200915175

發明專利說明書

（本說明書格式、順序及粗體字，請勿任意更動，※記號部分請勿填寫）

※ 申請案號：97131551

※ 申請日期：97.8.19　　※IPC 分類：G06F 7/58

一、發明名稱：（中文/英文）

基於移位加法運算之亂數產生器/

SHIFT-ADD BASED RANDOM NUMBER GENERATION

二、申請人：（共1人）

姓名或名稱：（中文/英文）

VNS 管理股份有限公司/VNS PORTFOLIO LLC

代表人：（中文/英文）

F. 艾力克 桑德斯/F. ERIC SAUNDERS

住居所或營業所地址：（中文/英文）

美國加州 95014 庫普堤諾市 20400 史帝芬溪大道第 500 室

國　籍：（中文/英文）

美國/US

三、發明人：（共1人）

姓　名：（中文/英文）

麥克 B. 蒙特夫立斯基/MICHAEL B. MONTVELISHSKY

國　籍：（中文/英文）

美國/US

四、聲明事項：

☐ 主張專利法第二十二條第二項☐第一款或☐第二款規定之事實，其事實發生日期為：　年　月　日。

☑ 申請前已向下列國家（地區）申請專利：

【格式請依：受理國家（地區）、申請日、申請案號　順序註記】

☑ 有主張專利法第二十七條第一項國際優先權：

1.美國、2007/9/24、60/974,820

2.美國、2008/4/18、12/148,511

☐ 無主張專利法第二十七條第一項國際優先權：

☐ 主張專利法第二十九條第一項國內優先權：

【格式請依：申請日、申請案號　順序註記】

☐ 主張專利法第三十條生物材料：

☐ 須寄存生物材料者：

國內生物材料 【格式請依：寄存機構、日期、號碼　順序註記】

國外生物材料 【格式請依：寄存國家、機構、日期、號碼　順序註記】

☐ 不須寄存生物材料者：

所屬技術領域中具有通常知識者易於獲得時，不須寄存。

九、發明說明：

【發明所屬之技術領域】

本發明有關於電子計算機以及具有處理架構並且執行指令處理之數位處理系統，更有關於可實現亂數產生的處理程序。

【先前技術】

強大且有效率的操作碼(operational codes, op-codes)對現今執行多任務的電腦處理器來說是非常重要的。例如，某些任務為乘法運算並且產生一連串的虛擬隨機數。

乘法算運是大家皆瞭解的工作，其重要性不需要加以贅述。相反的，此處所討論亂數與虛擬亂數產生器的優點是用來建立本發明的背景。

不論是否可選擇一組真的亂數，用於亂數的處理程序以及不論隨機性的現狀是否可受到證實皆是辯論的主題。由於電腦僅執行預先定義的指令來產生完全可預測的結果，因此足以接受電腦無法產生亂數的事實。然而，有許多電腦演算法可用來產生虛擬亂數，且使用具有這些演算法的電腦對許多重要的應用程式都有幫助。除非有人知道第一虛擬亂數出現於序列的何處，否則結果是完全不可預測的，因此基於此理由可交替使用亂數或虛擬亂數這兩個用法。

電腦產生的虛擬亂數具有兩種主要偏好。第一，虛擬

亂數序列正是重複相同實驗之數個疊代的結果。對某些科學實驗來說這項偏好是很重要的。此處的”種子”值係作為對電腦演算法的初始輸入，且接下來預期該電腦演算法在每個疊代中產生相同數字序列。

第二，虛擬亂數序列不僅看似隨機並且是完全不可預測的。對密碼以及許多其他應用來說這項偏好是很重要的。此處的”種子”值將會對基於相同種子之不同疊代產生完全不同的亂數序列。

