_1 …x_n），解包長度等於 $0 \leq n \leq 15$ 的 *Tuple* $t = (x_1, \ldots, x_n)$。如果 t 不是 *Tuple*，或如果 $|t| \neq n$，則丟擲型別檢查異常。
- 6F21 — UNSINGLE（t – x），解包單例 $t = (x)$。
- 6F22 — UNPAIR 或 UNCONS（t – x y），解包對 $t = (x, y)$。
- 6F23 — UNTRIPLE（t – x y z），解包三元組 $t = (x, y, z)$。
- 6F3k — UNPACKFIRST k（t – x_1 …x_k），解包 *Tuple* t 的前 $0 \leq k \leq 15$ 個元素。如果 $|t| < k$，丟擲型別檢查異常。
- 6F30 — CHKTUPLE（t – ），檢查 t 是否為 *Tuple*。
- 6F4n — EXPLODE n（t – x_1 …x_m m），解包 *Tuple* $t = (x_1, \ldots, x_m)$ 並傳回其長度 m，但僅當 $m \leq n \leq 15$ 時。否則丟擲型別檢查異常。
- 6F5k — SETINDEX k（t x – t'），計算僅在位置 t'_{k+1} 處與 t 不同的 *Tuple* t'，該位置設定為 x。換句話說，$|t'| = |t|$，對於 $i \neq k + 1$，$t'_i = t_i$，以及對於給定的 $0 \leq k \leq 15$，$t'_{k+1} = x$。如果 $k \geq |t|$，丟擲範圍檢查異常。
- 6F50 — SETFIRST（t x – t'），將 *Tuple* t 的第一個元件設定為 x 並傳回產生的 *Tuple* t'。

- 6F51 — SETSECOND（$t\ x - t'$），將 *Tuple* t 的第二個元件設定為 x 並傳回產生的 *Tuple* t'。
- 6F52 — SETTHIRD（$t\ x - t'$），將 *Tuple* t 的第三個元件設定為 x 並傳回產生的 *Tuple* t'。
- 6F6k — INDEXQ k（$t - x$），傳回 *Tuple* t 的第 k 個元素，其中 $0 \leq k \leq 15$。換句話說，如果 $t = (x_1, \ldots, x_n)$，則傳回 x_{k+1}。如果 $k \geq n$，或如果 t 為 *Null*，則傳回 *Null* 而非 x。
- 6F7k — SETINDEXQ k（$t\ x - t'$），將 *Tuple* t 的第 k 個元件設定為 x，其中 $0 \leq k < 16$，並傳回產生的 *Tuple* t'。如果 $|t| \leq k$，首先透過將所有新元件設定為 *Null* 將原始 *Tuple* 擴展到長度 $k + 1$。如果 t 的原始值為 *Null*，則將其視為空 *Tuple*。如果 t 不是 *Null* 或 *Tuple*，丟擲異常。如果 x 為 *Null* 且 $|t| \leq k$ 或 t 為 *Null*，則總是傳回 $t' = t$（並且不消耗元組建立 gas）。
- 6F80 — TUPLEVAR（$x_1 \ldots x_n\ n - t$），類似於 TUPLE 建立長度為 n 的新 *Tuple* t，但 $0 \leq n \leq 255$ 從堆疊取得。
- 6F81 — INDEXVAR（$t\ k - x$），類似於 INDEX k，但 $0 \leq k \leq 254$ 從堆疊取得。
- 6F82 — UNTUPLEVAR（$t\ n - x_1 \ldots x_n$），類似於 UNTUPLE n，但 $0 \leq n \leq 255$ 從堆疊取得。
- 6F83 — UNPACKFIRSTVAR（$t\ n - x_1 \ldots x_n$），類似於 UNPACKFIRST n，但 $0 \leq n \leq 255$ 從堆疊取得。
- 6F84 — EXPLODEVAR（$t\ n - x_1 \ldots x_m\ m$），類似於 EXPLODE n，但 $0 \leq n \leq 255$ 從堆疊取得。
- 6F85 — SETINDEXVAR（$t\ x\ k - t'$），類似於 SETINDEX k，但 $0 \leq k \leq 254$ 從堆疊取得。
- 6F86 — INDEXVARQ（$t\ k - x$），類似於 INDEXQ n，但 $0 \leq k \leq 254$ 從堆疊取得。
- 6F87 — SETINDEXVARQ（$t\ x\ k - t'$），類似於 SETINDEXQ k，但 $0 \leq k \leq 254$ 從堆疊取得。
- 6F88 — TLEN（$t - n$），傳回 *Tuple* 的長度。

- 6F89 — QTLEN（$t - n$ 或 -1），類似於 TLEN，但如果 t 不是 *Tuple*，則傳回 -1。
- 6F8A — ISTUPLE（$t - ?$），根據 t 是否為 *Tuple* 傳回 -1 或 0。
- 6F8B — LAST（$t - x$），傳回非空 *Tuple* t 的最後一個元素 $t_{|t|}$。
- 6F8C — TPUSH 或 COMMA（$t\ x - t'$），將值 x 附加到 *Tuple* $t = (x_1, \ldots, x_n)$，但僅當產生的 *Tuple* $t' = (x_1, \ldots, x_n, x)$ 的長度最多為 255 時。否則丟擲型別檢查異常。
- 6F8D — TPOP（$t - t'\ x$），從非空 *Tuple* $t = (x_1, \ldots, x_n)$ 分離最後一個元素 $x = x_n$，並傳回產生的 *Tuple* $t' = (x_1, \ldots, x_{n-1})$ 和原始最後一個元素 x。
- 6FA0 — NULLSWAPIF（$x - x$ 或 $\perp\ x$），在頂部 *Integer* x 下推入 *Null*，但僅當 $x \neq 0$ 時。
- 6FA1 — NULLSWAPIFNOT（$x - x$ 或 $\perp\ x$），在頂部 *Integer* x 下推入 *Null*，但僅當 $x = 0$ 時。可用於在靜默基本操作（如 PLDUXQ）之後進行堆疊對齊。
- 6FA2 — NULLROTRIF（$x\ y - x\ y$ 或 $\perp\ x\ y$），在從頂部算起的第二個堆疊條目下推入 *Null*，但僅當頂部 *Integer* y 非零時。
- 6FA3 — NULLROTRIFNOT（$x\ y - x\ y$ 或 $\perp\ x\ y$），在從頂部算起的第二個堆疊條目下推入 *Null*，但僅當頂部 *Integer* y 為零時。可用於在靜默基本操作（如 LDUXQ）之後進行堆疊對齊。
- 6FA4 — NULLSWAPIF2（$x - x$ 或 $\perp\ \perp\ x$），在頂部 *Integer* x 下推入兩個 *Null*，但僅當 $x \neq 0$ 時。等同於 NULLSWAPIF; NULLSWAPIF。
- 6FA5 — NULLSWAPIFNOT2（$x - x$ 或 $\perp\ \perp\ x$），在頂部 *Integer* x 下推入兩個 *Null*，但僅當 $x = 0$ 時。等同於 NULLSWAPIFNOT; NULLSWAPIFNOT。
- 6FA6 — NULLROTRIF2（$x\ y - x\ y$ 或 $\perp\ \perp\ x\ y$），在從頂部算起的第二個堆疊條目下推入兩個 *Null*，但僅當頂部 *Integer* y 非零時。等同於 NULLROTRIF; NULLROTRIF。
- 6FA7 — NULLROTRIFNOT2（$x\ y - x\ y$ 或 $\perp\ \perp\ x\ y$），在從頂部算起的第二個堆疊條目下推入兩個 *Null*，但僅當頂部 *Integer* y 為零時。等同於 NULLROTRIFNOT; NULLROTRIFNOT。

- 6FBij — INDEX2 i,j（$t - x$），對於 $0 \le i, j \le 3$ 恢復 $x = (t_{i+1})_{j+1}$。等同於 INDEX i; INDEX j。
- 6FB4 — CADR（$t - x$），恢復 $x = (t_2)_1$。
- 6FB5 — CDDR（$t - x$），恢復 $x = (t_2)_2$。
- 6FE_ijk — INDEX3 i,j,k（$t - x$），對於 $0 \le i, j, k \le 3$ 恢復 $x = \big((t_{i+1})_{j+1}\big)_{k+1}$。等同於 INDEX2 i,j; INDEX k。
- 6FD4 — CADDR（$t - x$），恢復 $x = \big((t_2)_2\big)_1$。
- 6FD5 — CDDDR（$t - x$），恢復 $x = \big((t_2)_2\big)_2$。

A.4 常數或字面基本操作

以下基本操作將某種型別和範圍的一個字面（或未命名常數）推入堆疊，該字面作為指令的一部分（立即引數）儲存。因此，如果立即引數缺失或太短，則丟擲「無效或太短的操作碼」異常（代碼 6）。

A.4.1. 整數和布林常數.

- 7i — PUSHINT x，$-5 \le x \le 10$，將整數 x 推入堆疊；這裡 i 等於 x 的低階四位元（即，$i = x \bmod 16$）。
- 70 — ZERO、FALSE 或 PUSHINT 0，推入零。
- 71 — ONE 或 PUSHINT 1。
- 72 — TWO 或 PUSHINT 2。
- 7A — TEN 或 PUSHINT 10。
- 7F — TRUE 或 PUSHINT -1。
- 80xx — PUSHINT xx，$-128 \le xx \le 127$。
- 81$xxxx$ — PUSHINT $xxxx$，$-2^{15} \le xxxx < 2^{15}$ 一個有符號 16 位元大端整數。
- 81FC18 — PUSHINT -1000。

- 82$lxxx$ — PUSHINT xxx，其中 5 位元 $0 \leq l \leq 30$ 決定有符號大端整數 xxx 的長度 $n = 8l + 19$。此指令的總長度為 $l + 4$ 位元組或 $n + 13 = 8l + 32$ 位元。
- 821005F5E100 — PUSHINT 10^8。
- 83xx — PUSHPOW2 $xx + 1$，（靜默地）推入 2^{xx+1}，$0 \leq xx \leq 255$。
- 83FF — PUSHNAN，推入 NaN。
- 84xx — PUSHPOW2DEC $xx + 1$，推入 $2^{xx+1} - 1$，$0 \leq xx \leq 255$。
- 85xx — PUSHNEGPOW2 $xx + 1$，推入 -2^{xx+1}，$0 \leq xx \leq 255$。
- 86、87 — 保留用於整數常數。

A.4.2. 常數切片、繼續、單元和參照. 下面列出的大多數指令推入字面切片、繼續、單元和單元參照，作為指令的立即引數儲存。因此，如果立即引數缺失或太短，則丟擲「無效或太短的操作碼」異常（代碼 6）。

- 88 — PUSHREF，將 cc.code 的第一個參照作為 *Cell* 推入堆疊（並從目前繼續中移除此參照）。
- 89 — PUSHREFSLICE，類似於 PUSHREF，但將單元轉換為 *Slice*。
- 8A — PUSHREFCONT，類似於 PUSHREFSLICE，但從單元製作簡單的普通 *Continuation*。
- 8Bxsss — PUSHSLICE sss，推入 cc.code 的（前綴）子切片，由其前 $8x + 4$ 位元和沒有參照（即，本質上是位元字串）組成，其中 $0 \leq x \leq 15$。假設有完成標記，這意味著所有尾隨零和最後一個二元一（如果存在）從此位元字串中移除。如果原始位元字串僅由零組成，將推入空切片。
- 8B08 — PUSHSLICE x8_，推入空切片（位元字串 `''）。
- 8B04 — PUSHSLICE x4_，推入位元字串 `0'。
- 8B0C — PUSHSLICE xC_，推入位元字串 `1'。
- 8Crxxssss — PUSHSLICE ssss，推入 cc.code 的（前綴）子切片，由其前 $1 \leq r + 1 \leq 4$ 個參照和最多前 $8xx + 1$ 位元資料組成，$0 \leq xx \leq 31$。也假設有完成標記。

- 8C01 等同於 PUSHREFSLICE。
- 8Drxxssssss — PUSHSLICE sssss，推入 cc.code 的子切片，由 $0 \leq r \leq 4$ 個參照和最多 $8xx + 6$ 位元資料組成，$0 \leq xx \leq 127$。假設有完成標記。
- 8DE_ — 未使用（保留）。
- 8F_rxxcccc — PUSHCONT cccc，其中 $cccc$ 是由 cc.code 的前 $0 \leq r \leq 3$ 個參照和前 $0 \leq xx \leq 127$ 個位元組製作的簡單普通繼續。
- 9xccc — PUSHCONT ccc，對於 $0 \leq x \leq 15$ 推入 x 位元組繼續。

A.5 算術基本操作

A.5.1. 加法、減法、乘法.

- A0 — ADD（$x\ y - x + y$），將兩個整數相加。
- A1 — SUB（$x\ y - x - y$）。
- A2 — SUBR（$x\ y - y - x$），等同於 SWAP; SUB。
- A3 — NEGATE（$x - -x$），等同於 MULCONST -1 或 ZERO; SUBR。請注意，如果 $x = -2^{256}$，它會觸發整數溢位異常。
- A4 — INC（$x - x + 1$），等同於 ADDCONST 1。
- A5 — DEC（$x - x - 1$），等同於 ADDCONST -1。
- A6cc — ADDCONST cc（$x - x + cc$），$-128 \leq cc \leq 127$。
- A7cc — MULCONST cc（$x - x \cdot cc$），$-128 \leq cc \leq 127$。
- A8 — MUL（$x\ y - xy$）。

A.5.2. 除法.

DIV、DIVMOD 或 MOD 操作的一般編碼是 A9$mscdf$，具有可選的預乘法和可選的除法或乘法替換為位移。變數一或二位元欄位 m、s、c、d 和 f 如下：

- $0 \leq m \leq 1$ — 指示是否有預乘法（MULDIV 操作及其變體），可能被左移替換。

- $0 \le s \le 2$ — 指示乘法或除法是否已被位移替換：$s = 0$—無替換，$s = 1$—除法被右移替換，$s = 2$—乘法被左移替換（僅對 $m = 1$ 可能）。
- $0 \le c \le 1$ — 指示位移運算子是否有常數單位元組引數 tt（如果 $s \neq 0$）。對於 $s = 0$，$c = 0$。如果 $c = 1$，則 $0 \le tt \le 255$，位移執行 $tt + 1$ 位元。如果 $s \neq 0$ 且 $c = 0$，則位移量作為範圍 $0 \ldots 256$ 中的堆疊頂部 *Integer* 提供給指令。
- $1 \le d \le 3$ — 指示需要哪些除法結果：1—僅商，2—僅餘數，3—兩者。
- $0 \le f \le 2$ — 捨入模式：0—下限，1—最接近整數，2—上限（參見 **1.5.6**）。

範例：

- 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（$x\ y - r$），其中 $q := \lfloor x/y \rfloor$，$r := x \bmod y := x - yq$。
- A90C — DIVMOD（$x\ y - q\ r$），其中 $q := \lfloor x/y \rfloor$，$r := x - yq$。
- A90D — DIVMODR（$x\ y - q'\ r'$），其中 $q' := \lfloor x/y + 1/2 \rfloor$，$r' := x - yq'$。
- A90E — DIVMODC（$x\ y - q''\ r''$），其中 $q'' := \lceil x/y \rceil$，$r'' := x - yq''$。
- A924 — 與 RSHIFT 相同：（$x\ y - \lfloor x \cdot 2^{-y} \rfloor$），對於 $0 \le y \le 256$。
- A934tt — 與 RSHIFT $tt + 1$ 相同：（$x - \lfloor x \cdot 2^{-tt-1} \rfloor$）。
- A938tt — MODPOW2 $tt + 1$：（$x - x \bmod 2^{tt+1}$）。
- A985 — MULDIVR（$x\ y\ z - q'$），其中 $q' = \lfloor xy/z + 1/2 \rfloor$。
- A988 — MULMOD（$x\ y\ z - r$），其中 $r = xy \bmod z = xy - qz$，$q = \lfloor xy/z \rfloor$。對於 $z \neq 0$，此操作總是成功並傳回正確的 r 值，即使中間結果 xy 或商 q 不適合 257 位元。

- A98C — MULDIVMOD ($x\ y\ z - q\ r$)，其中 $q := \lfloor x \cdot y/z \rfloor$，$r := x \cdot y \bmod z$ (與 Forth 中的 */MOD 相同)。
- A9A4 — MULRSHIFT ($x\ y\ z - \lfloor xy \cdot 2^{-z} \rfloor$)，對於 $0 \le z \le 256$。
- A9A5 — MULRSHIFTR ($x\ y\ z - \lfloor xy \cdot 2^{-z} + 1/2 \rfloor$)，對於 $0 \le z \le 256$。
- A9B4tt — MULRSHIFT $tt+1$ ($x\ y - \lfloor xy \cdot 2^{-tt-1} \rfloor$)。
- A9B5tt — MULRSHIFTR $tt+1$ ($x\ y - \lfloor xy \cdot 2^{-tt-1} + 1/2 \rfloor$)。
- A9C4 — LSHIFTDIV ($x\ y\ z - \lfloor 2^z x/y \rfloor$)，對於 $0 \le z \le 256$。
- A9C5 — LSHIFTDIVR ($x\ y\ z - \lfloor 2^z x/y + 1/2 \rfloor$)，對於 $0 \le z \le 256$。
- A9D4tt — LSHIFTDIV $tt+1$ ($x\ y - \lfloor 2^{tt+1} x/y \rfloor$)。
- A9D5tt — LSHIFTDIVR $tt+1$ ($x\ y - \lfloor 2^{tt+1} x/y + 1/2 \rfloor$)。

這些操作中最有用的是 DIV、DIVMOD、MOD、DIVR、DIVC、MODPOW2 t 和 RSHIFTR t(用於整數算術)；以及 MULDIVMOD、MULDIV、MULDIVR、LSHIFTDIVR t 和 MULRSHIFTR t (用於定點算術)。

A.5.3. 位移、邏輯運算.

- AAcc — LSHIFT $cc+1$ ($x - x \cdot 2^{cc+1}$)，$0 \le cc \le 255$。
- AA00 — LSHIFT 1，等同於 MULCONST 2 或 Forth 的 2*。
- ABcc — RSHIFT $cc+1$ ($x - \lfloor x \cdot 2^{-cc-1} \rfloor$)，$0 \le cc \le 255$。
- AC — LSHIFT ($x\ y - x \cdot 2^y$)，$0 \le y \le 1023$。
- AD — RSHIFT ($x\ y - \lfloor x \cdot 2^{-y} \rfloor$)，$0 \le y \le 1023$。
- AE — POW2 ($y - 2^y$)，$0 \le y \le 1023$，等同於 ONE; SWAP; LSHIFT。
- AF — 保留。
- B0 — AND ($x\ y - x \& y$)，兩個有符號整數 x 和 y 的位元「和」，符號擴展到無窮大。
- B1 — OR ($x\ y - x \vee y$)，兩個整數的位元「或」。

- B2 — XOR（$x\ y - x \oplus y$），兩個整數的位元「異或」。
- B3 — NOT（$x - x \oplus -1 = -1 - x$），整數的位元「非」。
- B4cc — FITS $cc+1$（$x - x$），對於 $0 \leq cc \leq 255$ 檢查 x 是否為 $cc+1$ 位元有符號整數（即，是否 $-2^{cc} \leq x < 2^{cc}$）。如果不是，則觸發整數溢位異常，或將 x 替換為 NaN（靜默版本）。
- B400 — FITS 1 或 CHKBOOL（$x - x$），檢查 x 是否為「布林值」（即，0 或 -1）。
- B5cc — UFITS $cc+1$（$x - x$），對於 $0 \leq cc \leq 255$ 檢查 x 是否為 $cc+1$ 位元無符號整數（即，是否 $0 \leq x < 2^{cc+1}$）。
- B500 — UFITS 1 或 CHKBIT，檢查 x 是否為二元數字（即，零或一）。
- B600 — FITSX（$x\ c - x$），對於 $0 \leq c \leq 1023$ 檢查 x 是否為 c 位元有符號整數。
- B601 — UFITSX（$x\ c - x$），對於 $0 \leq c \leq 1023$ 檢查 x 是否為 c 位元無符號整數。
- B602 — BITSIZE（$x - c$），計算最小 $c \geq 0$ 使得 x 適合 c 位元有符號整數（$-2^{c-1} \leq c < 2^{c-1}$）。
- B603 — UBITSIZE（$x - c$），計算最小 $c \geq 0$ 使得 x 適合 c 位元無符號整數（$0 \leq x < 2^c$），或丟擲範圍檢查異常。
- B608 — MIN（$x\ y - x$ 或 y），計算兩個整數 x 和 y 的最小值。
- B609 — MAX（$x\ y - x$ 或 y），計算兩個整數 x 和 y 的最大值。
- B60A — MINMAX 或 INTSORT2（$x\ y - x\ y$ 或 $y\ x$），排序兩個整數。如果任何引數為 NaN，此操作的靜默版本傳回兩個 NaN。
- B60B — ABS（$x - |x|$），計算整數 x 的絕對值。

A.5.4. 靜默算術基本操作. 我們選擇預設使所有算術操作「非靜默」（信號），並透過前綴建立它們的靜默對應。這樣的編碼肯定是次優的。尚不清楚是否應該以這種方式進行，還是透過預設使所有算術操作靜默來進行相反的方式，或者靜默和非靜默操作是否應該給予相同長度的操作碼；這只能透過實踐來解決。