由於在此沒有詳細探討亂數的特定應用，且事實上可以產生足以用於這些應用程式的虛擬亂數序列，因此在這種情況下需要對這些方法進行比較。例如，在許多狀況中碰到的問題並不在於產生虛擬亂數序列，而是在於證明這樣的序列對既定應用程式來說足夠隨機。事實上，對某些應用程式來說並不需要真正的隨機性而是需要明顯的隨機性。接下來的段落將探討既定序列隨機性的測試方法，並且略述關於”品質”虛擬隨機序列的特徵。

對某些應用程式來說，由於應用程式可能偏好明顯亂數與真正亂數的其中一種，因此區分明顯亂數與真正亂數是很重要的。即使實際上無法證明數字串列是隨機的，仍有許多獨特且有趣的方法來判定串列的隨機性。

假設一人觀看輪盤賭(roulette wheel)連續旋轉 22 次。觀察輪盤賭前 18 次旋轉的結果皆為隨機輸出，然而其第 19 至 22 次的旋轉出現相同之值。基於此觀點會出現兩種假設。第一種假設為，由於最近的四次旋轉皆出現相

同之值，因此下一次旋轉有很大的機率會再出現相同之值。第二種假設為，下一次的旋轉不會出現與前四次旋轉相同之值。上面兩種假設皆是錯誤的，但其導致具有明顯隨機與真正隨機的想法。和其他先前或未來的旋轉一樣，輪盤賭每一次新的旋轉皆具有相同的期望結果。這不代表每次的旋轉都是隨機的。應該說輪盤賭所有結果的基本分佈應該呈現均勻分佈，這是真正亂數序列的一種行為。四次連續旋轉產生相同之值不是明顯隨機行為的特性，但是對真正亂數序列來說包含四個連續相同之值並不是不合理的。

在另一個例子中，假設 iPod™ 使用者想要聽取音樂，但也想要聽取每一首新的音樂。若 iPod 包含真正隨機方法來選取下一首播放的歌曲，則可能會發生對相同歌曲連續播放數次的情況。然而這並不是裝置製造商最感興趣的，由於大部分的使用者在重複播放歌曲之前會想要聽取所有的歌曲，因此蘋果公司的 iPod 產品中包含真正亂數產生器。由於使用真正隨機方法來產生下一首播放歌曲將會與消費者的偏好產生衝突，因此製造商不大可能使用真正隨機方法來產生下一首播放歌曲。

上述兩例子展示了應用程式可預定義對明顯隨機與真正隨機數的需求。

現今沒有任何測試方法可以用來判斷數字序列是否為真正隨機。因此發展出許多測試方法。國家標準和科技研究機構 (National Institute of Standards and

Technology, NIST)使用其中的16種測試方法，其中結合這些測試的結果對問題中序列的隨機性程度提供更好的說明。NIST出版對這些測試方法的詳細說明，並且對該方法與這些說明中的實際與查覺的錯誤有激烈的爭論。值得注意的是，當每個測試提供了關於序列是否為隨機之問題的是或否的答案時，測試本身並不能保證隨機性。即使16個測試方法中的每個方法對隨機性問題之是或否的答案可能不盡相同，測試結果取決於待測序列的信心水準(confidence level)。因此，判斷既定序列測試的隨機性等級以及在宣告關於既定序列隨機性之前估計所有測試的結果是很重要的。

上述討論中用來測試”夠好的”隨機性方法中並沒有提到對隨機產生數字序列之”品質”的定義。即使序列通過NIST測試，該序列也可能不比沒通過或僅通過較低信心水準的其他序列更佳。

速度也是對許多應用程式來說一個相當重要的考量。因此，所產生的每個隨機序列皆應該可回答下列兩個問題。第一個問題為，多快產生序列中的第一項目？第二個問題為，多快產生該序列中連續的項目？這些問題的答案並不佔據任何NIST測試。然其卻是品質序列的重要特徵。另一重要的品質考量為序列中的數字不能太隨機(上述iPod的例子展示了這點)。太隨機的序列會降低序列的品質。品質隨機數序列將依附至雜訊範圍(noise sphere)的特徵(可以參考關於此專題的學術文件)。因此，必須瞭