- B7xx — QUIET 前綴，將任何算術操作轉換為其「靜默」變體，透過在其助記符前加上 Q 來指示。如果結果不適合 *Integer*，或如果其引數之一是 NaN，這些操作傳回 NaN 而不是丟擲整數溢位異常。請注意，這不擴展到必須在小範圍內（例如，0–1023）的位移量和其他參數。還要注意，如果提供了 *Integer* 以外型別的值，這不會停用型別檢查異常。

- B7A0 — QADD（$x\ y - x + y$），如果 x 和 y 是 *Integer*，則總是有效，但如果無法執行加法，則傳回 NaN。

- B7A8 — QMUL（$x\ y - xy$），如果 $-2^{256} \leq xy < 2^{256}$，則傳回 x 和 y 的乘積。否則傳回 NaN，即使 $x = 0$ 且 y 是 NaN。

- B7A904 — QDIV（$x\ y - \lfloor x/y \rfloor$），如果 $y = 0$，或如果 $y = -1$ 且 $x = -2^{256}$，或如果 x 或 y 中的任何一個是 NaN，則傳回 NaN。

- B7A98C — QMULDIVMOD（$x\ y\ z - q\ r$），其中 $q := \lfloor x \cdot y/z \rfloor$，$r := x \cdot y \bmod z$。如果 $z = 0$，或如果 x、y 或 z 中至少有一個是 NaN，則 q 和 r 都設定為 NaN。否則總是傳回 r 的正確值，但如果 $q < -2^{256}$ 或 $q \geq 2^{256}$，則 q 被替換為 NaN。

- B7B0 — QAND（$x\ y - x\&y$），位元「和」（類似於 AND），但如果 x 或 y 中的任何一個是 NaN，則傳回 NaN，而不是丟擲整數溢位異常。然而，如果引數之一為零，另一個為 NaN，則結果為零。

- B7B1 — QOR（$x\ y - x \vee y$），位元「或」。如果 $x = -1$ 或 $y = -1$，結果總是 -1，即使另一個引數是 NaN。

- B7B507 — QUFITS 8（$x - x'$），檢查 x 是否為無符號位元組（即，是否 $0 \leq x < 2^8$），如果不是這種情況，則將 x 替換為 NaN；否則保持 x 不變（即，如果 x 是無符號位元組或 NaN）。

A.6 比較基本操作

A.6.1. 整數比較. 所有整數比較基本操作傳回整數 -1（「真」）或 0（「假」）以指示比較的結果。我們不定義它們的「布林電路」對應，它們將根據比較的結果將控制轉移到 c0 或 c1。如果需要，可以借助 RETBOOL 模擬這些指令。

整數比較基本操作的靜默版本也可用，使用 QUIET 前綴（B7）編碼。如果正在比較的任何整數是 NaN，則靜默比較的結果也將是 NaN（「未定義」），而不是 -1（「是」）或 0（「否」），從而有效地支援三元邏輯。

- B8 — SGN（x – $\mathrm{sgn}(x)$），計算整數 x 的符號：如果 $x < 0$ 則為 -1，如果 $x = 0$ 則為 0，如果 $x > 0$ 則為 1。
- B9 — LESS（$x\ y$ – $x < y$），如果 $x < y$ 則傳回 -1，否則傳回 0。
- BA — EQUAL（$x\ y$ – $x = y$），如果 $x = y$ 則傳回 -1，否則傳回 0。
- BB — LEQ（$x\ y$ – $x \leq y$）。
- BC — GREATER（$x\ y$ – $x > y$）。
- BD — NEQ（$x\ y$ – $x \neq y$），等同於 EQUAL; NOT。
- BE — GEQ（$x\ y$ – $x \geq y$），等同於 LESS; NOT。
- BF — CMP（$x\ y$ – $\mathrm{sgn}(x - y)$），計算 $x - y$ 的符號：如果 $x < y$ 則為 -1，如果 $x = y$ 則為 0，如果 $x > y$ 則為 1。除非 x 或 y 是 NaN，否則這裡不會發生整數溢位。
- C0yy — EQINT yy（x – $x = yy$），對於 $-2^7 \leq yy < 2^7$。
- C000 — ISZERO，檢查整數是否為零。對應於 Forth 的 0=。
- C1yy — LESSINT yy（x – $x < yy$），對於 $-2^7 \leq yy < 2^7$。
- C100 — ISNEG，檢查整數是否為負數。對應於 Forth 的 0<。
- C101 — ISNPOS，檢查整數是否非正。
- C2yy — GTINT yy（x – $x > yy$），對於 $-2^7 \leq yy < 2^7$。
- C200 — ISPOS，檢查整數是否為正數。對應於 Forth 的 0>。
- C2FF — ISNNEG，檢查整數是否非負。
- C3yy — NEQINT yy（x – $x \neq yy$），對於 $-2^7 \leq yy < 2^7$。
- C4 — ISNAN（x – $x = \mathrm{NaN}$），檢查 x 是否為 NaN。
- C5 — CHKNAN（x – x），如果 x 是 NaN，則丟擲算術溢位異常。
- C6 — 保留用於整數比較。

A.6.2. 其他比較.

這些「其他比較」基本操作中的大多數實際上將 *Slice* 的資料部分作為位元字串進行比較。

- C700 — SEMPTY（$s - s = \emptyset$），檢查 *Slice* s 是否為空（即，不包含資料位元和單元參照）。
- C701 — SDEMPTY（$s - s \approx \emptyset$），檢查 *Slice* s 是否沒有資料位元。
- C702 — SREMPTY（$s - r(s) = 0$），檢查 *Slice* s 是否沒有參照。
- C703 — SDFIRST（$s - s_0 = 1$），檢查 *Slice* s 的第一個位元是否為一。
- C704 — SDLEXCMP（$s\ s' - c$），以字典序比較 s 的資料與 s' 的資料，根據結果傳回 -1、0 或 1。
- C705 — SDEQ（$s\ s' - s \approx s'$），檢查 s 和 s' 的資料部分是否一致，等同於 SDLEXCMP; ISZERO。
- C708 — SDPFX（$s\ s' - ?$），檢查 s 是否為 s' 的前綴。
- C709 — SDPFXREV（$s\ s' - ?$），檢查 s' 是否為 s 的前綴，等同於 SWAP; SDPFX。
- C70A — SDPPFX（$s\ s' - ?$），檢查 s 是否為 s' 的真前綴（即，與 s' 不同的前綴）。
- C70B — SDPPFXREV（$s\ s' - ?$），檢查 s' 是否為 s 的真前綴。
- C70C — SDSFX（$s\ s' - ?$），檢查 s 是否為 s' 的後綴。
- C70D — SDSFXREV（$s\ s' - ?$），檢查 s' 是否為 s 的後綴。
- C70E — SDPSFX（$s\ s' - ?$），檢查 s 是否為 s' 的真後綴。
- C70F — SDPSFXREV（$s\ s' - ?$），檢查 s' 是否為 s 的真後綴。
- C710 — SDCNTLEAD0（$s - n$），傳回 s 中前導零的數量。
- C711 — SDCNTLEAD1（$s - n$），傳回 s 中前導一的數量。
- C712 — SDCNTTRAIL0（$s - n$），傳回 s 中尾隨零的數量。
- C713 — SDCNTTRAIL1（$s - n$），傳回 s 中尾隨一的數量。

A.7 單元基本操作

單元基本操作主要是單元序列化基本操作（使用 *Builder*）或單元反序列化基本操作（使用 *Slice*）。

A.7.1. 單元序列化基本操作. 所有這些基本操作首先檢查 Builder 中是否有足夠的空間，然後才檢查被序列化的值的範圍。

- C8 — NEWC（$-\ b$），建立一個新的空 *Builder*。
- C9 — ENDC（$b\ -\ c$），將 *Builder* 轉換為普通 *Cell*。
- CAcc — STI $cc+1$（$x\ b\ -\ b'$），對於 $0 \le cc \le 255$，將有符號 $cc+1$ 位元整數 x 儲存到 *Builder* b 中，如果 x 不適合 $cc+1$ 位元，則丟擲範圍檢查異常。
- CBcc — STU $cc+1$（$x\ b\ -\ b'$），將無符號 $cc+1$ 位元整數 x 儲存到 *Builder* b 中。在所有其他方面，它類似於 STI。
- CC — STREF（$c\ b\ -\ b'$），將對 *Cell* c 的參照儲存到 *Builder* b 中。
- CD — STBREFR 或 ENDCST（$b\ b''\ -\ b$），等同於 ENDC; SWAP; STREF。
- CE — STSLICE（$s\ b\ -\ b'$），將 *Slice* s 儲存到 *Builder* b 中。
- CF00 — STIX（$x\ b\ l\ -\ b'$），對於 $0 \le l \le 257$，將有符號 l 位元整數 x 儲存到 b 中。
- CF01 — STUX（$x\ b\ l\ -\ b'$），對於 $0 \le l \le 256$，將無符號 l 位元整數 x 儲存到 b 中。
- CF02 — STIXR（$b\ x\ l\ -\ b'$），類似於 STIX，但引數順序不同。
- CF03 — STUXR（$b\ x\ l\ -\ b'$），類似於 STUX，但引數順序不同。
- CF04 — STIXQ（$x\ b\ l\ -\ x\ b\ f$ 或 $b'\ 0$），STIX 的靜默版本。如果 b 中沒有空間，則設定 $b'=b$ 和 $f=-1$。如果 x 不適合 l 位元，則設定 $b'=b$ 和 $f=1$。如果操作成功，b' 是新的 *Builder*，$f=0$。然而，$0 \le l \le 257$，如果不是這樣，則產生範圍檢查異常。
- CF05 — STUXQ（$x\ b\ l\ -\ b'\ f$）。
- CF06 — STIXRQ（$b\ x\ l\ -\ b\ x\ f$ 或 $b'\ 0$）。

- CF07 — STUXRQ（$b\ x\ l - b\ x\ f$ 或 $b'\ 0$）。
- CF08cc — STI $cc+1$ 的較長版本。
- CF09cc — STU $cc+1$ 的較長版本。
- CF0Acc — STIR $cc+1$（$b\ x - b'$），等同於 SWAP; STI $cc+1$。
- CF0Bcc — STUR $cc+1$（$b\ x - b'$），等同於 SWAP; STU $cc+1$。
- CF0Ccc — STIQ $cc+1$（$x\ b - x\ b\ f$ 或 $b'\ 0$）。
- CF0Dcc — STUQ $cc+1$（$x\ b - x\ b\ f$ 或 $b'\ 0$）。
- CF0Ecc — STIRQ $cc+1$（$b\ x - b\ x\ f$ 或 $b'\ 0$）。
- CF0Fcc — STURQ $cc+1$（$b\ x - b\ x\ f$ 或 $b'\ 0$）。
- CF10 — STREF（$c\ b - b'$）的較長版本。
- CF11 — STBREF（$b'\ b - b''$），等同於 SWAP; STBREFREV。
- CF12 — STSLICE（$s\ b - b'$）的較長版本。
- CF13 — STB（$b'\ b - b''$），將 $Builder\ b'$ 的所有資料附加到 $Builder\ b$。
- CF14 — STREFR（$b\ c - b'$）。
- CF15 — STBREFR（$b\ b' - b''$），STBREFR 的較長編碼。
- CF16 — STSLICER（$b\ s - b'$）。
- CF17 — STBR（$b\ b' - b''$），連接兩個 $Builder$，等同於 SWAP; STB。
- CF18 — STREFQ（$c\ b - c\ b\ -1$ 或 $b'\ 0$）。
- CF19 — STBREFQ（$b'\ b - b'\ b\ -1$ 或 $b''\ 0$）。
- CF1A — STSLICEQ（$s\ b - s\ b\ -1$ 或 $b'\ 0$）。
- CF1B — STBQ（$b'\ b - b'\ b\ -1$ 或 $b''\ 0$）。
- CF1C — STREFRQ（$b\ c - b\ c\ -1$ 或 $b'\ 0$）。
- CF1D — STBREFRQ（$b\ b' - b\ b'\ -1$ 或 $b''\ 0$）。

- CF1E — STSLICERQ（$b\ s - b\ s\ -1$ 或 $b''\ 0$）。
- CF1F — STBRQ（$b\ b' - b\ b'\ -1$ 或 $b''\ 0$）。
- CF20 — STREFCONST，等同於 PUSHREF; STREFR。
- CF21 — STREF2CONST，等同於 STREFCONST; STREFCONST。
- CF23 — ENDXC（$b\ x - c$），如果 $x \neq 0$，從 *Builder* b 建立特殊 或外來 單元（參見 **3.1.2**）。外來單元的型別必須儲存在 b 的前 8 位元中。如果 $x = 0$，它等同於 ENDC。否則，在建立外來單元之前，對 b 的資料和參照執行一些有效性檢查。
- CF28 — STILE4（$x\ b - b'$），儲存小端有符號 32 位元整數。
- CF29 — STULE4（$x\ b - b'$），儲存小端無符號 32 位元整數。
- CF2A — STILE8（$x\ b - b'$），儲存小端有符號 64 位元整數。
- CF2B — STULE8（$x\ b - b'$），儲存小端無符號 64 位元整數。
- CF30 — BDEPTH（$b - x$），傳回 *Builder* b 的深度。如果 b 中未儲存單元參照，則 $x = 0$；否則 x 是從 b 參照的單元的深度的最大值加一。
- CF31 — BBITS（$b - x$），傳回已儲存在 *Builder* b 中的資料位元數。
- CF32 — BREFS（$b - y$），傳回已儲存在 b 中的單元參照數。
- CF33 — BBITREFS（$b - x\ y$），傳回 b 中資料位元和單元參照的數量。
- CF35 — BREMBITS（$b - x'$），傳回仍可儲存在 b 中的資料位元數。
- CF36 — BREMREFS（$b - y'$）。
- CF37 — BREMBITREFS（$b - x'\ y'$）。
- CF38cc — BCHKBITS $cc + 1$（b –），檢查是否可以將 $cc + 1$ 位元儲存到 b 中，其中 $0 \leq cc \leq 255$。
- CF39 — BCHKBITS（$b\ x$ – ），檢查是否可以將 x 位元儲存到 b 中，$0 \leq x \leq 1023$。如果 b 中沒有空間容納 x 個更多位元，或如果 x 不在範圍 $0 \ldots 1023$ 內，則丟擲異常。

- CF3A — BCHKREFS（$b\ y$ – ），檢查是否可以將 y 個參照儲存到 b 中，$0 \le y \le 7$。
- CF3B — BCHKBITREFS（$b\ x\ y$ – ），檢查是否可以將 x 位元和 y 個參照儲存到 b 中，$0 \le x \le 1023$，$0 \le y \le 7$。
- CF3Ccc — BCHKBITSQ $cc+1$（b – ?），檢查是否可以將 $cc+1$ 位元儲存到 b 中，其中 $0 \le cc \le 255$。
- CF3D — BCHKBITSQ（$b\ x$ – ?），檢查是否可以將 x 位元儲存到 b 中，$0 \le x \le 1023$。
- CF3E — BCHKREFSQ（$b\ y$ – ?），檢查是否可以將 y 個參照儲存到 b 中，$0 \le y \le 7$。
- CF3F — BCHKBITREFSQ（$b\ x\ y$ – ?），檢查是否可以將 x 位元和 y 個參照儲存到 b 中，$0 \le x \le 1023$，$0 \le y \le 7$。
- CF40 — STZEROES（$b\ n$ – b'），將 n 個二元零儲存到 *Builder* b 中。
- CF41 — STONES（$b\ n$ – b'），將 n 個二元一儲存到 *Builder* b 中。
- CF42 — STSAME（$b\ n\ x$ – b'），將 n 個二元 x（$0 \le x \le 1$）儲存到 *Builder* b 中。
- CFC0_$xysss$ — STSLICECONST sss（b – b'），儲存由 $0 \le x \le 3$ 個參照和最多 $8y+1$ 資料位元組成的常數子切片 sss，$0 \le y \le 7$。假設有完成位元。
- CF81 — STSLICECONST `0' 或 STZERO（b – b'），儲存一個二元零。
- CF83 — STSLICECONST `1' 或 STONE（b – b'），儲存一個二元一。
- CFA2 — 等同於 STREFCONST。
- CFA3 — 幾乎等同於 STSLICECONST `1'; STREFCONST。
- CFC2 — 等同於 STREF2CONST。
- CFE2 — STREF3CONST。

A.7.2. 單元反序列化基本操作.

- D0 — CTOS（$c - s$），將 *Cell* 轉換為 *Slice*。請注意，c 必須是普通單元，或自動載入 以產生普通單元 c' 的外來單元（參見 **3.1.2**），之後轉換為 *Slice*。
- D1 — ENDS（$s -$ ），從堆疊中移除 *Slice* s，如果它不為空，則丟擲異常。
- D2cc — LDI $cc+1$（$s - x\ s'$），從 *Slice* s 載入（即，解析）有符號 $cc+1$ 位元整數 x，並將 s 的餘數傳回為 s'。
- D3cc — LDU $cc+1$（$s - x\ s'$），從 *Slice* s 載入無符號 $cc+1$ 位元整數 x。
- D4 — LDREF（$s - c\ s'$），從 s 載入單元參照 c。
- D5 — LDREFRTOS（$s - s'\ s''$），等同於 LDREF; SWAP; CTOS。
- D6cc — LDSLICE $cc+1$（$s - s''\ s'$），將 s 的下 $cc+1$ 位元切割成單獨的 *Slice* s''。
- D700 — LDIX（$s\ l - x\ s'$），從 *Slice* s 載入有符號 l 位元（$0 \le l \le 257$）整數 x，並將 s 的餘數傳回為 s'。
- D701 — LDUX（$s\ l - x\ s'$），從（s 的前 l 位元）載入無符號 l 位元整數 x，$0 \le l \le 256$。
- D702 — PLDIX（$s\ l - x$），從 *Slice* s 預載入有符號 l 位元整數，$0 \le l \le 257$。
- D703 — PLDUX($s\ l - x$)，從 s 預載入無符號 l 位元整數，$0 \le l \le 256$。
- D704 — LDIXQ（$s\ l - x\ s'\ -1$ 或 $s\ 0$），LDIX 的靜默版本：類似於 LDIX 從 s 載入有符號 l 位元整數，但傳回成功旗標，成功時等於 -1 或失敗時等於 0（如果 s 沒有 l 位元），而不是丟擲單元下溢異常。
- D705 — LDUXQ（$s\ l - x\ s'\ -1$ 或 $s\ 0$），LDUX 的靜默版本。
- D706 — PLDIXQ（$s\ l - x\ -1$ 或 0），PLDIX 的靜默版本。
- D707 — PLDUXQ（$s\ l - x\ -1$ 或 0），PLDUX 的靜默版本。
- D708cc — LDI $cc+1$（$s - x\ s'$），LDI 的較長編碼。

- D709cc — LDU $cc+1$ ($s - x\ s'$)，LDU 的較長編碼。
- D70Acc — PLDI $cc+1$ ($s - x$)，從 *Slice* s 預載入有符號 $cc+1$ 位元整數。
- D70Bcc — PLDU $cc+1$ ($s - x$)，從 s 預載入無符號 $cc+1$ 位元整數。
- D70Ccc — LDIQ $cc+1$ ($s - x\ s'\ -1$ 或 $s\ 0$)，LDI 的靜默版本。
- D70Dcc — LDUQ $cc+1$ ($s - x\ s'\ -1$ 或 $s\ 0$)，LDU 的靜默版本。
- D70Ecc — PLDIQ $cc+1$ ($s - x\ -1$ 或 0)，PLDI 的靜默版本。
- D70Fcc — PLDUQ $cc+1$ ($s - x\ -1$ 或 0)，PLDU 的靜默版本。
- D714_c — PLDUZ $32(c+1)$ ($s - s\ x$)，將 *Slice* s 的前 $32(c+1)$ 位元預載入到無符號整數 x 中，$0 \le c \le 7$。如果 s 短於必要長度，假設缺失的位元為零。此操作旨在與 IFBITJMP 和類似指令一起使用。
- D718 — LDSLICEX ($s\ l - s''\ s'$)，從 *Slice* s 載入前 $0 \le l \le 1023$ 位元到單獨的 *Slice* s'' 中，將 s 的餘數傳回為 s'。
- D719 — PLDSLICEX ($s\ l - s''$)，傳回 s 的前 $0 \le l \le 1023$ 位元為 s''。
- D71A — LDSLICEXQ ($s\ l - s''\ s'\ -1$ 或 $s\ 0$)，LDSLICEX 的靜默版本。
- D71B — PLDSLICEXQ ($s\ l - s'\ -1$ 或 0)，LDSLICEXQ 的靜默版本。
- D71Ccc — LDSLICE $cc+1$ ($s - s''\ s'$)，LDSLICE 的較長編碼。
- D71Dcc — PLDSLICE $cc+1$ ($s - s''$)，傳回 s 的前 $0 < cc+1 \le 256$ 位元為 s''。
- D71Ecc — LDSLICEQ $cc+1$ ($s - s''\ s'\ -1$ 或 $s\ 0$)，LDSLICE 的靜默版本。
- D71Fcc — PLDSLICEQ $cc+1$ ($s - s''\ -1$ 或 0)，PLDSLICE 的靜默版本。
- D720 — SDCUTFIRST ($s\ l - s'$)，傳回 s 的前 $0 \le l \le 1023$ 位元。它等同於 PLDSLICEX。
- D721 — SDSKIPFIRST ($s\ l - s'$)，傳回除前 $0 \le l \le 1023$ 位元之外的所有 s 位元。它等同於 LDSLICEX; NIP。

- D722 — SDCUTLAST（$s\ l - s'$），傳回 s 的最後 $0 \leq l \leq 1023$ 位元。
- D723 — SDSKIPLAST（$s\ l - s'$），傳回除最後 $0 \leq l \leq 1023$ 位元之外的所有 s 位元。
- D724 — SDSUBSTR（$s\ l\ l' - s'$），從偏移量 $0 \leq l \leq 1023$ 開始傳回 s 的 $0 \leq l' \leq 1023$ 位元，從而從 s 的資料中提取位元子字串。
- D726 — SDBEGINSX（$s\ s' - s''$），檢查 s 是否以（s' 的資料位元）開始，並在成功時從 s 中移除 s'。失敗時丟擲單元反序列化異常。基本操作 SDPFXREV 可視為 SDBEGINSX 的靜默版本。
- D727 — SDBEGINSXQ（$s\ s' - s''\ -1$ 或 $s\ 0$），SDBEGINSX 的靜默版本。
- D72A_$xsss$ — SDBEGINS（$s - s''$），檢查 s 是否以長度為 $8x + 3$（假設有繼續位元）的常數位元字串 sss 開始，其中 $0 \leq x \leq 127$，並在成功時從 s 中移除 sss。
- D72802 — SDBEGINS `0'（$s - s''$），檢查 s 是否以二元零開始。
- D72806 — SDBEGINS `1'（$s - s''$），檢查 s 是否以二元一開始。
- D72E_$xsss$ — SDBEGINSQ（$s - s''\ -1$ 或 $s\ 0$），SDBEGINS 的靜默版本。
- D730 — SCUTFIRST（$s\ l\ r - s'$），傳回 s 的前 $0 \leq l \leq 1023$ 位元和前 $0 \leq r \leq 4$ 個參照。
- D731 — SSKIPFIRST（$s\ l\ r - s'$）。
- D732 — SCUTLAST（$s\ l\ r - s'$），傳回 s 的最後 $0 \leq l \leq 1023$ 資料位元和最後 $0 \leq r \leq 4$ 個參照。
- D733 — SSKIPLAST（$s\ l\ r - s'$）。
- D734 — SUBSLICE（$s\ l\ r\ l'\ r' - s'$），在跳過前 $0 \leq l \leq 1023$ 位元和前 $0 \leq r \leq 4$ 個參照後，從 *Slice* s 傳回 $0 \leq l' \leq 1023$ 位元和 $0 \leq r' \leq 4$ 個參照。
- D736 — SPLIT（$s\ l\ r - s'\ s''$），將 s 的前 $0 \leq l \leq 1023$ 資料位元和前 $0 \leq r \leq 4$ 個參照拆分為 s'，將 s 的餘數傳回為 s''。
- D737 — SPLITQ（$s\ l\ r - s'\ s''\ -1$ 或 $s\ 0$），SPLIT 的靜默版本。

- D739 — XCTOS（$c - s\ ?$），將普通或外來單元轉換為 *Slice*，就好像它是普通單元一樣。傳回一個旗標，指示 c 是否為外來單元。如果是這種情況，其型別稍後可以從 s 的前八位元反序列化。
- D73A — XLOAD（$c - c'$），載入外來單元 c 並傳回普通單元 c'。如果 c 已經是普通單元，則不做任何事情。如果無法載入 c，則丟擲異常。
- D73B — XLOADQ（$c - c'\ -1$ 或 $c\ 0$），像 XLOAD 一樣載入外來單元 c，但在失敗時傳回 0。
- D741 — SCHKBITS（$s\ l\ -$），檢查 *Slice* s 中是否至少有 l 資料位元。如果不是這種情況，則丟擲單元反序列化（即，單元下溢）異常。
- D742 — SCHKREFS（$s\ r\ -$），檢查 *Slice* s 中是否至少有 r 個參照。
- D743 — SCHKBITREFS（$s\ l\ r\ -$），檢查 *Slice* s 中是否至少有 l 資料位元和 r 個參照。
- D745 — SCHKBITSQ（$s\ l\ -\ ?$），檢查 *Slice* s 中是否至少有 l 資料位元。
- D746 — SCHKREFSQ（$s\ r\ -\ ?$），檢查 *Slice* s 中是否至少有 r 個參照。
- D747 — SCHKBITREFSQ（$s\ l\ r\ -\ ?$），檢查 *Slice* s 中是否至少有 l 資料位元和 r 個參照。
- D748 — PLDREFVAR（$s\ n\ -\ c$），對於 $0 \leq n \leq 3$，傳回 *Slice* s 的第 n 個單元參照。
- D749 — SBITS（$s - l$），傳回 *Slice* s 中的資料位元數。
- D74A — SREFS（$s - r$），傳回 *Slice* s 中的參照數。
- D74B — SBITREFS（$s - l\ r$），傳回 s 中的資料位元數和參照數。
- D74E_n — PLDREFIDX n（$s - c$），傳回 *Slice* s 的第 n 個單元參照，其中 $0 \leq n \leq 3$。
- D74C — PLDREF（$s - c$），預載入 *Slice* 的第一個單元參照。
- D750 — LDILE4（$s - x\ s'$），載入小端序有符號 32 位元整數。
- D751 — LDULE4（$s - x\ s'$），載入小端序無符號 32 位元整數。

- D752 — LDILE8（$s - x\ s'$），載入小端序有符號 64 位元整數。
- D753 — LDULE8（$s - x\ s'$），載入小端序無符號 64 位元整數。
- D754 — PLDILE4（$s - x$），預載入小端序有符號 32 位元整數。
- D755 — PLDULE4（$s - x$），預載入小端序無符號 32 位元整數。
- D756 — PLDILE8（$s - x$），預載入小端序有符號 64 位元整數。
- D757 — PLDULE8（$s - x$），預載入小端序無符號 64 位元整數。
- D758 — LDILE4Q（$s - x\ s'\ -1$ 或 $s\ 0$），靜默載入小端序有符號 32 位元整數。
- D759 — LDULE4Q（$s - x\ s'\ -1$ 或 $s\ 0$），靜默載入小端序無符號 32 位元整數。
- D75A — LDILE8Q（$s - x\ s'\ -1$ 或 $s\ 0$），靜默載入小端序有符號 64 位元整數。
- D75B — LDULE8Q（$s - x\ s'\ -1$ 或 $s\ 0$），靜默載入小端序無符號 64 位元整數。
- D75C — PLDILE4Q（$s - x\ -1$ 或 0），靜默預載入小端序有符號 32 位元整數。
- D75D — PLDULE4Q（$s - x\ -1$ 或 0），靜默預載入小端序無符號 32 位元整數。
- D75E — PLDILE8Q（$s - x\ -1$ 或 0），靜默預載入小端序有符號 64 位元整數。
- D75F — PLDULE8Q（$s - x\ -1$ 或 0），靜默預載入小端序無符號 64 位元整數。
- D760 — LDZEROES（$s - n\ s'$），傳回 s 中開頭零位元的計數 n，並從 s 中移除這些位元。
- D761 — LDONES（$s - n\ s'$），傳回 s 中開頭一位元的計數 n，並從 s 中移除這些位元。

- D762 — LDSAME（$s\ x - n\ s'$），傳回 s 中等於 $0 \le x \le 1$ 的開頭位元計數 n，並從 s 中移除這些位元。
- D764 — SDEPTH（$s - x$），傳回 *Slice* s 的深度。如果 s 沒有參照，則 $x = 0$；否則 x 為從 s 參照的單元深度的最大值加一。
- D765 — CDEPTH（$c - x$），傳回 *Cell* c 的深度。如果 c 沒有參照，則 $x = 0$；否則 x 為從 c 參照的單元深度的最大值加一。如果 c 是 *Null* 而非 *Cell*，則傳回零。

A.8 繼續和控制流程基本操作

A.8.1. 無條件控制流程基本操作.

- D8 — EXECUTE 或 CALLX(c –)，呼叫 或執行 繼續 c(即，cc $\leftarrow c \circ_0$ cc)。
- D9 — JMPX（c – ），跳躍，或轉移控制到繼續 c（即，cc $\leftarrow c \circ_0$ c0，或者更準確地說 cc $\leftarrow (c \circ_0$ c0$) \circ_1$ c1）。之前當前繼續 cc 的餘數被丟棄。
- DApr — CALLXARGS p,r（c – ），以 p 個參數呼叫繼續 c 並期望 r 個回傳值，$0 \le p \le 15$，$0 \le r \le 15$。
- DB0p — CALLXARGS p,-1（c – ），以 $0 \le p \le 15$ 個參數呼叫繼續 c，期望任意數量的回傳值。
- DB1p — JMPXARGS p（c – ），跳躍到繼續 c，僅從當前堆疊傳遞頂端的 $0 \le p \le 15$ 個值給它（當前堆疊的餘數被丟棄）。
- DB2r — RETARGS r，回傳 到 c0，從當前堆疊取得 $0 \le r \le 15$ 個回傳值。
- DB30 — RET 或 RETTRUE，回傳 到 c0 的繼續（即，執行 cc $\leftarrow$ c0）。當前繼續 cc 的餘數被丟棄。大約等同於 PUSH c0; JMPX。
- DB31 — RETALT 或 RETFALSE，回傳 到 c1 的繼續（即，cc $\leftarrow$ c1）。大約等同於 PUSH c1; JMPX。
- DB32 — BRANCH 或 RETBOOL(f –)，如果整數 $f \neq 0$，執行 RETTRUE，或如果 $f = 0$，執行 RETFALSE。

- DB34 — CALLCC（c – ），以當前繼續呼叫，轉移控制到 c，將 cc 的舊值壓入 c 的堆疊（而非丟棄它或將它寫入新的 c0）。
- DB35 — JMPXDATA（c – ），類似於 CALLCC，但當前繼續的餘數（cc 的舊值）在壓入 c 的堆疊之前被轉換為 *Slice*。
- DB36pr — CALLCCARGS p,r（c – ），類似於 CALLXARGS，但將 cc 的舊值（連同原始堆疊的頂端 $0 \le p \le 15$ 個值）壓入新呼叫的繼續 c 的堆疊，將 cc.nargs 設定為 $-1 \le r \le 14$。
- DB38 — CALLXVARARGS（$c\ p\ r$ – ），類似於 CALLXARGS，但從堆疊取得 $-1 \le p, r \le 254$。接下來的三個操作也從堆疊取得 p 和 r，兩者都在範圍 $-1 \ldots 254$ 內。
- DB39 — RETVARARGS（$p\ r$ – ），類似於 RETARGS。
- DB3A — JMPXVARARGS（$c\ p\ r$ – ），類似於 JMPXARGS。
- DB3B — CALLCCVARARGS（$c\ p\ r$ – ），類似於 CALLCCARGS。
- DB3C — CALLREF，等同於 PUSHREFCONT; CALLX。
- DB3D — JMPREF，等同於 PUSHREFCONT; JMPX。
- DB3E — JMPREFDATA，等同於 PUSHREFCONT; JMPXDATA。
- DB3F — RETDATA，等同於 PUSH c0; JMPXDATA。以此方式，當前繼續的餘數被轉換為 *Slice* 並回傳給呼叫者。

A.8.2. 條件式控制流程基本操作.

- DC — IFRET（f – ），執行 RET，但僅當整數 f 非零時。如果 f 是 NaN，丟擲整數溢位異常。
- DD — IFNOTRET（f – ），執行 RET，但僅當整數 f 為零時。
- DE — IF（$f\ c$ – ），對 c 執行 EXECUTE（即，執行 c），但僅當整數 f 非零時。否則僅丟棄兩個值。
- DF — IFNOT（$f\ c$ – ），執行繼續 c，但僅當整數 f 為零時。否則僅丟棄兩個值。
- E0 — IFJMP（$f\ c$ – ），跳躍到 c（類似於 JMPX），但僅當 f 非零時。

- E1 — IFNOTJMP（$f\ c\ -$），跳躍到 c（類似於 JMPX），但僅當 f 為零時。
- E2 — IFELSE（$f\ c\ c'\ -$），如果整數 f 非零，執行 c，否則執行 c'。等同於 CONDSELCHK; EXECUTE。
- E300 — IFREF（$f\ -$），等同於 PUSHREFCONT; IF，並最佳化為當 $f = 0$ 時單元參照實際上不會載入到 *Slice* 然後轉換為普通 *Continuation*。類似的備註適用於接下來的三個基本操作。
- E301 — IFNOTREF（$f\ -$），等同於 PUSHREFCONT; IFNOT。
- E302 — IFJMPREF（$f\ -$），等同於 PUSHREFCONT; IFJMP。
- E303 — IFNOTJMPREF（$f\ -$），等同於 PUSHREFCONT; IFNOTJMP。
- E304 — CONDSEL（$f\ x\ y\ -\ x$ 或 y），如果整數 f 非零，傳回 x，否則傳回 y。請注意，不會對 x 和 y 執行型別檢查；因此，它更像是條件式堆疊操作。大致等同於 ROT; ISZERO; INC; ROLLX; NIP。
- E305 — CONDSELCHK（$f\ x\ y\ -\ x$ 或 y），與 CONDSEL 相同，但首先檢查 x 和 y 是否具有相同型別。
- E308 — IFRETALT（$f\ -$），如果整數 $f \neq 0$，執行 RETALT。
- E309 — IFNOTRETALT（$f\ -$），如果整數 $f = 0$，執行 RETALT。
- E30D — IFREFELSE（$f\ c\ -$），等同於 PUSHREFCONT; SWAP; IFELSE，並最佳化為當 $f = 0$ 時單元參照實際上不會載入到 *Slice* 然後轉換為普通 *Continuation*。類似的備註適用於接下來的兩個基本操作：*Cell* 僅在必要時轉換為 *Continuation*。
- E30E — IFELSEREF（$f\ c\ -$），等同於 PUSHREFCONT; IFELSE。
- E30F — IFREFELSEREF（$f\ -$），等同於 PUSHREFCONT; PUSHREFCONT; IFELSE。
- E310–E31F — 保留給帶有中斷運算子的迴圈，參見下面的 **A.8.3**。
- E39_n — IFBITJMP n（$x\ c\ -\ x$），檢查整數 x 中的位元 $0 \leq n \leq 31$ 是否設定，如果是，則對繼續 c 執行 JMPX。值 x 留在堆疊中。

- E3B_n — IFNBITJMP n（x c – x），如果整數 x 中的位元 $0 \le n \le 31$ 未設定，則跳躍到 c。

- E3D_n — IFBITJMPREF n（x – x），如果整數 x 中的位元 $0 \le n \le 31$ 已設定，則執行 JMPREF。

- E3F_n — IFNBITJMPREF n（x – x），如果整數 x 中的位元 $0 \le n \le 31$ 未設定，則執行 JMPREF。

A.8.3. 控制流程基本操作：迴圈. 下面列出的大多數迴圈基本操作都是在非常規繼續的幫助下實作的，例如 ec_until（參見 **4.1.5**），將迴圈主體和原始當前繼續 cc 儲存為此非常規繼續的引數。通常會建構一個合適的非常規繼續，然後儲存到迴圈主體繼續保存列表中作為 c0；之後，修改後的迴圈主體繼續被載入到 cc 並以通常的方式執行。所有這些迴圈基本操作都有 *BRK 版本，適用於跳出迴圈；它們還將 c1 設定為原始當前繼續（或對於 *ENDBRK 版本為原始 c0），並將舊的 c1 儲存到原始當前繼續的保存列表中（或對於 *ENDBRK 版本儲存到原始 c0 的保存列表中）。

- E4 — REPEAT（n c – ），如果整數 n 非負，則執行繼續 c n 次。如果 $n \ge 2^{31}$ 或 $n < -2^{31}$，產生範圍檢查異常。請注意，c 代碼內的 RET 作為 continue，而非 break。應使用替代（實驗性）迴圈或替代 RETALT（連同迴圈前的 SETEXITALT）來從迴圈中 break。

- E5 — REPEATEND（n – ），類似於 REPEAT，但應用於當前繼續 cc。

- E6 — UNTIL（c – ），執行繼續 c，然後從結果堆疊彈出整數 x。如果 x 為零，執行此迴圈的另一個迭代。此基本操作的實際實作涉及非常規繼續 ec_until（參見 **4.1.5**），其引數設定為迴圈主體（繼續 c）和原始當前繼續 cc。然後將此非常規繼續儲存到 c 的保存列表中作為 c.c0，然後執行修改後的 c。其他迴圈基本操作也以類似方式在合適的非常規繼續的幫助下實作。

- E7 — UNTILEND（– ），類似於 UNTIL，但在迴圈中執行當前繼續 cc。當迴圈退出條件滿足時，執行 RET。

- E8 — WHILE（c' c – ），執行 c' 並從結果堆疊彈出整數 x。如果 x 為零，退出迴圈並將控制轉移到原始 cc。如果 x 非零，執行 c，然後開始新的迭代。

- E9 — WHILEEND（c' – ），類似於 WHILE，但使用當前繼續 cc 作為迴圈主體。

- EA — AGAIN（c – ），類似於 REPEAT，但無限次執行 c。RET 僅開始無限迴圈的新迭代，只能透過異常或 RETALT（或顯式 JMPX）退出。
- EB — AGAINEND（– ），類似於 AGAIN，但對當前繼續 cc 執行。
- E314 — REPEATBRK（n c – ），類似於 REPEAT，但也在將 c1 的舊值儲存到原始 cc 的保存列表後，將 c1 設定為原始 cc。以此方式，RETALT 可用於跳出迴圈主體。
- E315 — REPEATENDBRK（n – ），類似於 REPEATEND，但也在將 c1 的舊值儲存到原始 c0 的保存列表後，將 c1 設定為原始 c0。等同於 SAMEALTSAVE; REPEATEND。
- E316 — UNTILBRK（c – ），類似於 UNTIL，但也以與 REPEATBRK 相同的方式修改 c1。
- E317 — UNTILENDBRK（– ），等同於 SAMEALTSAVE; UNTILEND。
- E318 — WHILEBRK（c' c – ），類似於 WHILE，但也以與 REPEATBRK 相同的方式修改 c1。
- E319 — WHILEENDBRK（c – ），等同於 SAMEALTSAVE; WHILEEND。
- E31A — AGAINBRK（c – ），類似於 AGAIN，但也以與 REPEATBRK 相同的方式修改 c1。
- E31B — AGAINENDBRK（– ），等同於 SAMEALTSAVE; AGAINEND。

A.8.4. 操作繼續的堆疊.

- ECrn — SETCONTARGS r,n（x_1 x_2…x_r c – c'），類似於 SETCONTARGS r，但將 c.nargs 設定為 c' 的堆疊最終大小加上 n。換句話說，將 c 轉換為閉包 或部分應用函數，缺少 $0 \le n \le 14$ 個引數。
- EC0n — SETNUMARGS n 或 SETCONTARGS 0,n（c – c'），將 c.nargs 設定為 n 加上 c 的堆疊的當前深度，其中 $0 \le n \le 14$。如果 c.nargs 已設定為非負值，則不執行任何操作。
- ECrF — SETCONTARGS r 或 SETCONTARGS r,-1（x_1 x_2…x_r c – c'），將 $0 \le r \le 15$ 個值 $x_1 \ldots x_r$ 壓入（繼續 c 的副本的）堆疊，從 x_1 開始。如果 c 的堆疊最終深度大於 c.nargs，則產生堆疊溢位異常。

- ED0p — RETURNARGS p ($-$)，僅在當前堆疊中保留頂端 $0 \leq p \leq 15$ 個值（有點類似於 ONLYTOPX），所有未使用的底部值不會被丟棄，而是以與 SETCONTARGS 相同的方式儲存到繼續 c0 中。

- ED10 — RETURNVARARGS (p $-$)，類似於 RETURNARGS，但從堆疊取得整數 $0 \leq p \leq 255$。

- ED11 — SETCONTVARARGS (x_1 x_2...x_r c r n $-$ c')，類似於 SETCONTARGS，但從堆疊取得 $0 \leq r \leq 255$ 和 $-1 \leq n \leq 255$。

- ED12 — SETNUMVARARGS (c n $-$ c')，其中 $-1 \leq n \leq 255$。如果 $n = -1$，此操作不執行任何操作 ($c' = c$)。否則其動作類似於 SETNUMARGS n，但從堆疊取得 n。

A.8.5. 建立簡單繼續和閉包.