解若序列數值產生的太慢、太過隨機或是沒有依附至某些雜訊標準，則即使該序列通過所有 NIST 測試仍可能沒有用的這一點是很重要的。

【發明內容】

因此，本發明 目的為提供一種基於移位加法運算之亂數產生器，有助於處理器中的各種操作。

本發明實施例提供一種虛擬亂數產生系統，包括處理器以及一邏輯。該處理器具有包含第一值之第一記憶體以及包含第二值之第二記憶體。該邏輯於循環條件成立時執行+*操作。

【實施方式】

為讓本發明之上述和其他目的、特徵、和優點能更明顯易懂，下文特舉出較佳實施例，並配合所附圖式，作說明如下：

實施例：

第 9 圖顯示本發明較佳實施例之基於移位加法運算之亂數產生程序 300。

本發明基於移位加法運算之亂數產生程序 300(第 9 圖)為本發明人公司所發明移位加法運算機制的應用。基於此理由，以下會先對移位加法運算機制作說明。

SEAFORTH™ 24A 裝置上的+*操作碼

移位加法運算系統 100(第 6 圖)可用於各種任務，包括乘法運算以及產生虛擬隨機數，然其並非用以限定本發明的範圍。在 Venture Forth™ 程式語言中的移位加法運算系統 100 具有”+*”操作碼。在呈現更多詳細實施例之前，考慮由加州 Cupertino 市的 IntellaSys™ 公司(TPL 集團的一員)製造之 SEAforth™ 24a 裝置的上下文中的簡單範例是有幫助的。

傳統 SEAforth™ 24a 具有以 24 堆疊為基礎之為處理器核心，其中皆使用 Venture Forth™ 程式語言。第 1 圖(先前技術)顯示為此語言以十六進位、助憶(mnemonic)以及二進位表示法表示之 32 種操作碼(op-code)的圖表。這些操作碼被分為兩種主要類型，分別為記憶指令(memory instruction)以及算術邏輯單元(arithmetic logic unit, ALU)指令，每部分皆具有 16 個操作碼。第 1 圖的左半部為記憶指令，而第 1 圖的右半部為 ALU 指令。清楚區別操作碼之間的方法為記憶指令最左邊的位元為”0”，而 ALU 指令最左邊的位元為”1”。另外，不論十六進位或二進位表示法的操作碼皆具有上述特性。本發明感興趣的”+*”操作碼顯示於右半部的最上面。

第 2 圖(先前技術)顯示 SEAforth™ 24a 裝置中每個核心的一般架構的示意圖。SEAforth™ 24a 中除了與此處不相關之 B 暫存器與 PC 暫存器之外的所有暫存器皆為 18 位元寬。程式設計者可透過不同的兩種方法來選擇構成

SEAforth™ 24a 中 18 位元寬暫存器空間的位元(對某些使用 A 暫存器的操作碼具有有限的例外)。第一種方法為將此空間分為四個相同的擴充槽(slot),稱為擴充槽 0、擴充槽 1、擴充槽 2 以及擴充槽 3。由於 18 除以 4 會產生餘數,因此這些擴充槽的位元長度不盡相同。前三個擴充槽(擴充槽 0、擴充槽 1 以及擴充槽 2)各具有 5 位元,而擴充槽 3 僅具有 3 位元。

第 3a-b 圖(先前技術)顯示 SEAforth™ 24a 裝置中 18 位元寬的暫存器的方塊圖,其中第 3a 圖顯示位元 0 至 17 的實際配置,而第 3b 圖顯示位元-2 至 17 的概念配置。在第 3a 圖中,位元 13-17 形成擴充槽 0,位元 8-12 形成擴充槽 1,位元 3-7 形成擴充槽 2,位元 0-2 形成擴充槽 3。SEAforth™ 24a 裝置的設計者通常會指出 18 位元寬的暫存器各包括 3 以及 3/5 個指令,這點出一個問題也就是擴充槽 3 是否有意義,因為沒有任何一個第 1 圖中的操作碼符合擴充槽 3。第 3b 圖顯示 SEAforth™ 24a 裝置的設計者如何處理此問題。設計者僅藉由將兩個最低有效位元(位元-1 以及位元-2)硬佈線至接地點或零而使某些操作碼符合擴充槽 3。當然,由於擴充槽 3 僅具有 3 位元的空間(而不是 5 位元的空間),因此符合擴充槽 3 的幾個操作碼被限制為 32 個可能操作碼中的 8 個。值得注意的是,這些操作碼為:

```
$00        00000b     ;(return)
$04        00100b       Unext
```