- ED1E — BLESS (s $-$ c)，將 *Slice* s 轉換為簡單的普通繼續 c，其中 c.code $= s$ 且堆疊和保存列表為空。

- ED1F — BLESSVARARGS (x_1...x_r s r n $-$ c)，等同於 ROT; BLESS; ROTREV; SETCONTVARARGS。

- EErn — BLESSARGS r, n (x_1...x_r s $-$ c)，其中 $0 \leq r \leq 15$，$-1 \leq n \leq 14$，等同於 BLESS; SETCONTARGS r, n。n 的值在指令內由 4 位元整數 $n \bmod 16$ 表示。

- EE0n — BLESSNUMARGS n 或 BLESSARGS 0,n (s $-$ c)，也將 *Slice* s 轉換為 *Continuation* c，但將 c.nargs 設定為 $0 \leq n \leq 14$。

A.8.6. 繼續保存列表和控制暫存器的操作.

- ED4i — PUSH c(i) 或 PUSHCTR c(i) ($-$ x)，壓入控制暫存器 c(i) 的當前值。如果在當前代碼頁中不支援該控制暫存器，或如果它沒有值，則觸發異常。

- ED44 — PUSH c4 或 PUSHROOT，壓入「全域資料根」單元參照，從而啟用對持久性智慧合約資料的存取。

- ED5i — POP c(i) 或 POPCTR c(i) (x $-$)，從堆疊彈出值 x 並將其儲存到控制暫存器 c(i)（如果在當前代碼頁中支援）。請注意，如果控制暫存器僅接受特定型別的值，則可能發生型別檢查異常。

- ED54 — POP c4 或 POPROOT，設定「全域資料根」單元參照，從而允許修改持久性智慧合約資料。
- ED6i — SETCONT c(i) 或 SETCONTCTR c(i)（$x\ c - c'$），將 x 儲存到繼續 c 的保存列表中作為 c(i)，並傳回結果繼續 c'。幾乎所有繼續操作都可以用 SETCONTCTR、POPCTR 和 PUSHCTR 來表達。
- ED7i — SETRETCTR c(i)（x –），等同於 PUSH c0; SETCONTCTR c(i); POP c0。
- ED8i — SETALTCTR c(i)（x –），等同於 PUSH c1; SETCONTCTR c(i); POP c0。
- ED9i — POPSAVE c(i) 或 POPCTRSAVE c(i)（x –），類似於 POP c(i)，但也將 c(i) 的舊值儲存到繼續 c0 中。等同於（除了異常）SAVECTR c(i); POP c(i)。
- EDAi — SAVE c(i) 或 SAVECTR c(i)（– ），將 c(i) 的當前值儲存到繼續 c0 的保存列表中。如果 c0 的保存列表中已存在 c(i) 的項目，則不執行任何操作。等同於 PUSH c(i); SETRETCTR c(i)。
- EDBi — SAVEALT c(i) 或 SAVEALTCTR c(i)（– ），類似於 SAVE c(i)，但將 c(i) 的當前值儲存到 c1 的保存列表中，而非 c0。
- EDCi — SAVEBOTH c(i) 或 SAVEBOTHCTR c(i)（– ），等同於 DUP; SAVE c(i); SAVEALT c(i)。
- EDE0 — PUSHCTRX（$i - x$），類似於 PUSHCTR c(i)，但從堆疊取得 i，$0 \leq i \leq 255$。請注意，此基本操作是少數「奇特」基本操作之一，它們不像堆疊操作基本操作那樣是多型的，同時也沒有明確定義的參數和回傳值型別，因為 x 的型別取決於 i。
- EDE1 — POPCTRX（$x\ i$ – ），類似於 POPCTR c(i)，但從堆疊取得 $0 \leq i \leq 255$。
- EDE2 — SETCONTCTRX（$x\ c\ i - c'$），類似於 SETCONTCTR c(i)，但從堆疊取得 $0 \leq i \leq 255$。
- EDF0 — COMPOS 或 BOOLAND（$c\ c' - c''$），計算組合 $c \circ_0 c'$，其含義為「執行 c，如果成功，執行 c'」（如果 c 是布林電路）或簡單地「執行 c，然後 c'」。等同於 SWAP; SETCONT c0。

- EDF1 — COMPOSALT 或 BOOLOR（$c\ c'$ – c''），計算替代組合 $c \circ_1 c'$，其含義為「執行 c，如果不成功，執行 c'」（如果 c 是布林電路）。等同於 SWAP; SETCONT c1。

- EDF2 — COMPOSBOTH（$c\ c'$ – c''），計算 $(c \circ_0 c') \circ_1 c'$，其含義為「計算布林電路 c，然後計算 c'，無論 c 的結果如何」。

- EDF3 — ATEXIT（c – ），設定 c0 ← $c \circ_0$ c0。換句話說，c 將在退出當前子程式之前執行。

- EDF4 — ATEXITALT（c – ），設定 c1 ← $c \circ_1$ c1。換句話說，c 將在透過其替代回傳路徑退出當前子程式之前執行。

- EDF5 — SETEXITALT（c – ），設定 c1 ← $(c \circ_0$ c0$) \circ_1$ c1。以此方式，後續的 RETALT 將首先執行 c，然後將控制轉移到原始 c0。例如，這可用於退出巢狀迴圈。

- EDF6 — THENRET（c – c'），計算 $c' := c \circ_0$ c0

- EDF7 — THENRETALT（c – c'），計算 $c' := c \circ_0$ c1

- EDF8 — INVERT（– ），交換 c0 和 c1。

- EDF9 — BOOLEVAL（c – ?），執行 cc ← $\big(c \circ_0 ((\text{PUSH } -1) \circ_0 \text{cc})\big) \circ_1 ((\text{PUSH } 0) \circ_0 \text{cc})$。如果 c 代表布林電路，淨效果是評估它並在繼續之前將 -1 或 0 壓入堆疊。

- EDFA — SAMEALT（– ），設定 $c_1 := c_0$。等同於 PUSH c0; POP c1。

- EDFB — SAMEALTSAVE（– ），設定 $c_1 := c_0$，但首先將 c_1 的舊值儲存到 c_0 的保存列表中。等同於 SAVE c1; SAMEALT。

- EErn — BLESSARGS r, n（$x_1 \ldots x_r\ s$ – c），在 **A.8.4** 中描述。

A.8.7. 字典子程式呼叫和跳躍.

- F0n — CALL n 或 CALLDICT n（– n），呼叫 c3 中的繼續，將整數 $0 \le n \le 255$ 作為引數壓入其堆疊。大約等同於 PUSHINT n; PUSH c3; EXECUTE。

- F12_n — CALL n 對於 $0 \le n < 2^{14}$（– n），針對較大 n 值的 CALL n 編碼。

- F16_n — JMP n 或 JMPDICT n（– n），跳躍到 c3 中的繼續，將整數 $0 \le n < 2^{14}$ 作為其引數壓入。大約等同於 PUSHINT n; PUSH c3; JMPX。

- F1A_n — PREPARE n 或 PREPAREDICT n（– n c），等同於 PUSHINT n; PUSH c3，對於 $0 \le n < 2^{14}$。以此方式，CALL n 大約等同於 PREPARE n; EXECUTE，而 JMP n 大約等同於 PREPARE n; JMPX。例如，在這裡可以使用 CALLARGS 或 CALLCC 代替 EXECUTE。

A.9 異常產生和處理基本操作

A.9.1. 丟擲異常.

- F22_nn — THROW nn（– 0 nn），以參數零丟擲異常 $0 \le nn \le 63$。換句話說，它將控制轉移到 c2 中的繼續，將 0 和 nn 壓入其堆疊，並完全丟棄舊堆疊。

- F26_nn — THROWIF nn（f – ），僅當整數 $f \neq 0$ 時，以參數零丟擲異常 $0 \le nn \le 63$。

- F2A_nn — THROWIFNOT nn（f – ），僅當整數 $f = 0$ 時，以參數零丟擲異常 $0 \le nn \le 63$。

- F2C4_nn — THROW nn 對於 $0 \le nn < 2^{11}$，針對較大 nn 值的 THROW nn 編碼。

- F2CC_nn — THROWARG nn（x – x nn），以參數 x 丟擲異常 $0 \le nn < 2^{11}$，透過將 x 和 nn 複製到 c2 的堆疊並將控制轉移到 c2。

- F2D4_nn — THROWIF nn（f – ）對於 $0 \le nn < 2^{11}$。

- F2DC_nn — THROWARGIF nn（x f – ），僅當整數 $f \neq 0$ 時，以參數 x 丟擲異常 $0 \le nn < 2^{11}$。

- F2E4_nn — THROWIFNOT nn（f – ）對於 $0 \le nn < 2^{11}$。

- F2EC_nn — THROWARGIFNOT nn（x f – ），僅當整數 $f = 0$ 時，以參數 x 丟擲異常 $0 \le nn < 2^{11}$。

- F2F0 — THROWANY（n – 0 n），以參數零丟擲異常 $0 \le n < 2^{16}$。大約等同於 PUSHINT 0; SWAP; THROWARGANY。

- F2F1 — THROWARGANY（$x\ n - x\ n$），以參數 x 丟擲異常 $0 \le n < 2^{16}$，將控制轉移到 c2 中的繼續。大約等同於 PUSH c2; JMPXARGS 2。
- F2F2 — THROWANYIF（$n\ f -$），僅當 $f \neq 0$ 時，以參數零丟擲異常 $0 \le n < 2^{16}$。
- F2F3 — THROWARGANYIF（$x\ n\ f -$），僅當 $f \neq 0$ 時，以參數 x 丟擲異常 $0 \le n < 2^{16}$。
- F2F4 — THROWANYIFNOT（$n\ f -$），僅當 $f = 0$ 時，以參數零丟擲異常 $0 \le n < 2^{16}$。
- F2F5 — THROWARGANYIFNOT（$x\ n\ f -$），僅當 $f = 0$ 時，以參數 x 丟擲異常 $0 \le n < 2^{16}$。

A.9.2. 捕捉和處理異常.

- F2FF — TRY（$c\ c' -$），將 c2 設定為 c'，首先將 c2 的舊值儲存到 c' 的保存列表和當前繼續的保存列表中，當前繼續儲存在 c.c0 和 c'.c0 中。然後類似於 EXECUTE 地執行 c。如果 c 沒有丟擲任何異常，c2 的原始值會在從 c 回傳時自動恢復。如果發生異常，執行轉移到 c'，但在此過程中恢復 c2 的原始值，以便 c' 可以在無法自行處理時透過 THROWANY 重新丟擲異常。
- F3pr — TRYARGS p,r（$c\ c' -$），類似於 TRY，但內部使用 CALLARGS p,r 代替 EXECUTE。以此方式，除了頂端 $0 \le p \le 15$ 個堆疊元素外的所有元素都將儲存到當前繼續的堆疊中，然後在從 c 或 c' 回傳時恢復，將 c 或 c' 的結果堆疊的頂端 $0 \le r \le 15$ 個值複製為回傳值。

A.10 字典操作基本操作

TVM 的字典支援在 **3.3** 中詳細討論。字典的基本操作列在 **3.3.10** 中，而字典操作基本操作的分類在 **3.3.11** 中提供。在這裡我們使用這些章節中引入的概念和符號。

字典作為 TVM 堆疊值有兩種不同的表示：

- *Slice* s，包含 TL-B 型別 $HashmapE(n, X)$ 的值的序列化。換句話說，s 由一個等於零的位元（如果字典為空）或一個等於一的位元和對包含二元樹根的 *Cell* 的參照組成，即 $Hashmap(n, X)$ 型別的序列化值。

- 「也許 $Cell$」$c^?$，即一個值，它要麼是 $Cell$(如前所述包含 $Hashmap(n, X)$ 型別的序列化值）要麼是 $Null$（對應於空字典）。當使用「也許 $Cell$」$c^?$ 來表示字典時，我們通常在堆疊符號中將其表示為 D。

下面列出的大多數字典基本操作接受並傳回第二種形式的字典，這對於堆疊操作更方便。然而，較大 TL-B 物件內的序列化字典使用第一種表示。

以 F4 和 F5 開頭的操作碼保留給字典操作。

A.10.1. 字典建立.

- 6D — NEWDICT（– D），傳回新的空字典。它是 PUSHNULL 的替代助記符，參見 **A.3.1**。
- 6E — DICTEMPTY（D – ?），檢查字典 D 是否為空，並相應地傳回 -1 或 0。它是 ISNULL 的替代助記符，參見 **A.3.1**。

A.10.2. 字典序列化和反序列化.

- CE — STDICTS（s b – b'），將 $Slice$ 表示的字典 s 儲存到 $Builder$ b 中。它實際上是 STSLICE 的同義詞。
- F400 — STDICT 或 STOPTREF（D b – b'），將字典 D 儲存到 $Builder$ b 中，傳回結果 $Builder$ b'。換句話說，如果 D 是單元，執行 STONE 和 STREF；如果 D 是 $Null$，執行 NIP 和 STZERO；否則丟擲型別檢查異常。
- F401 — SKIPDICT 或 SKIPOPTREF（s – s'），等同於 LDDICT; NIP。
- F402 — LDDICTS（s – s' s''），從 $Slice$ s 載入（解析）（$Slice$ 表示的）字典 s'，並將 s 的餘數傳回為 s''。這是所有 $HashmapE(n, X)$ 字典型別的「拆分函數」。
- F403 — PLDDICTS（s – s'），從 $Slice$ s 預載入（$Slice$ 表示的）字典 s'。大約等同於 LDDICTS; DROP。
- F404 — LDDICT 或 LDOPTREF（s – D s'），從 $Slice$ s 載入（解析）字典 D，並將 s 的餘數傳回為 s'。可應用於字典或任意 $(\hat{}Y)^?$ 型別的值。
- F405 — PLDDICT 或 PLDOPTREF（s – D），從 $Slice$ s 預載入字典 D。大約等同於 LDDICT; DROP。

- F406 — LDDICTQ（$s - D\ s'\ -1$ 或 $s\ 0$），LDDICT 的靜默版本。
- F407 — PLDDICTQ（$s - D\ -1$ 或 0），PLDDICT 的靜默版本。

A.10.3. GET **字典操作.**

- F40A — DICTGET（$k\ D\ n - x\ -1$ 或 0），在 n 位元鍵的 $HashmapE(n, X)$ 型別字典 D 中查找鍵 k（由 *Slice* 表示，其前 $0 \leq n \leq 1023$ 資料位元用作鍵）。成功時，傳回找到的值作為 *Slice* x。
- F40B — DICTGETREF（$k\ D\ n - c\ -1$ 或 0），類似於 DICTGET，但成功時對 x 應用 LDREF; ENDS。此操作對於 $HashmapE(n, \hat{}Y)$ 型別的字典很有用。
- F40C — DICTIGET（$i\ D\ n - x\ -1$ 或 0），類似於 DICTGET，但使用有符號（大端序）n 位元 *Integer* i 作為鍵。如果 i 不適合 n 位元，傳回 0。如果 i 是 NaN，丟擲整數溢位異常。
- F40D — DICTIGETREF（$i\ D\ n - c\ -1$ 或 0），結合 DICTIGET 與 DICTGETREF：它使用有符號 n 位元 *Integer* i 作為鍵，並在成功時傳回 *Cell* 而非 *Slice*。
- F40E — DICTUGET（$i\ D\ n - x\ -1$ 或 0），類似於 DICTIGET，但使用無符號（大端序）n 位元 *Integer* i 作為鍵。
- F40F — DICTUGETREF（$i\ D\ n - c\ -1$ 或 0），類似於 DICTIGETREF，但使用無符號 n 位元 *Integer* 鍵 i。

A.10.4. SET/REPLACE/ADD **字典操作.** 以下字典基本操作的助記符以系統化方式根據正規表達式 DICT[, I, U](SET, REPLACE, ADD)[GET][REF] 構造，取決於使用的鍵型別（*Slice* 或有符號或無符號 *Integer*）、要執行的字典操作，以及接受和傳回值的方式（作為 *Cell* 或作為 *Slice*）。因此，我們僅對某些基本操作提供詳細說明，假設此資訊足以讓讀者理解其餘基本操作的精確動作。

- F412 — DICTSET（$x\ k\ D\ n - D'$），在字典 D（也由 *Slice* 表示）中將與 n 位元鍵 k（如 DICTGET 中由 *Slice* 表示）關聯的值設定為值 x（再次為 *Slice*），並傳回結果字典 D'。
- F413 — DICTSETREF（$c\ k\ D\ n - D'$），類似於 DICTSET，但將值設定為對 *Cell* c 的參照。

- F414 — DICTISET（$x\ i\ D\ n - D'$），類似於 DICTSET，但鍵由（大端序）有符號 n 位元整數 i 表示。如果 i 不適合 n 位元，產生範圍檢查異常。
- F415 — DICTISETREF（$c\ i\ D\ n - D'$），類似於 DICTSETREF，但鍵為有符號 n 位元整數，如 DICTISET 中。
- F416 — DICTUSET（$x\ i\ D\ n - D'$），類似於 DICTISET，但 i 為無符號 n 位元整數。
- F417 — DICTUSETREF（$c\ i\ D\ n - D'$），類似於 DICTISETREF，但 i 為無符號。
- F41A — DICTSETGET（$x\ k\ D\ n - D'\ y\ -1$ 或 $D'\ 0$），結合 DICTSET 與 DICTGET：它將對應於鍵 k 的值設定為 x，但也傳回與所討論的鍵關聯的舊值 y（如果存在）。
- F41B — DICTSETGETREF（$c\ k\ D\ n - D'\ c'\ -1$ 或 $D'\ 0$），類似於 DICTSETGET，結合 DICTSETREF 與 DICTGETREF。
- F41C — DICTISETGET($x\ i\ D\ n - D'\ y\ -1$ 或 $D'\ 0$)，類似於 DICTSETGET，但鍵由大端序有符號 n 位元 *Integer* i 表示。
- F41D — DICTISETGETREF（$c\ i\ D\ n - D'\ c'\ -1$ 或 $D'\ 0$），使用有符號 *Integer* i 作為鍵的 DICTSETGETREF 版本。
- F41E — DICTUSETGET($x\ i\ D\ n - D'\ y\ -1$ 或 $D'\ 0$)，類似於 DICTISETGET，但 i 為無符號 n 位元整數。
- F41F — DICTUSETGETREF（$c\ i\ D\ n - D'\ c'\ -1$ 或 $D'\ 0$）。
- F422 — DICTREPLACE（$x\ k\ D\ n - D'\ -1$ 或 $D\ 0$），Replace 操作，類似於 DICTSET，但僅當鍵 k 已存在於 D 中時，才將字典 D 中鍵 k 的值設定為 x。
- F423 — DICTREPLACEREF（$c\ k\ D\ n - D'\ -1$ 或 $D\ 0$），DICTSETREF 的 Replace 對應項。
- F424 — DICTIREPLACE（$x\ i\ D\ n - D'\ -1$ 或 $D\ 0$），使用有符號 n 位元 *Integer* i 作為鍵的 DICTREPLACE 版本。
- F425 — DICTIREPLACEREF（$c\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。

- F426 — DICTUREPLACE（$x\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F427 — DICTUREPLACEREF（$c\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F42A — DICTREPLACEGET（$x\ k\ D\ n - D'\ y\ -1$ 或 $D\ 0$），DICTSETGET 的 REPLACE 對應項：成功時，也傳回與所討論的鍵關聯的舊值。
- F42B — DICTREPLACEGETREF（$c\ k\ D\ n - D'\ c'\ -1$ 或 $D\ 0$）。
- F42C — DICTIREPLACEGET（$x\ i\ D\ n - D'\ y\ -1$ 或 $D\ 0$）。
- F42D — DICTIREPLACEGETREF（$c\ i\ D\ n - D'\ c'\ -1$ 或 $D\ 0$）。
- F42E — DICTUREPLACEGET（$x\ i\ D\ n - D'\ y\ -1$ 或 $D\ 0$）。
- F42F — DICTUREPLACEGETREF（$c\ i\ D\ n - D'\ c'\ -1$ 或 $D\ 0$）。
- F432 — DICTADD（$x\ k\ D\ n - D'\ -1$ 或 $D\ 0$），DICTSET 的 ADD 對應項：將字典 D 中與鍵 k 關聯的值設定為 x，但僅當它尚未存在於 D 中時。
- F433 — DICTADDREF（$c\ k\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F434 — DICTIADD（$x\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F435 — DICTIADDREF（$c\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F436 — DICTUADD（$x\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F437 — DICTUADDREF（$c\ i\ D\ n - D'\ -1$ 或 $D\ 0$）。
- F43A — DICTADDGET（$x\ k\ D\ n - D'\ -1$ 或 $D\ y\ 0$），DICTSETGET 的 ADD 對應項：將字典 D 中與鍵 k 關聯的值設定為 x，但僅當鍵 k 尚未存在於 D 中時。否則，僅傳回舊值 y 而不更改字典。
- F43B — DICTADDGETREF（$c\ k\ D\ n - D'\ -1$ 或 $D\ c'\ 0$），DICTSETGETREF 的 ADD 對應項。
- F43C — DICTIADDGET（$x\ i\ D\ n - D'\ -1$ 或 $D\ y\ 0$）。
- F43D — DICTIADDGETREF（$c\ i\ D\ n - D'\ -1$ 或 $D\ c'\ 0$）。
- F43E — DICTUADDGET（$x\ i\ D\ n - D'\ -1$ 或 $D\ y\ 0$）。

- F43F — DICTUADDGETREF（$c\ i\ D\ n$ – $D'\ -1$ 或 $D\ c'\ 0$）。

A.10.5. 接受建構器的 SET 字典操作變體. 以下基本操作接受新值作為 *Builder* b 而非 *Slice* x，如果需要從堆疊中計算的幾個組件序列化值，這通常更方便。（這反映在相應的使用 *Slice* 的 SET 基本操作的助記符後附加 B。）淨效果大致等同於透過 ENDC; CTOS 將 b 轉換為 *Slice* 並執行 **A.10.4** 中列出的相應基本操作。

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

A.10.6. Delete 字典操作.

- F459 — DICTDEL（k D n – D' -1 或 D 0），從字典 D 中刪除由 *Slice* k 表示的 n 位元鍵。如果鍵存在，傳回修改後的字典 D' 和成功旗標 -1。否則，傳回原始字典 D 和 0。
- F45A — DICTIDEL（i D n – D' ?），DICTDEL 的版本，鍵由有符號 n 位元 *Integer* i 表示。如果 i 不適合 n 位元，僅傳回 D 0（「未找到鍵，字典未修改」）。
- F45B — DICTUDEL（i D n – D' ?），類似於 DICTIDEL，但 i 為無符號 n 位元整數。
- F462 — DICTDELGET（k D n – D' x -1 或 D 0），從字典 D 中刪除由 *Slice* k 表示的 n 位元鍵。如果鍵存在，傳回修改後的字典 D'、與鍵 k 關聯的原始值 x（由 *Slice* 表示）和成功旗標 -1。否則，傳回原始字典 D 和 0。
- F463 — DICTDELGETREF(k D n – D' c -1 或 D 0)，類似於 DICTDELGET，但成功時對 x 應用 LDREF; ENDS，以便傳回的值 c 為 *Cell*。
- F464 — DICTIDELGET（i D n – D' x -1 或 D 0），使用有符號 n 位元整數 i 作為鍵的基本操作 DICTDELGET 變體。
- F465 — DICTIDELGETREF（i D n – D' c -1 或 D 0），傳回 *Cell* 而非 *Slice* 的基本操作 DICTIDELGET 變體。
- F466 — DICTUDELGET（i D n – D' x -1 或 D 0），使用無符號 n 位元整數 i 作為鍵的基本操作 DICTDELGET 變體。
- F467 — DICTUDELGETREF（i D n – D' c -1 或 D 0），傳回 *Cell* 而非 *Slice* 的基本操作 DICTUDELGET 變體。

A.10.7. 「也許參照」字典操作. 以下操作假設字典用於儲存 $Cell^?$ 型別的值 $c^?$（「*Maybe Cell*」），這特別可用於將字典作為值儲存在其他字典中。表示如下：如果 $c^?$ 是 *Cell*，它被儲存為沒有資料位元且恰好有一個對此 *Cell* 的參照的值。如果 $c^?$ 是 *Null*，則對應的鍵必須完全不存在於字典中。

- F469 — DICTGETOPTREF（k D n – $c^?$），DICTGETREF 的變體，如果鍵 k 不存在於字典 D 中，則傳回 *Null* 而非值 $c^?$。

- F46A — DICTIGETOPTREF（$i\ D\ n$ – $c^?$），類似於 DICTGETOPTREF，但鍵由有符號 n 位元 *Integer* i 給定。如果鍵 i 超出範圍，也傳回 *Null*。
- F46B — DICTUGETOPTREF（$i\ D\ n$ – $c^?$），類似於 DICTGETOPTREF，但鍵由無符號 n 位元 *Integer* i 給定。
- F46D — DICTSETGETOPTREF（$c^?\ k\ D\ n$ – $D'\ \tilde{c}^?$），DICTGETOPTREF 和 DICTSETGETREF 的變體，將字典 D 中對應於鍵 k 的值設定為 $c^?$（如果 $c^?$ 是 *Null*，則刪除鍵），並傳回舊值 $\tilde{c}^?$（如果鍵 k 之前不存在，則傳回 *Null*）。
- F46E — DICTISETGETOPTREF（$c^?\ i\ D\ n$ – $D'\ \tilde{c}^?$），類似於基本操作 DICTSETGETOPTREF，但使用有符號 n 位元 *Integer* i 作為鍵。如果 i 不適合 n 位元，丟擲範圍檢查異常。
- F46F — DICTUSETGETOPTREF（$c^?\ i\ D\ n$ – $D'\ \tilde{c}^?$），類似於基本操作 DICTSETGETOPTREF，但使用無符號 n 位元 *Integer* i 作為鍵。

A.10.8. 前綴碼字典操作. 這些是用於建構前綴碼字典（參見 **3.4.2**）的一些基本操作。前綴碼字典的主要應用是反序列化 TL-B 序列化資料結構，或更一般地，解析前綴碼。因此，大多數前綴碼字典將是常數並在編譯時建立，而非由以下基本操作建立。

前綴碼字典的一些 GET 操作可在 **A.10.11** 中找到。其他前綴碼字典操作包括：

- F470 — PFXDICTSET（$x\ k\ D\ n$ – $D'\ -1$ 或 $D\ 0$）。
- F471 — PFXDICTREPLACE（$x\ k\ D\ n$ – $D'\ -1$ 或 $D\ 0$）。
- F472 — PFXDICTADD（$x\ k\ D\ n$ – $D'\ -1$ 或 $D\ 0$）。
- F473 — PFXDICTDEL（$k\ D\ n$ – $D'\ -1$ 或 $D\ 0$）。

這些基本操作與其非前綴碼對應項 DICTSET 等（參見 **A.10.4**）完全相似，明顯的區別是即使 SET 在前綴碼字典中也可能失敗，因此 PFXDICTSET 也必須傳回成功旗標。

A.10.9. GetNext 和 GetPrev 操作的變體.

- F474 — DICTGETNEXT（$k\ D\ n$ – $x'\ k'\ -1$ 或 0），計算字典 D 中字典順序大於 k 的最小鍵 k'，並傳回 k'（由 *Slice* 表示）以及關聯的值 x'（也由 *Slice* 表示）。

- F475 — DICTGETNEXTEQ($k\ D\ n - x'\ k'\ -1$ 或 0)，類似於 DICTGETNEXT，但計算字典順序大於或等於 k 的最小鍵 k'。
- F476 — DICTGETPREV ($k\ D\ n - x'\ k'\ -1$ 或 0)，類似於 DICTGETNEXT，但計算字典順序小於 k 的最大鍵 k'。
- F477 — DICTGETPREVEQ($k\ D\ n - x'\ k'\ -1$ 或 0)，類似於 DICTGETPREV，但計算字典順序小於或等於 k 的最大鍵 k'。
- F478 — DICTIGETNEXT ($i\ D\ n - x'\ i'\ -1$ 或 0)，類似於 DICTGETNEXT，但將字典 D 中的所有鍵解釋為大端序有符號 n 位元整數，並計算大於 *Integer* i（不一定適合 n 位元）的最小鍵 i'。
- F479 — DICTIGETNEXTEQ（$i\ D\ n - x'\ i'\ -1$ 或 0)。
- F47A — DICTIGETPREV（$i\ D\ n - x'\ i'\ -1$ 或 0)。
- F47B — DICTIGETPREVEQ（$i\ D\ n - x'\ i'\ -1$ 或 0)。
- F47C — DICTUGETNEXT ($i\ D\ n - x'\ i'\ -1$ 或 0)，類似於 DICTGETNEXT，但將字典 D 中的所有鍵解釋為大端序無符號 n 位元整數，並計算大於 *Integer* i（不一定適合 n 位元，且不一定非負）的最小鍵 i'。
- F47D — DICTUGETNEXTEQ（$i\ D\ n - x'\ i'\ -1$ 或 0)。
- F47E — DICTUGETPREV（$i\ D\ n - x'\ i'\ -1$ 或 0)。
- F47F — DICTUGETPREVEQ（$i\ D\ n - x'\ i'\ -1$ 或 0)。

A.10.10. GetMin、GetMax、RemoveMin、RemoveMax 操作.

- F482 — DICTMIN（$D\ n - x\ k\ -1$ 或 0)，計算字典 D 中的最小鍵 k（由具有 n 資料位元的 *Slice* 表示)，並傳回 k 以及關聯的值 x。
- F483 — DICTMINREF（$D\ n - c\ k\ -1$ 或 0)，類似於 DICTMIN，但將值中的唯一參照傳回為 *Cell* c。
- F484 — DICTIMIN（$D\ n - x\ i\ -1$ 或 0)，有點類似於 DICTMIN，但在假設所有鍵都是大端序有符號 n 位元整數的情況下計算最小鍵 i。請注意，傳回的鍵和值可能與 DICTMIN 和 DICTUMIN 計算的不同。
- F485 — DICTIMINREF（$D\ n - c\ i\ -1$ 或 0)。

- F486 — DICTUMIN（$D\ n - x\ i\ -1$ 或 0），類似於 DICTMIN，但將鍵傳回為無符號 n 位元 *Integer* i。
- F487 — DICTUMINREF（$D\ n - c\ i\ -1$ 或 0）。
- F48A — DICTMAX（$D\ n - x\ k\ -1$ 或 0），計算字典 D 中的最大鍵 k（由具有 n 資料位元的 *Slice* 表示），並傳回 k 以及關聯的值 x。
- F48B — DICTMAXREF（$D\ n - c\ k\ -1$ 或 0）。
- F48C — DICTIMAX（$D\ n - x\ i\ -1$ 或 0）。
- F48D — DICTIMAXREF（$D\ n - c\ i\ -1$ 或 0）。
- F48E — DICTUMAX（$D\ n - x\ i\ -1$ 或 0）。
- F48F — DICTUMAXREF（$D\ n - c\ i\ -1$ 或 0）。
- F492 — DICTREMMIN（$D\ n - D'\ x\ k\ -1$ 或 $D\ 0$），計算字典 D 中的最小鍵 k（由具有 n 資料位元的 *Slice* 表示），從字典中移除 k，並傳回 k 以及關聯的值 x 和修改後的字典 D'。
- F493 — DICTREMMINREF($D\ n - D'\ c\ k\ -1$ 或 $D\ 0$)，類似於 DICTREMMIN，但將值中的唯一參照傳回為 *Cell* c。
- F494 — DICTIREMMIN（$D\ n - D'\ x\ i\ -1$ 或 $D\ 0$），有點類似於 DICTREMMIN，但在假設所有鍵都是大端序有符號 n 位元整數的情況下計算最小鍵 i。請注意，傳回的鍵和值可能與 DICTREMMIN 和 DICTUREMMIN 計算的不同。
- F495 — DICTIREMMINREF（$D\ n - D'\ c\ i\ -1$ 或 $D\ 0$）。
- F496 — DICTUREMMIN($D\ n - D'\ x\ i\ -1$ 或 $D\ 0$)，類似於 DICTREMMIN，但將鍵傳回為無符號 n 位元 *Integer* i。
- F497 — DICTUREMMINREF（$D\ n - D'\ c\ i\ -1$ 或 $D\ 0$）。
- F49A — DICTREMMAX（$D\ n - D'\ x\ k\ -1$ 或 $D\ 0$），計算字典 D 中的最大鍵 k（由具有 n 資料位元的 *Slice* 表示），從字典中移除 k，並傳回 k 以及關聯的值 x 和修改後的字典 D'。
- F49B — DICTREMMAXREF（$D\ n - D'\ c\ k\ -1$ 或 $D\ 0$）。

- F49C — DICTIREMMAX（$D\ n$ – $D'\ x\ i\ -1$ 或 $D\ 0$）。
- F49D — DICTIREMMAXREF（$D\ n$ – $D'\ c\ i\ -1$ 或 $D\ 0$）。
- F49E — DICTUREMMAX（$D\ n$ – $D'\ x\ i\ -1$ 或 $D\ 0$）。
- F49F — DICTUREMMAXREF（$D\ n$ – $D'\ c\ i\ -1$ 或 $D\ 0$）。

A.10.11. 特殊 GET 字典和前綴碼字典操作，以及常數字典.

- F4A0 — DICTIGETJMP（$i\ D\ n$ – ），類似於 DICTIGET（參見 **A.10.12**），但成功時將 x BLESS 為繼續並後續對其執行 JMPX。失敗時，不執行任何操作。這對於實作 switch/case 建構很有用。
- F4A1 — DICTUGETJMP（$i\ D\ n$ – ），類似於 DICTIGETJMP，但執行 DICTUGET 而非 DICTIGET。
- F4A2 — DICTIGETEXEC（$i\ D\ n$ – ），類似於 DICTIGETJMP，但使用 EXECUTE 而非 JMPX。
- F4A3 — DICTUGETEXEC（$i\ D\ n$ – ），類似於 DICTUGETJMP，但使用 EXECUTE 而非 JMPX。
- F4A6_n — DICTPUSHCONST n（– $D\ n$），壓入非空常數字典 D（作為 $Cell^?$）及其鍵長度 $0 \le n \le 1023$，儲存為指令的一部分。字典本身從當前繼續的剩餘參照的第一個建立。以此方式，完整的 DICTPUSHCONST 指令可以透過首先序列化 xF4A8_，然後非空字典本身（一個 1 位元和一個單元參照），然後無符號 10 位元整數 n（就好像透過 STU 10 指令）來取得。空字典可以透過 NEWDICT 基本操作（參見 **A.10.1**）壓入。
- F4A8 — PFXDICTGETQ（$s\ D\ n$ – $s'\ x\ s''\ -1$ 或 $s\ 0$），在由 $Cell^?\ D$ 和 $0 \le n \le 1023$ 表示的前綴碼字典（參見 **3.4.2**）中查找 *Slice* s 的唯一前綴。如果找到，s 的前綴傳回為 s'，對應的值（也是 *Slice*）傳回為 x。s 的餘數傳回為 *Slice* s''。如果 s 的前綴不是前綴碼字典 D 中的鍵，則傳回未更改的 s 和零旗標以指示失敗。
- F4A9 — PFXDICTGET（$s\ D\ n$ – $s'\ x\ s''$），類似於 PFXDICTGET，但失敗時丟擲單元反序列化失敗異常。
- F4AA — PFXDICTGETJMP（$s\ D\ n$ – $s'\ s''$ 或 s），類似於 PFXDICTGETQ，但成功時將值 x BLESS 為 *Continuation* 並如同 JMPX 一樣將控制轉移給它。失敗時，傳回未更改的 s 並繼續執行。

- F4AB — PFXDICTGETEXEC（s D n – s' s''），類似於 PFXDICTGETJMP，但 EXEC 找到的繼續而非跳躍到它。失敗時，丟擲單元反序列化異常。
- F4AE_n — PFXDICTCONSTGETJMP n 或 PFXDICTSWITCH n（s – s' s'' 或 s），結合 DICTPUSHCONST n（對於 $0 \leq n \leq 1023$）與 PFXDICTGETJMP。
- F4BC — DICTIGETJMPZ（i D n – i 或無），DICTIGETJMP 的變體，失敗時傳回索引 i。
- F4BD — DICTUGETJMPZ（i D n – i 或無），DICTUGETJMP 的變體，失敗時傳回索引 i。
- F4BE — DICTIGETEXECZ（i D n – i 或無），DICTIGETEXEC 的變體，失敗時傳回索引 i。
- F4BF — DICTUGETEXECZ（i D n – i 或無），DICTUGETEXEC 的變體，失敗時傳回索引 i。

A.10.12. SubDict **字典操作.**

- F4B1 — SUBDICTGET（k l D n – D'），建構由 $HashmapE(n, X)$ 型別字典 D（具有 n 位元鍵）中所有以長度 l 的前綴 k（由 *Slice* 表示，其前 $0 \leq l \leq n \leq 1023$ 資料位元用作鍵）開頭的鍵組成的子字典。成功時，傳回與 $HashmapE(n, X)$ 相同型別的新子字典作為 *Slice* D'。
- F4B2 — SUBDICTIGET（x l D n – D'），SUBDICTGET 的變體，前綴由有符號大端序 l 位元 *Integer* x 表示，其中必然 $l \leq 257$。
- F4B3 — SUBDICTUGET（x l D n – D'），SUBDICTGET 的變體，前綴由無符號大端序 l 位元 *Integer* x 表示，其中必然 $l \leq 256$。
- F4B5 — SUBDICTRPGET（k l D n – D'），類似於 SUBDICTGET，但從新字典 D' 的所有鍵中移除公共前綴 k，它變成 $HashmapE(n - l, X)$ 型別。
- F4B6 — SUBDICTIRPGET（x l D n – D'），SUBDICTRPGET 的變體，前綴由有符號大端序 l 位元 *Integer* x 表示，其中必然 $l \leq 257$。
- F4B7 — SUBDICTURPGET（x l D n – D'），SUBDICTRPGET 的變體，前綴由無符號大端序 l 位元 *Integer* x 表示，其中必然 $l \leq 256$。
- F4BC–F4BF — 在 **A.10.11** 中由 DICT…Z 基本操作使用。

A.11 應用程式特定基本操作

操作碼範圍 F8…FB 保留給應用程式特定基本操作。當 TVM 用於執行 TON 區塊鏈智慧合約時，這些應用程式特定基本操作實際上是 TON 區塊鏈特定的。

A.11.1. 外部動作和對區塊鏈配置資料的存取. 下面列出的一些基本操作假裝產生一些外部可見的動作，例如向另一個智慧合約傳送訊息。實際上，TVM 中智慧合約的執行除了修改 TVM 狀態外從不產生任何效果。所有外部動作都被收集到儲存在特殊暫存器 c5（「輸出動作」）中的連結串列中。此外，一些基本操作使用保存在儲存於 c7（「臨時資料根」，參見 **1.3.2**）的 *Tuple* 的第一個組件中的資料。智慧合約可以自由修改保存在單元 c7 中的任何其他資料，前提是第一個參照保持完整（否則某些應用程式特定基本操作在呼叫時可能會丟擲異常）。

下面列出的大多數基本操作使用 16 位元操作碼。

A.11.2. Gas 相關基本操作. 以下基本操作中，只有前兩個是「純」的，意思是它們不使用 c5 或 c7。

- F800 — ACCEPT，將當前 gas 限制 g_l 設定為其最大允許值 g_m，並將 gas 信用 g_c 重設為零（參見 **1.4**），在此過程中將 g_r 的值減少 g_c。換句話說，當前智慧合約同意購買一些 gas 以完成當前交易。此動作是處理外部訊息所必需的，因為外部訊息本身不帶任何值（因此沒有 gas）。

- F801 — SETGASLIMIT（g – ），將當前 gas 限制 g_l 設定為 g 和 g_m 的最小值，並將 gas 信用 g_c 重設為零。如果到目前為止消耗的 gas（包括當前指令）超過 g_l 的結果值，則在設定新 gas 限制之前丟擲（未處理的）gas 不足異常。請注意，使用引數 $g \geq 2^{63} - 1$ 的 SETGASLIMIT 等同於 ACCEPT。

- F802 — BUYGAS（x – ），計算可以用 x 奈克購買的 gas 量，並以與 SETGASLIMIT 相同的方式相應地設定 g_l。

- F804 — GRAMTOGAS（x – g），計算可以用 x 奈克購買的 gas 量。如果 x 為負，傳回 0。如果 g 超過 $2^{63} - 1$，則替換為此值。

- F805 — GASTOGRAM（g – x），計算 g gas 的奈克價格。

- F806–F80E — 保留給 gas 相關基本操作。

- F80F — COMMIT（– ），提交暫存器 c4（「持久性資料」）和 c5（「動作」）的當前狀態，以便即使稍後丟擲異常，當前執行也被視為「成功」並使用儲存的值。

A.11.3. 偽隨機數產生器基本操作. 偽隨機數產生器使用隨機種子（參數 #6，參見 **A.11.4**），一個無符號 256 位元 *Integer*，以及（有時）保存在 c7 中的其他資料。在 TON 區塊鏈中執行智慧合約之前，隨機種子的初始值是智慧合約地址和全域區塊隨機種子的雜湊。如果在一個區塊內有同一智慧合約的多次執行，那麼所有這些執行都將具有相同的隨機種子。例如，這可以透過在首次使用偽隨機數產生器之前執行 LTIME; ADDRAND 來修復。

- F810 — RANDU256（– x），產生新的偽隨機無符號 256 位元 *Integer* x。演算法如下：如果 r 是隨機種子的舊值，視為 32 位元組陣列（透過建構無符號 256 位元整數的大端序表示），則計算其 $\text{SHA512}(r)$；此雜湊的前 32 位元組儲存為隨機種子的新值 r'，剩餘的 32 位元組傳回為下一個隨機值 x。

- F811 — RAND（y – z），在範圍 $0 \ldots y-1$（或 $y \ldots -1$，如果 $y < 0$）中產生新的偽隨機整數 z。更準確地說，如 RAND256U 中產生無符號隨機值 x；然後計算 $z := \lfloor xy/2^{256} \rfloor$。等同於 RANDU256; MULRSHIFT 256。

- F814 — SETRAND(x –)，將隨機種子設定為無符號 256 位元 *Integer* x。

- F815 — ADDRAND（x – ），將無符號 256 位元 *Integer* x 混合到隨機種子 r 中，透過將隨機種子設定為兩個 32 位元組字串串聯的 SHA256：第一個具有舊種子 r 的大端序表示，第二個具有 x 的大端序表示。

- F810–F81F — 保留給偽隨機數產生器基本操作。

A.11.4. 配置基本操作. 以下基本操作讀取儲存在 c7 的 *Tuple* 的第一個組件中的 *Tuple* 中提供的配置資料。每當呼叫 TVM 來執行 TON 區塊鏈智慧合約時，此 *Tuple* 由 *SmartContractInfo* 結構初始化；配置基本操作假設它保持完整。

- F82i — GETPARAM i（– x），對於 $0 \le i < 16$，從 c7 提供的 *Tuple* 傳回第 i 個參數。等同於 PUSH c7; FIRST; INDEX i。如果這些內部操作之一失敗，則丟擲適當的型別檢查或範圍檢查異常。

- F823 — NOW（– x），傳回當前 Unix 時間作為 *Integer*。如果無法從 c7 恢復請求的值，則相應地丟擲型別檢查或範圍檢查異常。等同於 GETPARAM 3。
- F824 — BLOCKLT（– x），傳回當前區塊的起始邏輯時間。等同於 GETPARAM 4。
- F825 — LTIME（– x），傳回當前交易的邏輯時間。等同於 GETPARAM 5。
- F826 — RANDSEED（– x），傳回當前隨機種子作為無符號 256 位元 *Integer*。等同於 GETPARAM 6。
- F827 — BALANCE（– t），傳回智慧合約的剩餘餘額作為 *Tuple*，由 *Integer*（奈克中的剩餘 Gram 餘額）和 *Maybe Cell*（具有 32 位元鍵的字典，表示「額外貨幣」的餘額）組成。等同於 GETPARAM 7。請注意，RAW 基本操作（例如 SENDRAWMSG）不會更新此欄位。
- F828 — MYADDR（– s），傳回當前智慧合約的內部地址作為具有 MsgAddressInt 的 *Slice*。如有必要，可以使用 PARSESTDADDR 或 REWRITESTDADDR 等基本操作進一步解析它。等同於 GETPARAM 8。
- F829 — CONFIGROOT（– D），傳回具有當前全域配置字典的 *Maybe Cell* D。等同於 GETPARAM 9。
- F830 — CONFIGDICT（– D 32），傳回全域配置字典及其鍵長度（32）。等同於 CONFIGROOT; PUSHINT 32。
- F832 — CONFIGPARAM（i – c −1 或 0），傳回具有整數索引 i 的全域配置參數的值作為 *Cell* c，以及指示成功的旗標。等同於 CONFIGDICT; DICTIGETREF。
- F833 — CONFIGOPTPARAM（i – $c^?$），傳回具有整數索引 i 的全域配置參數的值作為 *Maybe Cell* $c^?$。等同於 CONFIGDICT; DICTIGETOPTREF。
- F820—F83F — 保留給配置基本操作。

A.11.5. 全域變數基本操作. 「全域變數」可能有助於實作一些高階智慧合約語言。它們實際上儲存為 c7 的 *Tuple* 的組件：第 k 個全域變數就是此 *Tuple* 的第 k 個組件，對於 $1 \leq k \leq 254$。按慣例，第 0 個組件用於 **A.11.4** 的「配置參數」，因此它不可作為全域變數使用。

- F840 — GETGLOBVAR（$k - x$），對於 $0 \leq k < 255$，傳回第 k 個全域變數。等同於 PUSH c7; SWAP; INDEXVARQ（參見 **A.3.2**）。

- F85_k — GETGLOB k（$- x$），對於 $1 \leq k \leq 31$，傳回第 k 個全域變數。等同於 PUSH c7; INDEXQ k。

- F860 — SETGLOBVAR（x k $-$），對於 $0 \leq k < 255$，將 x 指派給第 k 個全域變數。等同於 PUSH c7; ROTREV; SETINDEXVARQ; POP c7。

- F87_k — SETGLOB k（$x -$），對於 $1 \leq k \leq 31$，將 x 指派給第 k 個全域變數。等同於 PUSH c7; SWAP; SETINDEXQ k; POP c7。

A.11.6. 雜湊和密碼學基本操作.

- F900 — HASHCU（$c - x$），計算 *Cell* c 的表示雜湊（參見 **3.1.5**）並將其傳回為 256 位元無符號整數 x。對於簽署和檢查由單元樹表示的任意實體的簽章很有用。

- F901 — HASHSU（$s - x$），計算 *Slice* s 的雜湊並將其傳回為 256 位元無符號整數 x。結果與僅包含來自 s 的資料和參照的普通單元被建立並由 HASHCU 計算其雜湊的情況相同。

- F902 — SHA256U（$s - x$），計算 *Slice* s 的資料位元的 SHA256。如果 s 的位元長度不能被八整除，丟擲單元下溢異常。雜湊值傳回為 256 位元無符號整數 x。

- F910 — CHKSIGNU（h s k $-$?），使用公鑰 k（也由 256 位元無符號整數表示）檢查雜湊 h（256 位元無符號整數，通常計算為某些資料的雜湊）的 Ed25519 簽章 s。簽章 s 必須是包含至少 512 資料位元的 *Slice*；僅使用前 512 位元。如果簽章有效，結果為 -1，否則為 0。請注意，CHKSIGNU 等同於 ROT; NEWB; STU 256; ENDB; NEWC; ROTREV; CHKSIGNS，即，等同於 CHKSIGNS，第一個引數 d 設定為包含 h 的 256 位元 *Slice*。因此，如果 h 計算為某些資料的雜湊，這些資料被雜湊兩次，第二次雜湊發生在 CHKSIGNS 內部。

- F911 — CHKSIGNS（d s k $-$?），檢查 s 是否是使用公鑰 k 對 *Slice* d 的資料部分的有效 Ed25519 簽章，類似於 CHKSIGNU。如果 *Slice* d 的位元長度不能被八整除，丟擲單元下溢異常。Ed25519 簽章的驗證是標準的，使用 SHA256 將 d 簡化為實際簽署的 256 位元數字。

- F912–F93F — 保留給雜湊和密碼學基本操作。

A.11.7. 雜項基本操作.

- F940 — CDATASIZEQ（$c\ n - x\ y\ z\ -1$ 或 0），遞迴計算以 *Cell* c 為根的 dag 中不同單元的計數 x、資料位元 y 和單元參照 z，有效地傳回考慮到相等單元的識別後此 dag 使用的總儲存空間。x、y 和 z 的值透過此 dag 的深度優先遍歷計算，使用已訪問單元雜湊的雜湊表來防止訪問已訪問的單元。已訪問單元的總計數 x 不能超過非負 *Integer* n；否則在訪問第 $(n+1)$ 個單元之前計算被中止，並傳回零以指示失敗。如果 c 是 *Null*，傳回 $x = y = z = 0$。

- F941 — CDATASIZE（$c\ n - x\ y\ z$），CDATASIZEQ 的非靜默版本，失敗時丟擲單元溢位異常（8）。

- F942 — SDATASIZEQ（$s\ n - x\ y\ z\ -1$ 或 0），類似於 CDATASIZEQ，但接受 *Slice* s 而非 *Cell*。傳回的 x 值不考慮包含切片 s 本身的單元；然而，s 的資料位元和單元參照在 y 和 z 中被計入。

- F943 — SDATASIZE（$s\ n - x\ y\ z$），SDATASIZEQ 的非靜默版本，失敗時丟擲單元溢位異常（8）。

- F944–F97F — 保留給不屬於任何其他特定類別的雜項 TON 特定基本操作。

A.11.8. 貨幣操作基本操作.

- FA00 — LDGRAMS 或 LDVARUINT16（$s - x\ s'$），從 *CellSlice* s 載入（反序列化）Gram 或 VarUInteger 16 金額，並將金額作為 *Integer* x 連同 s 的餘數 s' 一起傳回。x 的預期序列化由 4 位元無符號大端序整數 l 組成，後跟 x 的 $8l$ 位元無符號大端序表示。淨效果大約等同於 LDU 4; SWAP; LSHIFT 3; LDUX。

- FA01 — LDVARINT16（$s - x\ s'$），類似於 LDVARUINT16，但載入有符號 *Integer* x。大約等同於 LDU 4; SWAP; LSHIFT 3; LDIX。

- FA02 — STGRAMS 或 STVARUINT16（$b\ x - b'$），將範圍 $0 \ldots 2^{120} - 1$ 中的 *Integer* x 儲存（序列化）到 *Builder* b 中，並傳回結果 *Builder* b'。x 的序列化由 4 位元無符號大端序整數 l 組成，它是最小整數 $l \geq 0$，使得 $x < 2^{8l}$，後跟 x 的 $8l$ 位元無符號大端序表示。如果 x 不屬於支援的範圍，丟擲範圍檢查異常。

- FA03 — STVARINT16（$b\ x - b'$），類似於 STVARUINT16，但序列化範圍 $-2^{119} \ldots 2^{119} - 1$ 中的有符號 *Integer* x。

- FA04 — LDVARUINT32（$s - x\ s'$），從 *CellSlice* s 載入（反序列化）VarUInteger 32，並將反序列化的值傳回為 *Integer* $0 \leq x < 2^{248}$。x 的預期序列化由 5 位元無符號大端序整數 l 組成，後跟 x 的 $8l$ 位元無符號大端序表示。淨效果大約等同於 LDU 5; SWAP; SHIFT 3; LDUX。
- FA05 — LDVARINT32（$s - x\ s'$），從 *CellSlice* s 反序列化 VarInteger 32，並將反序列化的值傳回為 *Integer* $-2^{247} \leq x < 2^{247}$。
- FA06 — STVARUINT32（$b\ x - b'$），將 *Integer* $0 \leq x < 2^{248}$ 序列化為 VarUInteger 32。
- FA07 — STVARINT32（$b\ x - b'$），將 *Integer* $-2^{247} \leq x < 2^{247}$ 序列化為 VarInteger 32。
- FA08–FA1F — 保留給貨幣操作基本操作。

A.11.9. 訊息和地址操作基本操作. 下面列出的訊息和地址操作基本操作根據以下 TL-B 方案（參見 **3.3.4**）序列化和反序列化值：

```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 9) external_address:(bits len)
             = MsgAddressExt;
anycast_info$_ 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;
```

反序列化的 MsgAddress 由 *Tuple* t 表示如下：

- addr_none 由 $t = (0)$ 表示，即，包含恰好一個等於零的 *Integer* 的 *Tuple*。
- addr_extern 由 $t = (1, s)$ 表示，其中 *Slice* s 包含欄位 external_address。換句話說，t 是一對（由兩個項目組成的 *Tuple*），包含一個等於一的 *Integer* 和 *Slice* s。
- addr_std 由 $t = (2, u, x, s)$ 表示，其中 u 要麼是 *Null*（如果 anycast 不存在）要麼是包含 rewrite_pfx 的 *Slice* s'（如果 anycast 存在）。接下來，*Integer* x 是 workchain_id，*Slice* s 包含 address。
- addr_var 由 $t = (3, u, x, s)$ 表示，其中 u、x 和 s 與 addr_std 具有相同的含義。

定義了以下使用上述慣例的基本操作：

- FA40 — LDMSGADDR（$s - s'\ s''$），從 *CellSlice* s 載入唯一作為有效 MsgAddress 的前綴，並將此前綴 s' 和 s 的餘數 s'' 都傳回為 *CellSlice*。
- FA41 — LDMSGADDRQ（$s - s'\ s''\ -1$ 或 $s\ 0$），LDMSGADDR 的靜默版本：成功時，壓入額外的 -1；失敗時，壓入原始 s 和零。
- FA42 — PARSEMSGADDR($s - t$)，將包含有效 MsgAddress 的 *CellSlice* s 分解為具有此 MsgAddress 的獨立欄位的 *Tuple* t。如果 s 不是有效的 MsgAddress，丟擲單元反序列化異常。
- FA43 — PARSEMSGADDRQ（$s - t\ -1$ 或 0），PARSEMSGADDR 的靜默版本：錯誤時傳回零而非丟擲異常。
- FA44 — REWRITESTDADDR（$s - x\ y$），解析包含有效 MsgAddressInt（通常是 msg_addr_std）的 *CellSlice* s，將 anycast（如果存在）的重寫應用於地址的相同長度前綴，並將 workchain x 和 256 位元地址 y 都傳回為 *Integer*。如果地址不是 256 位元，或如果 s 不是 MsgAddressInt 的有效序列化，丟擲單元反序列化異常。
- FA45 — REWRITESTDADDRQ($s - x\ y\ -1$ 或 0)，基本操作 REWRITESTDADDR 的靜默版本。
- FA46 — REWRITEVARADDR（$s - x\ s'$），REWRITESTDADDR 的變體，將（重寫的）地址傳回為 *Slice* s，即使它不正好是 256 位元長（由 msg_addr_var 表示）。

- FA47 — REWRITEVARADDRQ($s - x\ s'\ -1$ 或 0)，基本操作 REWRITEVARADDR 的靜默版本。

- FA48–FA5F — 保留給訊息和地址操作基本操作。

A.11.10. 出站訊息和輸出動作基本操作.

- FB00 — SENDRAWMSG（$c\ x\ -$），傳送包含在 *Cell* c 中的原始訊息，它應該包含正確序列化的物件 Message X，唯一的例外是來源地址允許具有虛擬值 addr_none（將自動替換為當前智慧合約地址），而 ihr_fee、fwd_fee、created_lt 和 created_at 欄位可以具有任意值（在當前交易的動作階段期間將重寫為正確值）。整數參數 x 包含旗標。目前 $x = 0$ 用於普通訊息；$x = 128$ 用於攜帶當前智慧合約的所有剩餘餘額的訊息（而非訊息中原本指示的值）；$x = 64$ 用於除訊息中最初指示的值外還攜帶入站訊息的所有剩餘值的訊息（如果未設定位元 0，則從此金額中扣除 gas 費用）；$x' = x + 1$ 表示傳送者希望單獨支付轉帳費用；$x' = x + 2$ 表示在動作階段處理此訊息時產生的任何錯誤都應被忽略。最後，$x' = x + 32$ 表示如果當前帳戶的結果餘額為零，則必須銷毀該帳戶。此旗標通常與 $+128$ 一起使用。

- FB02 — RAWRESERVE（$x\ y\ -$），建立一個輸出動作，它將從帳戶的剩餘餘額中保留恰好 x 奈克（如果 $y = 0$）、最多 x 奈克（如果 $y = 2$）或除 x 奈克外的所有（如果 $y = 1$ 或 $y = 3$）。它大致等同於建立一個攜帶 x 奈克（或 $b - x$ 奈克，其中 b 是剩餘餘額）給自己的出站訊息，以便後續輸出動作無法花費超過餘數的金額。y 中的位元 $+2$ 表示如果無法保留指定金額，外部動作不會失敗；相反，保留所有剩餘餘額。y 中的位元 $+8$ 表示在執行任何進一步動作之前 $x \leftarrow -x$。y 中的位元 $+4$ 表示在執行任何其他檢查和動作之前，x 增加當前帳戶的原始餘額（在計算階段之前），包括所有額外貨幣。目前 x 必須是非負整數，y 必須在範圍 $0 \ldots 15$ 內。

- FB03 — RAWRESERVEX（$x\ D\ y\ -$），類似於 RAWRESERVE，但也接受字典 D（由 *Cell* 或 *Null* 表示）與額外貨幣。以此方式可以保留 Gram 以外的貨幣。

- FB04 — SETCODE（$c\ -$），建立一個輸出動作，將此智慧合約代碼更改為 *Cell* c 給出的代碼。請注意，此更改僅在當前智慧合約執行成功終止後生效。

- FB06 — SETLIBCODE（$c\ x\ -$），建立一個輸出動作，透過新增或刪除 *Cell* c 中給定的代碼庫來修改此智慧合約庫的集合。如果 $x = 0$，則

如果庫之前存在於集合中，則實際刪除該庫（如果不存在，此動作不執行任何操作）。如果 $x = 1$，則將庫新增為私有庫，如果 $x = 2$，則將庫新增為公共庫（如果當前智慧合約駐留在主鏈中，則可供所有智慧合約使用）；如果庫之前存在於集合中，則其公共/私有狀態根據 x 更改。x 的值不是 $0 \ldots 2$ 無效。

- FB07 — CHANGELIB（h x –），類似於 SETLIBCODE 建立輸出動作，但接受庫雜湊作為無符號 256 位元整數 h 而非庫代碼。如果 $x \neq 0$ 且具有雜湊 h 的庫不在此智慧合約的庫集合中，則此輸出動作將失敗。
- FB08–FB3F — 保留給輸出動作基本操作。

A.12 Debug 基本操作

以 FE 開頭的操作碼保留給 *debug* 基本操作。這些基本操作具有已知的固定操作長度，並且表現為（多位元組）NOP 操作。特別是，它們從不更改堆疊內容，並且從不丟擲異常，除非沒有足夠的位元來完全解碼操作碼。但是，當在啟用 debug 模式的 TVM 實例中呼叫時，這些基本操作可以將特定輸出產生到 TVM 實例的文字 debug 日誌中，從不影響 TVM 狀態（因此從 TVM 的角度來看，debug 模式下 debug 基本操作的行為與非 debug 模式完全相同）。例如，debug 基本操作可能傾印堆疊頂部附近的全部或部分值，顯示 TVM 的當前狀態等。

A.12.1. 作為多位元組 NOP 的 Debug 基本操作.

- FEnn — DEBUG nn，對於 $0 \leq nn < 240$，是雙位元組 NOP。
- FEF$nssss$ — DEBUGSTR $ssss$，對於 $0 \leq n < 16$，是 $(n + 3)$ 位元組 NOP，$(n + 1)$ 位元組「內容字串」$ssss$ 也被跳過。

A.12.2. 作為無副作用操作的 Debug 基本操作. 接下來我們描述可能（並且實際上）在 TVM 版本中實作的 debug 基本操作。請注意，另一個 TVM 實作可以自由地將這些代碼用於其他 debug 目的，或將它們視為多位元組 NOP。每當這些基本操作需要來自堆疊的一些引數時，它們檢查這些引數，但將它們保持在堆疊中。如果堆疊中的值不足，或它們具有不正確的型別，debug 基本操作可能會將錯誤訊息輸出到 debug 日誌，或表現為 NOP，但它們不能丟擲異常。

- FE00 — DUMPSTK，傾印堆疊（最多頂端 255 個值）並顯示總堆疊深度。

- FE0n — DUMPSTKTOP n，$1 \leq n < 15$，從堆疊傾印頂端 n 個值，從最深的開始。如果有 $d < n$ 個值可用，僅傾印 d 個值。
- FE10 — HEXDUMP，以十六進位形式傾印 s0，無論它是 *Slice* 還是 *Integer*。
- FE11 — HEXPRINT，類似於 HEXDUMP，除了 s0 的十六進位表示不會立即輸出，而是串聯到輸出文字緩衝區。
- FE12 — BINDUMP，以二進位形式傾印 s0，類似於 HEXDUMP。
- FE13 — BINPRINT，將 s0 的二進位表示輸出到文字緩衝區。
- FE14 — STRDUMP，將 s0 處的 *Slice* 傾印為 UTF-8 字串。
- FE15 — STRPRINT，類似於 STRDUMP，但將字串輸出到文字緩衝區(不帶回車)。
- FE1E — DEBUGOFF，停用所有 debug 輸出，直到透過 DEBUGON 重新啟用。更準確地說，此基本操作增加內部計數器，當嚴格為正時停用所有 debug 操作（除了 DEBUGOFF 和 DEBUGON）。
- FE1F — DEBUGON，啟用 debug 輸出（在 TVM 的 debug 版本中）。
- FE2n — DUMP s(n)，$0 \leq n < 15$，傾印 s(n)。
- FE3n — PRINT s(n)，$0 \leq n < 15$，將 s(n) 的文字表示（不帶任何前導或尾隨空格或回車）串聯到文字緩衝區，該緩衝區將在任何其他 debug 操作的輸出之前輸出。
- FEC0--FEEF — 將這些操作碼用於自訂/實驗性 debug 操作。
- FEFn*ssss* — DUMPTOSFMT *ssss*，根據 $(n+1)$ 位元組字串 *ssss* 格式化傾印 s0。此字串可能包含除錯器支援的 TL-B 型別名稱的（前綴）。如果字串以零位元組開始，僅將其（不帶第一個位元組）輸出到 debug 日誌。如果字串以等於一的位元組開始，將其串聯到緩衝區，該緩衝區將在任何其他 debug 操作的輸出之前輸出（有效地輸出不帶回車的字串）。
- FEFn00*ssss* — LOGSTR *ssss*，字串 *ssss* 長度為 n 位元組。
- FEF000 — LOGFLUSH，將所有待處理的 debug 輸出從緩衝區刷新到 debug 日誌。

- FEFn01$ssss$ — PRINTSTR $ssss$，字串 $ssss$ 長度為 n 位元組。

A.13 代碼頁基本操作

以下以位元組 FF 開頭的基本操作通常在智慧合約代碼或庫子程式的最開始使用以選擇另一個 TVM 代碼頁。請注意，我們期望所有代碼頁包含具有相同代碼的這些基本操作，否則切換回另一個代碼頁可能是不可能的（參見 **5.1.8**）。

- FFnn — SETCP nn，選擇 TVM 代碼頁 $0 \leq nn < 240$。如果不支援代碼頁，丟擲無效操作碼異常。
- FF00 — SETCP0，選擇本文件中描述的 TVM（測試）代碼頁零。
- FFFz — SETCP $z - 16$，對於 $1 \leq z \leq 15$，選擇 TVM 代碼頁 $z - 16$。負代碼頁 $-13 \ldots -1$ 保留給驗證其他代碼頁中 TVM 執行所需的 TVM 限制版本，如 **B.2.6** 中所述。負代碼頁 -14 保留給實驗性代碼頁，不同 TVM 實作之間不一定相容，應在 TVM 的生產版本中停用。
- FFF0 — SETCPX（c –），選擇在堆疊頂端傳遞的 $-2^{15} \leq c < 2^{15}$ 的代碼頁 c。

B TVM 的正式屬性和規範

本附錄討論在 TON 區塊鏈中執行智慧合約並在之後驗證此類執行所必需的 TVM 的某些正式屬性。

B.1 TVM 狀態的序列化

回想一下，用於在區塊鏈中執行智慧合約的虛擬機器必須是確定性的，否則每次執行的驗證將需要將所有執行的中間步驟包含到區塊中，或至少包含執行不確定性操作時做出的選擇。

此外，此類虛擬機器的狀態 必須是（唯一）可序列化的，以便即使狀態本身通常不包含在區塊中，其雜湊 仍然是明確定義的，並且可以包含在區塊中以進行驗證。

B.1.1. TVM 堆疊值. TVM 堆疊值可以序列化如下：

```
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;
```

其中，`vm_stk_tinyint` 在代碼頁零中從未由 TVM 使用；它僅在受限模式中使用。

B.1.2. TVM 堆疊. TVM 堆疊可以序列化如下：

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

B.1.3. TVM 控制暫存器. TVM 中的控制暫存器可以序列化如下：

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

B.1.4. TVM Gas 限制. TVM 中的 Gas 限制可以序列化如下：

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

B.1.5. TVM 程式庫環境. TVM 程式庫環境可以序列化如下：

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

B.1.6. TVM 繼續. TVM 中的繼續可以序列化如下：

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

B.1.7. TVM 狀態. TVM 的總狀態可以序列化如下：

```
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;
```

當 TVM 初始化時，其狀態由 `vms_init` 描述，通常將 `step` 設定為零。TVM 的步進函數對 `vms_finished` 狀態不執行任何操作，並將所有其他狀態轉換為 `vms_running`、`vms_exception` 或 `vms_finished`，並將 `step` 增加一。

B.2 TVM 的步進函數

TVM 的正式規範將由步進函數 $f : \mathit{VmState} \to \mathit{VmState}$ 的定義來完成。此函數確定性地將有效的 VM 狀態轉換為有效的後續 VM 狀態，並且如果原始狀態無效，則允許丟擲異常或傳回無效的後續狀態。

B.2.1. 步進函數的高層級定義. 我們可能會用高層級的函數式編程語言呈現 TVM 步進函數的非常長的正式定義。然而，這樣的規範主要對（人類）開發者有用作為參考。我們選擇了另一種更適合計算機自動化正式驗證的方法。

B.2.2. 步進函數的操作性定義. 請注意，步進函數 f 是從單元樹到單元樹的明確定義的可計算函數。因此，它可以由通用圖靈機計算。然後，在這樣的機器上計算 f 的程式 P 將提供步進函數 f 的機器可檢查規範。此程式 P 實際上是此圖靈機上 TVM 的模擬器。

B.2.3. TVM 內部的 TVM 模擬器參考實作. 我們看到 TVM 的步進函數可以由另一台機器上 TVM 模擬器的參考實作來定義。一個顯而易見的想法是使用 TVM 本身，因為它非常適合處理單元樹。然而，如果我們對 TVM 的特定實作有疑慮並想要檢查它，那麼 TVM 內部的自身模擬器並不是很有用。例如，如果這樣的模擬器通過呼叫 `DICTISET` 指令本身來解釋該指令，那麼 TVM 底層實作中的錯誤將仍然不會被注意到。

B.2.4. 最小版本 TVM 內的參考實作. 我們看到使用 TVM 本身作為 TVM 模擬器參考實作的主機機器不會產生什麼洞察。一個更好的想法是定義一個精簡版的 *TVM*，它僅支援最少的基本操作和 64 位元整數算術，並為此精簡版的 TVM 提供 TVM 步進函數 f 的參考實作 P。

在這種情況下，只需仔細實作和檢查少數基本操作即可獲得精簡版的 TVM，並將在此精簡版上運行的參考實作 P 與正在驗證的完整自訂 TVM 實作進行比較。特別是，如果對自訂 TVM 實作的特定執行的有效性有任何疑慮，現在可以在參考實作的幫助下輕鬆解決。

B.2.5. 與 TON 區塊鏈的相關性. TON 區塊鏈採用這種方法來驗證 TVM 的執行（例如，那些用於智慧合約處理入站訊息的執行），當驗證者的結果

彼此不匹配時。在這種情況下，儲存在主鏈中作為可配置參數的 TVM 參考實作（從而定義 TVM 的當前版本）用於獲得正確的結果。

B.2.6. 代碼頁 −1. TVM 的代碼頁 −1 保留給精簡版的 TVM。其主要目的是執行完整 TVM 步進函數的參考實作。此代碼頁僅包含使用「微整數」(64 位元有符號整數) 的算術基本操作的特殊版本；因此，TVM 的 257 位元 *Integer* 算術必須根據 64 位元算術來定義。橢圓曲線密碼學基本操作也直接在代碼頁 −1 中實作，而不使用任何第三方程式庫。最後，SHA256 雜湊函數的參考實作也在代碼頁 −1 中提供。

B.2.7. 代碼頁 −2. 這個引導過程可以進一步迭代，通過為僅支援布林值 (或整數 0 和 1) 的更簡單版本的 TVM——「代碼頁 −2」——編寫精簡版 TVM 的模擬器。然後，代碼頁 −1 中使用的所有 64 位元算術都需要通過布林操作來定義，從而為代碼頁 −1 中使用的精簡版 TVM 提供參考實作。這樣，如果一些 TON 區塊鏈驗證者對其 64 位元算術的結果不一致，他們可以回歸到此參考實作以找到正確答案。[30]

[30]TVM 的初步版本不使用代碼頁 −2 來實現此目的。這在未來可能會改變。

C 堆疊和暫存器機器的代碼密度

本附錄擴展了 **2.2** 中提供的堆疊操作基本操作的一般考量，解釋了 TVM 對這類基本操作的選擇，並比較了堆疊機器和暫存器機器在使用的基本操作數量和代碼密度方面的差異。我們通過比較優化編譯器為相同原始檔案生成的機器代碼，針對不同（抽象的）堆疊和暫存器機器來做到這一點。

結果表明，堆疊機器（至少那些配備了 **2.2.1** 中描述的基本堆疊操作基本操作的機器）具有遠優越的代碼密度。此外，堆疊機器在其他算術和任意資料處理操作方面具有出色的可擴展性，特別是如果考慮到由優化編譯器自動生成的機器代碼。

C.1 樣本葉函數

我們首先比較由（假想的）優化編譯器為幾個抽象暫存器和堆疊機器生成的機器代碼，這些機器代碼對應於包含葉函數（即不呼叫任何其他函數的函數）定義的相同高層級語言原始碼。對於暫存器機器和堆疊機器，我們遵循 **2.1** 中引入的表示法和慣例。

C.1.1. 葉函數的樣本原始檔案. 我們考慮的原始檔案包含一個函數 f，它接受六個（整數）參數 a、b、c、d、e、f，並傳回兩個（整數）值 x 和 y，它們是以下兩個線性方程組的解：

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

該函數的原始碼，在類似於 C 的編程語言中，可能如下所示：

```
(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);
}
```

我們假設（參見 **2.1**）我們考慮的暫存器機器在暫存器 `r0`…`r5` 中接受六個參數 $a…f$，並在 `r0` 和 `r1` 中傳回兩個值 x 和 y。我們還假設暫存器機器有 16 個暫存器，並且堆疊機器可以通過其堆疊操作基本操作直接存取 `s0` 到 `s15`；堆疊機器將在 `s5` 到 `s0` 中接受參數，並在 `s0` 和 `s1` 中傳回兩個值，這與暫存器機器有些相似。最後，我們最初假設允許暫存器機器破壞所有暫存器中的值（這對堆疊機器稍微不公平）；稍後將重新檢視此假設。

C.1.2. 三位址暫存器機器. 三位址暫存器機器（參見 **2.1.7**）的機器代碼（或者更確切地說是相應的組合語言代碼）可能如下所示：

```
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
```

我們使用了 12 個操作和至少 23 個位元組（每個操作使用 $3 \times 4 = 12$ 位元來指示涉及的三個暫存器，以及至少 4 位元來指示執行的操作；因此我們需要兩或三個位元組來編碼每個操作）。更實際的估計將是 34 個位元組（每個算術操作三個位元組）或 31 個位元組（加法和減法兩個位元組，乘法和除法三個位元組）。

C.1.3. 雙位址暫存器機器. 雙位址暫存器機器的機器代碼可能如下所示：

```
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
```

我們使用了 16 個操作；樂觀地假設每個操作（除了 RET）都可以用兩個位元組編碼，此代碼將需要 31 個位元組。[31]

C.1.4. 單位址暫存器機器. 單位址暫存器機器的機器代碼可能如下所示：

```
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
```

我們使用了 23 個操作；如果我們假設所有算術操作和 XCHG 都是單位元組編碼，而 MOV 是雙位元組編碼，則代碼的總大小將是 29 個位元組。但是請注意，為了獲得上面顯示的緊湊代碼，我們必須選擇特定的計算順序，並大量使用乘法的交換律。（例如，我們在 af 之前計算 bc，並在 af 之後立即計算 $af - bc$。）編譯器是否能夠自行進行所有這些優化尚不清楚。

[31]將此代碼與優化 C 編譯器為 x86-64 架構生成的代碼進行比較是很有趣的。

首先，x86-64 的整數除法操作使用單位址形式，雙長度的被除數必須在累加器對 r2:r0 中提供。商也在 r0 中傳回。因此，需要添加兩個單到雙擴展操作（CDQ 或 CQO）和至少一個移動操作。

其次，算術和移動操作使用的編碼不如我們上面的範例樂觀，平均每個操作需要約三個位元組。結果，我們獲得 32 位元整數總共 43 個位元組，64 位元整數 68 個位元組。

C.1.5. 具有基本堆疊基本操作的堆疊機器. 配備 **2.2.1** 中描述的基本堆疊操作基本操作的堆疊機器的機器代碼可能如下所示：

```
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
```

我們使用了 29 個操作；假設所有涉及的堆疊操作（包括 `XCHG s1,s`(i)）都是單位元組編碼，我們也使用了 29 個代碼位元組。請注意，對於單位元組編碼，「非系統化」操作 `ROT`（等同於 `XCHG s1; XCHG s2`）將操作和位元組計數減少到 28。這表明從 Forth 借用的此類「非系統化」操作在某些情況下確實可以減少代碼大小。

還請注意，我們在此代碼中隱式使用了乘法的交換律，計算 $de - bf$ 而不是高層級語言原始碼中指定的 $ed - bf$。如果我們不允許這樣做，則需要在第三個 IMUL 之前插入一個額外的 XCHG s1，將代碼的總大小增加一個操作和一個位元組。

上面呈現的代碼可能是由相當簡單的編譯器產生的，該編譯器只是按照它們出現的順序計算所有表達式和子表達式，然後在每個操作之前重新排列堆疊頂部附近的參數，如 **2.2.2** 中所述。這裡唯一的「手動」優化涉及在 af 之前計算 ec；可以檢查另一個順序將導致稍短的 28 個操作和位元組的代碼（或 29 個，如果我們不允許使用乘法的交換律），但 ROT 優化將不適用。

C.1.6. 具有複合堆疊基本操作的堆疊機器. 具有複合堆疊基本操作的堆疊機器（參見 **2.2.3**）不會顯著改進上面呈現的代碼的代碼密度，至少在使用的位元組方面。唯一的區別是，如果我們不允許使用乘法的交換律，則在第三個 IMUL 之前插入的額外 XCHG s1 可能會與之前的兩個操作 XCHG s3、PUSH s2 組合成一個複合操作 PUXC s2,s3；我們在下面提供生成的代碼。為了使這不那麼冗餘，我們展示一個代碼版本，按照原始原始檔案中指定的那樣，在 ec 之前計算子表達式 af。我們看到這將六個操作（從第 15 行開始）替換為五個其他操作，並禁用了 ROT 優化：

```
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
```

我們總共使用了 27 個操作和 28 個位元組，與之前的版本（使用 ROT 優化）相同。但是，我們在這裡沒有使用乘法的交換律，因此我們可以說複合堆疊操作基本操作使我們能夠將代碼大小從 29 個位元組減少到 28 個位元組。

再次請注意，上述代碼可能是由簡單的編譯器生成的。手動優化可能會導致更緊湊的代碼；例如，我們可以使用複合操作（例如 XCHG3）來不僅為下一個算術操作預先準備 s0 和 s1 的正確值，還為之後的算術操作準備 s2 的值。下一節提供了這種優化的範例。

C.1.7. 具有複合堆疊基本操作和手動優化代碼的堆疊機器. 具有複合堆疊基本操作的堆疊機器的先前代碼版本可以手動優化如下。

通過盡可能與之前的 XCHG、PUSH 和算術操作交換 XCHG 操作，我們獲得代碼片段 XCHG s2,s6; XCHG s1,s0; XCHG s0,s5，然後可以用複合操作 XCHG3 s6,s0,s5 替換。此複合操作將允許雙位元組編碼，從而導致僅使用 21 個操作的 27 位元組代碼：

```
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
```

有趣的是，此版本的堆疊機器代碼僅包含 9 個堆疊操作基本操作用於 11 個算術操作。然而，不清楚優化編譯器是否能夠自行以這種方式重新組織代碼。

C.2 樣本葉函數的機器代碼比較

表 **1** 總結了對應於 **C.1.1** 中描述的相同原始檔案的機器代碼的屬性，為假想的三位址暫存器機器（參見 **C.1.2**）生成，具有「樂觀」和「實際」指令編碼；雙位址機器（參見 **C.1.3**）；單位址機器（參見 **C.1.4**）；以及類似於 TVM 的堆疊機器，僅使用基本堆疊操作基本操作（參見 **C.1.5**）或同時使用基本和複合堆疊基本操作（參見 **C.1.7**）。

表 **1** 中的列的含義如下：

- 「操作」——使用的指令數量，分為「資料」（即暫存器機器的暫存器移動和交換指令，以及堆疊機器的堆疊操作指令）和「算術」（用於加、減、乘和除整數的指令）。「總計」比這兩者的總和多一個，因為機器代碼末尾還有一個單位元組的 RET 指令。

- 「代碼位元組」——使用的代碼位元組總量。

- 「操作碼空間」——假定指令編碼中由資料和算術指令使用的「操作碼空間」（即指令編碼的第一個位元組的可能選擇）的部分。例如，三位址機器的「樂觀」編碼假設所有算術指令 op r(i), r(j), r(k) 都是雙位元組編碼。然後每個算術指令將消耗 $16/256 = 1/16$ 的操作碼空間。請注意，對於堆疊機器，我們在所有情況下都假設 XCHG s(i)、PUSH s(i) 和 POP s(i) 是單位元組編碼，僅在基本堆疊指令情況下還增加了 XCHG s1,s(i)。至於複合堆疊操作，我們假設 PUSH3、XCHG3、XCHG2、XCPU、PUXC、PUSH2 是雙位元組編碼，但 XCHG s1,s(i) 不是。

	操作			代碼位元組			操作碼空間		
機器	資料	算術	總計	資料	算術	**總計**	資料	**算術**	總計
3-位址（樂觀）	0	11	12	0	22	**23**	0/256	**64/256**	65/256
3-位址（實際）	0	11	12	0	30	**31**	0/256	**34/256**	35/256
2-位址	4	11	16	8	22	**31**	1/256	**4/256**	6/256
1-位址	11	11	23	17	11	**29**	17/256	**64/256**	82/256
堆疊（基本）	16	11	28	16	11	**28**	64/256	**4/256**	69/256
堆疊（複合）	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 1: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本葉函數生成（參見 **C.1.1**）。反映**代碼密度** 和**可擴展性** 到其他操作的兩個最重要的列以粗體字標記。這兩列中較小的值更好。

「代碼位元組」列反映了特定樣本原始碼的代碼密度。但是，「操作碼空間」也很重要，因為它反映了將實現的密度擴展到其他類別操作（例如，如果要用字串操作等補充算術操作）的可擴展性。這裡「算術」子列比「資料」子列更重要，因為這種擴展不需要進一步的資料操作操作。

我們看到，假設所有三暫存器算術操作都是雙位元組編碼的「樂觀」編碼的三位址暫存器機器實現了最佳代碼密度，僅需要 23 個位元組。然而，這是有代價的：每個算術操作消耗 1/16 的操作碼空間，因此四個操作已經使用了操作碼空間的四分之一。在保留如此高的代碼密度的同時，最多可以向此架構添加 11 個其他操作，無論是算術操作還是非算術操作。另一方面，當我們考慮三位址機器的「實際」編碼時，僅對最常用的加法/減法操作使用雙位元組編碼（對於較少使用的乘法/除法操作使用較長的編碼，反映了可能的擴展操作可能屬於此類的事實），那麼三位址機器不再提供如此吸引人的代碼密度。

實際上，雙位址機器此時變得同樣有吸引力：它能夠實現與「實際」編碼的三位址機器相同的 31 個位元組的代碼大小，僅使用 6/256 的操作碼空間！然而，31 個位元組是此表中最差的結果。

單位址機器使用 29 個位元組，比雙位址機器略少。然而，它使用四分之一的操作碼空間用於其算術操作，妨礙了其可擴展性。在這方面，它類似於「樂觀」編碼的三位址機器，但需要 29 個位元組而不是 23 個！因此，就可擴展性（由算術操作使用的操作碼空間反映）與代碼密度相比，完全沒有理由使用單位址機器。

最後，堆疊機器在代碼密度方面贏得了競爭（27 或 28 個位元組），僅輸給「樂觀」編碼的三位址機器（然而，在可擴展性方面卻很糟糕）。

總結：雙位址機器和堆疊機器在其他算術或資料處理指令方面實現了最佳可擴展性（每個這樣的指令僅使用 1/256 的代碼空間），而堆疊機器還以微弱優勢實現了最佳代碼密度。堆疊機器使用其代碼空間的很大一部分

(超過四分之一)用於資料(即堆疊)操作指令;然而,這並不嚴重妨礙可擴展性,因為堆疊操作指令佔據操作碼空間的恆定部分,無論所有其他指令和擴展如何。

雖然可能仍然傾向於使用雙位址暫存器機器,但我們將很快解釋(參見 **C.3**)為什麼雙位址暫存器機器在實踐中提供的代碼密度和可擴展性比此表所顯示的更差。

至於在僅具有基本堆疊操作基本操作的堆疊機器或同時支援複合堆疊基本操作的堆疊機器之間進行選擇,更複雜的堆疊機器的情況似乎較弱:它僅提供一或兩個更少的代碼位元組,代價是使用相當多的操作碼空間用於堆疊操作,並且使用這些額外指令的優化代碼對於程式設計師來說很難編寫,對於編譯器來說也很難自動生成。

C.2.1. 暫存器呼叫慣例:某些暫存器必須由函數保留. 到目前為止,我們僅考慮了一個函數的機器代碼,而沒有考慮此函數與同一程式中其他函數之間的相互作用。

通常一個程式由多個函數組成,當一個函數不是「簡單」或「葉」函數時,它必須呼叫其他函數。因此,被呼叫的函數是否保留所有或至少某些暫存器變得很重要。如果它保留除用於傳回結果的暫存器之外的所有暫存器,則呼叫者可以安全地將其本地和臨時變數保存在某些暫存器中;然而,被呼叫者需要將它將用於其臨時值的所有暫存器保存到某處(通常保存到堆疊中,堆疊也存在於暫存器機器上),然後恢復原始值。另一方面,如果允許被呼叫的函數破壞所有暫存器,則可以以 **C.1.2**、**C.1.3** 和 **C.1.4** 中描述的方式編寫它,但呼叫者現在將負責在呼叫之前將其所有臨時值保存到堆疊中,並在之後恢復這些值。

在大多數情況下,暫存器機器的呼叫慣例要求保留某些但不是所有暫存器。我們將假設 $m \leq n$ 個暫存器將由函數保留(除非它們用於傳回值),並且這些暫存器是 $\mathtt{r}(n-m) \ldots \mathtt{r}(n-1)$。情況 $m = 0$ 對應於迄今為止考慮的「被呼叫者可以自由破壞所有暫存器」的情況;這對呼叫者來說相當痛苦。情況 $m = n$ 對應於「被呼叫者必須保留所有暫存器」的情況;正如我們稍後將看到的,這對被呼叫者來說相當痛苦。在實踐中通常使用 m 約為 $n/2$ 的值。

以下各節考慮我們具有 $n = 16$ 個暫存器的暫存器機器的 $m = 0$、$m = 8$ 和 $m = 16$ 的情況。

C.2.2. 情況 $m = 0$:沒有要保留的暫存器. 此情況已在 **C.2** 和表 **1** 中考慮並總結。

C.2.3. 情況 $m = n = 16$:必須保留所有暫存器. 此情況對被呼叫的函數來說是最痛苦的。對於像我們一直考慮的葉函數來說特別困難,它們完全不

機器	r	操作 資料	操作 算術	操作 總計	代碼位元組 資料	代碼位元組 算術	代碼位元組 **總計**	操作碼空間 資料	操作碼空間 **算術**	操作碼空間 總計
3-位址（樂觀）	*4*	8	11	20	8	22	**31**	32/256	**64/256**	97/256
3-位址（實際）	*4*	8	11	20	8	30	**39**	32/256	**34/256**	67/256
2-位址	*5*	14	11	26	18	22	**41**	33/256	**4/256**	38/256
1-位址	*6*	23	11	35	29	11	**41**	49/256	**64/256**	114/256
堆疊（基本）	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
堆疊（複合）	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 2: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本葉函數生成（參見 **C.1.1**），假設被呼叫的函數必須保留所有 16 個暫存器（$m = n = 16$）。標記為 r 的新列表示要保存和恢復的暫存器數量，與表 **1** 相比導致 $2r$ 個更多的操作和代碼位元組。暫存器機器的新增 `PUSH` 和 `POP` 指令也使用 32/256 的操作碼空間。對應於堆疊機器的兩行保持不變。

受益於其他函數在被呼叫時保留某些暫存器這一事實——它們不呼叫任何函數，但必須自己保留所有暫存器。

為了估計假設 $m = n = 16$ 的後果，我們將假設我們所有的暫存器機器都配備了堆疊，以及單位元組指令 `PUSH r`(i) 和 `POP r`(i)，它們將暫存器推入/彈出堆疊。例如， **C.1.2** 中提供的三位址機器代碼破壞了暫存器 `r2`、`r3`、`r6` 和 `r7` 中的值；這意味著此函數的代碼必須在開始處增加四個指令 `PUSH r2`; `PUSH r3`; `PUSH r6`; `PUSH r7`，並在 `RET` 指令之前增加四個指令 `POP r7`; `POP r6`; `POP r3`; `POP r2`，以便從堆疊中恢復這些暫存器的原始值。這四個額外的 `PUSH`/`POP` 對將操作計數和代碼大小（以位元組為單位）增加 $4 \times 2 = 8$。可以對其他暫存器機器進行類似的分析，從而得到表 **2**。

我們看到，在這些假設下，堆疊機器在代碼密度方面是明顯的贏家，並且在可擴展性方面處於獲勝組。

C.2.4. 情況 $m = 8$，$n = 16$：必須保留暫存器 `r8`...`r15`. 此情況的分析類似於前一種情況。結果總結在表 **3** 中。

請注意，除了「操作碼空間」列和單位址機器的行之外，生成的表與表 **1** 非常相似。因此， **C.2** 的結論在此情況下仍然適用，但有一些小的修改。然而，我們必須強調，這些結論僅對葉函數有效，即不呼叫其他函數的函數。除了最簡單的程式之外，任何程式都將有許多非葉函數，特別是如果我們正在最小化生成的機器代碼大小（這在大多數情況下阻止了函數的內聯）。

C.2.5. 使用二進制代碼而不是位元組代碼進行更公平的比較. 讀者可能已經注意到，我們之前對 k-位址暫存器機器和堆疊機器的討論在很大程度上取決於我們堅持完整指令必須由整數個位元組編碼。如果我們被允許使用「位元」或「二進制代碼」而不是位元組代碼來編碼指令，我們可以更均勻

機器	r	操作 資料	操作 算術	操作 總計	代碼位元組 資料	代碼位元組 算術	代碼位元組 **總計**	操作碼空間 資料	操作碼空間 **算術**	操作碼空間 總計
3-位址（樂觀）	*0*	0	11	12	0	22	**23**	32/256	**64/256**	97/256
3-位址（實際）	*0*	0	11	12	0	30	**31**	32/256	**34/256**	67/256
2-位址	*0*	4	11	16	8	22	**31**	33/256	**4/256**	38/256
1-位址	*1*	13	11	25	19	11	**31**	49/256	**64/256**	114/256
堆疊（基本）	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
堆疊（複合）	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 3: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本葉函數生成（參見 **C.1.1**），假設被呼叫的函數僅必須保留 16 個暫存器中的最後 8 個（$m = 8$，$n = 16$）。此表類似於表 **2**，但 r 的值較小。

地平衡不同機器使用的操作碼空間。例如，三位址機器的 `SUB` 的操作碼必須是 4 位元（對代碼密度好，對操作碼空間不好）或 12 位元（對代碼密度非常不好），因為完整指令必須佔據 8 位元的倍數（例如 16 或 24 位元），並且其中 $3 \cdot 4 = 12$ 個位元必須用於三個暫存器名稱。

因此，讓我們擺脫這個限制。

現在我們可以使用任意數量的位元來編碼指令，我們可以為所有考慮的機器選擇相同長度的所有操作碼。例如，所有算術指令都可以有 8 位元操作碼，就像堆疊機器一樣，每個使用 1/256 的操作碼空間；然後三位址暫存器機器將使用 20 位元來編碼每個完整的算術指令。暫存器機器上的所有 `MOV`、`XCHG`、`PUSH` 和 `POP` 都可以假設具有 4 位元操作碼，因為這是我們對堆疊機器上最常見的堆疊操作基本操作所做的。這些變化的結果顯示在表 **4** 中。

我們可以看到，各種機器的性能更加平衡，堆疊機器仍然是代碼密度方面的贏家，但三位址機器享有它真正值得的第二名。如果我們要考慮解碼速度和平行執行指令的可能性，我們將不得不選擇三位址機器，因為它僅使用 12 個指令而不是 21 個。

C.3 樣本非葉函數

本節比較了樣本非葉函數的不同暫存器機器的機器代碼。再次，我們假設被呼叫的函數保留 $m = 0$、$m = 8$ 或 $m = 16$ 個暫存器，其中 $m = 8$ 代表大多數現代編譯器和作業系統所做的折衷。

C.3.1. 非葉函數的樣本原始碼. 樣本原始檔案可以通過在我們解決兩個線性方程組的函數中（參見 **C.1.1**）用自訂的 *Rational* 類型（由指向記憶體中物件的指標表示）替換內建整數類型來獲得：

機器	r	操作 資料	算術	總計	代碼位元組 資料	算術	**總計**	操作碼空間 資料	**算術**	總計
3-位址	*0*	0	11	12	0	27.5	**28.5**	64/256	**4/256**	69/256
2-位址	*0*	4	11	16	6	22	**29**	64/256	**4/256**	69/256
1-位址	*1*	13	11	25	16	16.5	**32.5**	64/256	**4/256**	69/256
堆疊（基本）	*0*	16	11	28	16	11	**28**	64/256	**4/256**	69/256
堆疊（複合）	*0*	9	11	21	15	11	**27**	84/256	**4/256**	89/256

Table 4: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本葉函數生成（參見 **C.1.1**），假設僅必須由函數保留 16 個暫存器中的 8 個（$m = 8$，$n = 16$）。這次我們可以使用位元組的分數來編碼指令，以便匹配不同機器使用的操作碼空間。所有算術指令都有 8 位元操作碼，所有資料/堆疊操作指令都有 4 位元操作碼。在其他方面，此表類似於表 **3**。

```
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
}
```

我們將忽略與在記憶體中（例如在堆積中）分配 *Rational* 類型的新物件以及防止記憶體洩漏相關的所有問題。我們可以假設被呼叫的子程式 `r_sub`、`r_mul` 等通過在預分配的緩衝區中推進某個指標來簡單地分配新物件，並且未使用的物件稍後由垃圾回收器釋放，該垃圾回收器在被分析的代碼之外。

現在，有理數將由指標、地址或引用表示，這些將適合我們假設的暫存器機器的暫存器或我們堆疊機器的堆疊。如果我們想使用 TVM 作為這些堆疊機器的實例，我們應該使用 *Cell* 類型的值來表示對記憶體中 *Rational* 類型物件的此類引用。

我們假設子程式（或函數）由特殊的 `CALL` 指令呼叫，該指令由三個位元組編碼，包括要呼叫的函數的規範（例如「全域函數表」中的索引）。

C.3.2. 三位址和雙位址暫存器機器，$m = 0$ 保留的暫存器. 因為我們的樣本函數根本不使用內建算術指令，所以我們假設的三位址和雙位址暫存器機器的編譯器將產生相同的機器代碼。除了之前引入的 PUSH r(i) 和 POP r(i) 單位元組指令之外，我們假設我們的雙位址和三位址機器支援以下雙位元組指令：MOV r(i),s(j)、MOV s(j),r(i) 和 XCHG r(i),s(j)，對於 $0 \le i, j \le 15$。這些指令僅佔據 3/256 的操作碼空間，因此它們的添加似乎非常自然。

我們首先假設 $m = 0$（即所有子程式都可以自由破壞所有暫存器的值）。在這種情況下，我們的 r_f 機器代碼不必保留任何暫存器，但必須在呼叫任何子程式之前將包含有用值的所有暫存器保存到堆疊中。大小優化編譯器可能會產生以下代碼：

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

```
CALL r_mul  // ec
XCHG r0,s1  // a ; .. Dx D ec f
POP r1      // f ; .. Dx D ec
CALL r_mul  // af
POP r1      // ec; .. Dx D
CALL r_sub  // Dy:=af-ec
XCHG r0,s1  // Dx; .. Dy D
MOV r1,s0   // D
CALL r_div  // x:=Dx/D
XCHG r0,s1  // Dy; .. x D
POP r1      // D ; .. x
CALL r_div  // y:=Dy/D
MOV r1,r0   // y
POP r0      // x ; ..
RET
```

我們使用了 41 個指令：17 個單位元組（八個 PUSH/POP 對和一個 RET）、13 個雙位元組（MOV 和 XCHG；其中 11 個「新」的，涉及堆疊）和 11 個三位元組（CALL），總共 $17 \cdot 1 + 13 \cdot 2 + 11 \cdot 3 = 76$ 個位元組。[32]

C.3.3. 三位址和雙位址暫存器機器，$m = 8$ 保留的暫存器. 現在我們有八個暫存器，r8 到 r15，由子程式呼叫保留。我們可以將一些中間值保存在那裡，而不是將它們推入堆疊。然而，這樣做的代價是我們選擇使用的每個這樣的暫存器都有一個 PUSH/POP 對，因為我們的函數還需要保留其原始值。似乎在這種代價下使用這些暫存器並不能改善代碼的密度，因此對於 $m = 8$ 保留的暫存器，三位址和雙位址機器的最佳代碼與 **C.3.2** 中提供的代碼相同，總共 42 個指令和 74 個代碼位元組。

C.3.4. 三位址和雙位址暫存器機器，$m = 16$ 保留的暫存器. 這次所有 暫存器都必須由子程式保留，不包括用於傳回結果的暫存器。這意味著我們的代碼必須保留 r2 到 r5 的原始值，以及它用於臨時值的任何其他暫存器。編寫我們子程式代碼的一種直接方法是將暫存器 r2 到例如 r8 推入堆疊，然後執行所需的所有操作，使用 r6–r8 用於中間值，最後從堆疊中恢復暫存器。然而，這不會優化代碼大小。我們選擇另一種方法：

```
PUSH r0     // STACK: a
PUSH r1     // STACK: a b
```

[32]優化編譯器在啟用大小優化的情況下為 x86-64 架構為此函數產生的代碼實際上佔用了 150 個位元組，主要是因為實際指令編碼大約是我們樂觀假設的兩倍長。

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

我們使用了 39 個指令：11 個單位元組、17 個雙位元組（其中 5 個「新」指令）和 11 個三位元組，總共 $11 \cdot 1 + 17 \cdot 2 + 11 \cdot 3 = 78$ 個位元組。有點矛盾的是，代碼大小（以位元組為單位）比之前的情況稍長（參見 **C.3.2**），與人們可能期望的相反。這部分是由於我們假設涉及堆疊的「新」MOV 和 XCHG 指令是雙位元組編碼，類似於「舊」指令。大多數現有架構（例如 x86-64）對其對應於我們「新」移動和交換指令的指令使用較長的編碼（甚至可能是兩倍長），與「通常」的暫存器-暫存器指令相比。考慮到這一點，我們看到，假設新操作的三位元組編碼，我們在這裡會獲得 83 個位元組（而 **C.3.2** 中的代碼為 87 個），假設四位元組編碼則為 88 個位元組（而為 98 個）。這表明，對於沒有暫存器-堆疊移動和交換操作的優化編碼的雙位址架構，$m = 16$ 保留的暫存器可能會導致某些非葉函數的代碼稍短，但代價是葉函數（參見 **C.2.3** 和 **C.2.4**）將變得相當長。

C.3.5. 單位址暫存器機器，$m = 0$ 保留的暫存器. 對於我們的單位址暫存器機器，我們假設新的暫存器-堆疊指令僅通過累加器工作。因此，我們有三個新指令，LD s(j)（等同於雙位址機器的 MOV r0,s(j)）、ST s(j)（等同於 MOV s(j),r0）和 XCHG s(j)（等同於 XCHG r0,s(j)）。為了使與雙位址機器的比較更有趣，我們假設這些新指令的單位元組編碼，儘管這將消耗 $48/256 = 3/16$ 的操作碼空間。

通過將 **C.3.2** 中提供的代碼改編為單位址機器，我們獲得以下內容：

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

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

我們使用了 45 個指令：34 個單位元組和 11 個三位元組，總共 67 個位元組。與 **C.3.2** 中雙位址和三位址機器使用的 76 個位元組相比，我們再次看到，單位址暫存器機器代碼可能比雙暫存器機器的代碼更密集，代價是使用更多的操作碼空間（正如 **C.2** 中所示）。然而，這次額外的 3/16 操作碼空間用於資料操作指令，這些指令不依賴於特定的算術操作或呼叫的使用者函數。

C.3.6. 單位址暫存器機器，$m = 8$ 保留的暫存器. 如 **C.3.3** 中所述，在子程式呼叫之間保留 `r8`–`r15` 並不能改善我們先前編寫的代碼的大小，因此單位址機器對於 $m = 8$ 將使用 **C.3.5** 中提供的相同代碼。

C.3.7. 單位址暫存器機器，$m = 16$ 保留的暫存器. 我們簡單地將 **C.3.4** 中提供的代碼改編為單位址暫存器機器：

```
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
```

我們使用了 40 個指令：18 個單位元組、11 個雙位元組和 11 個三位元組，總共 $18 \cdot 1 + 11 \cdot 2 + 11 \cdot 3 = 73$ 個位元組。

C.3.8. 具有基本堆疊基本操作的堆疊機器. 我們重用 **C.1.5** 中提供的代碼，只需將算術基本操作（VM 指令）替換為子程式呼叫。唯一的實質性修改是在第三次乘法之前插入先前可選的 `XCHG s1`，因為即使是優化編譯器現在也無法知道 `CALL r_mul` 是否是可交換操作。我們還使用了「尾遞迴優化」，將最後的 `CALL r_div` 後接 `RET` 替換為 `JMP r_div`。

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

```
XCHG s1     // D Dy Dx
PUSH s2     // D Dy Dx D
CALL r_div  // D Dy x
XCHG s2     // x Dy D
JMP r_div   // x y
```

我們使用了 29 個指令；假設所有堆疊操作都是單位元組編碼，CALL 和 JMP 指令是三位元組編碼，我們最終得到 51 個位元組。

C.3.9. 具有複合堆疊基本操作的堆疊機器. 我們再次重用 **C.1.7** 中提供的代碼，將算術基本操作替換為子程式呼叫並進行尾遞迴優化：

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

此代碼僅使用 20 個指令，9 個與堆疊相關，11 個與控制流相關（CALL 和 JMP），總共 48 個位元組。

機器	m	操作 資料	控制	總計	代碼位元組 資料	控制	**總計**	操作碼空間 資料	**算術**	總計
3-位址	*0,8*	29	12	41	42	34	**76**	35/256	**34/256**	72/256
	16	27	12	39	44	34	**78**			
2-位址	*0,8*	29	12	41	42	34	**76**	37/256	**4/256**	44/256
	16	27	12	39	44	34	**78**			
1-位址	*0,8*	33	12	45	33	34	**67**	97/256	**64/256**	164/256
	16	28	12	40	39	34	**73**			
堆疊（基本）	–	18	11	29	18	33	**51**	64/256	**4/256**	71/256
堆疊（複合）	–	9	11	20	15	33	**48**	84/256	**4/256**	91/256

Table 5: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本非葉函數生成（參見 **C.3.1**），假設被呼叫的子程式必須保留 16 個暫存器中的 m 個。

C.4 樣本非葉函數的機器代碼比較

表 **5** 總結了對應於 **C.3.1** 中提供的相同原始檔案的機器代碼的屬性。我們僅考慮「實際」編碼的三位址機器。三位址和雙位址機器具有相同的代碼密度屬性，但在操作碼空間的利用方面有所不同。單位址機器，令人驚訝的是，設法產生了比雙位址和三位址機器更短的代碼，代價是使用了超過一半的所有操作碼空間。堆疊機器是此代碼密度競賽中的明顯贏家，而不會損害其出色的可擴展性（以用於算術和其他資料轉換指令的操作碼空間來衡量）。

C.4.1. 與葉函數結果的結合. 將此表與 **C.2** 中樣本葉函數的結果進行比較是有教益的，該結果總結在表 **1**（對於 $m = 0$ 保留的暫存器）和非常相似的表 **3**（對於 $m = 8$ 保留的暫存器）中，並且，如果仍然對 $m = 16$ 的情況感興趣（結果證明在幾乎所有情況下都比 $m = 8$ 差），還可以參考表 **2**。

我們看到，堆疊機器在非葉函數上擊敗了所有暫存器機器。至於葉函數，只有具有算術指令「樂觀」編碼的三位址機器能夠擊敗堆疊機器，以 15% 的優勢獲勝，但損害了其可擴展性。然而，同一台三位址機器為非葉函數產生的代碼長 25%。如果典型程式由大約相等比例的葉函數和非葉函數組成，那麼堆疊機器仍然會獲勝。

C.4.2. 使用二進制代碼而不是位元組代碼進行更公平的比較. 與 **C.2.5** 類似，我們可以通過使用任意二進制代碼而不是位元組代碼來編碼指令，並匹配不同機器的資料操作和算術指令使用的操作碼空間，來對不同的暫存器機器和堆疊機器進行更公平的比較。此修改比較的結果總結在表 **6** 中。我們看到堆疊機器仍然以很大的優勢獲勝，同時使用更少的操作碼空間用於堆疊/資料操作。

機器	m	操作 資料	控制	總計	代碼位元組 資料	控制	**總計**	操作碼空間 資料	**算術**	總計
3-位址	*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-位址	*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-位址	*0,8*	33	12	45	33	34	**67**	112/256	**4/256**	119/256
	16	28	12	40	33.5	34	**67.5**			
堆疊（基本）	–	18	11	29	18	33	**51**	64/256	**4/256**	71/256
堆疊（複合）	–	9	11	20	15	33	**48**	84/256	**4/256**	91/256

Table 6: 假設 3-位址、2-位址、1-位址和堆疊機器的機器代碼屬性摘要，為樣本非葉函數生成（參見 **C.3.1**），假設被呼叫的子程式必須保留 16 個暫存器中的 m 個。這次我們使用位元組的分數來編碼指令，從而實現更公平的比較。在其他方面，此表類似於表 **5**。

C.4.3. 與實際機器的比較. 請注意，我們的假設暫存器機器已經過相當優化，以產生比實際存在的暫存器機器更短的代碼；後者除了代碼密度和可擴展性之外，還受到其他設計考量的約束，例如向後相容性、更快的指令解碼、相鄰指令的平行執行、編譯器自動產生優化代碼的容易程度等。

例如，非常流行的雙位址暫存器架構 x86-64 產生的代碼大約是我們對雙位址機器「理想」結果的兩倍長。另一方面，我們對堆疊機器的結果直接適用於 TVM，TVM 是在考慮本附錄中提出的考量的情況下明確設計的。此外，實際的 TVM 代碼甚至比表 **5** 中顯示的更短（以位元組為單位），因為存在雙位元組 `CALL` 指令，允許 TVM 從 `c3` 的字典呼叫多達 256 個使用者定義的函數。這意味著，如果要專門考慮 TVM，而不是抽象堆疊機器，應該從表 **5** 中堆疊機器的結果中減去 10 個位元組；這產生了大約 40 個位元組（或更短）的代碼大小，幾乎是抽象雙位址或三位址機器的一半。

C.4.4. 自動生成優化代碼. 一個有趣的點是，我們樣本中的堆疊機器代碼可能已經由一個非常簡單的優化編譯器自動生成，該編譯器在呼叫每個基本操作或呼叫函數之前適當地重新排列堆疊頂部附近的值，如 **2.2.2** 和 **2.2.5** 中所述。唯一的例外是 **C.1.7** 中描述的不重要的「手動」`XCHG3` 優化，它使我們能夠將代碼再縮短一個位元組。

相比之下， **C.3.2** 和 **C.3.3** 中顯示的高度優化（關於大小）的暫存器機器代碼不太可能由優化編譯器自動產生。因此，如果我們比較編譯器生成的代碼而不是手動生成的代碼，堆疊機器在代碼密度方面的優勢將更加顯著。