$08	01000b	@p+
$0C	01100b	!p+
$10	10000b	+*
$14	10100b	+
$18	11000b	Dup
$1C	11100b	·(nop)

程式設計者選擇構成 SEAforth™ 24a 中 18 位元寬暫存器空間的位元的第二種方法為不要將 18 位元寬的暫存器分為擴充槽，而是將暫存器視為包括單一 18 位元的二進位值。這種方法與以擴充槽為基礎的第一種方法截然不同，但實際上這兩種表示法是相同的。第 4a-b 圖(先前技術)顯示此範例的方塊圖。第 4a 圖顯示填滿四個·(nop)操作碼的擴充槽，而第 4b 圖顯示填滿數字 236775(無號二進位(unsigned binary))的暫存器。參照第 1 圖可以理解第 4a-b 圖中的二進位位元值非常相似。這代表程式設計者必須判斷暫存器包含數字或是包含四個操作碼。

第 5a-b 圖(先前技術)分別顯示存在於 SEAforth™ 24a 裝置的每個核心中之返回與資料堆疊元件的示意圖。第 5a 圖顯示返回堆疊區域如何包含叫做”R”(R 暫存器)的上層暫存器以及八個暫存器的環形緩衝器(circular buffer)。第 5b 圖顯示資料堆疊區域如何包括叫做”T”(T 暫存器)的上層暫存器、T 暫存器下方又叫做”S”(S 暫存器)的第二暫存器以及八個暫存器的環形緩衝器。返回堆疊因此包括九個暫存器，且資料堆疊包括十個暫存器。在

下列實施例中僅需要考慮資料堆疊區域。

TBL. 1-4 中顯示在一組假設的+*範例中的 T 暫存器與 S 暫存器之值。為了方便說明，所附圖示僅顯示 4 位元欄位寬度。值得注意的是，在執行+*操作碼期間若 S 暫存器中的值維持不變，則 T 暫存器中的值會改變。為了避免對構成數值之位元與具有該值之記憶體位置兩者之間的混淆，在此稱為數值中的位元以及記憶體中的位元位置。因此，一數值具有最高有效位元(MSB)以及最低有效位元(LSB)且記憶體中的位置具有高位元(high bit, HB)位置以及低位元(LB)位置。

TBL. 1 顯示初始設置於 T 暫存器之數值 1 以及設置於 S 暫存器之數值 3。由於此處 T 的低位元(LB)位置為 1，因此在執行+*操作碼期間：

(1)將 S 與 T 相加並將結果放到 T 中(參照 TBL. 2)；以及

(2)T 的內容向右移位並且將 0 放到位元 4(參照 TBL. 3)。

將 0 放到位元 4 的原因會於下文中說明。

現在將 TBL. 3 中 T 與 S 的內容用於第二實施例。由於 T 現在的 LB 位置為 0，因此在執行其他+*操作碼期間：

(1)T 的內容向右移位並且將 0 放置於位元 4(參照 TBL. 4)。

同樣的，將 0 放到位元 4 的原因會於下文中說明。另外，值得注意的是，將 T 中的所有位元向右移位與在執行

+*操作碼之前將T的LB位置填入1或0沒有任何關係。取而代之的是，將T中的所有位元向右移位與+*操作碼本身有關聯。

這兩個範例幾乎說明了與+*操作碼有關的所有動作。但在此處沒有說明為什麼將0填入位元4，下文中將會說明這麼做的原因。

+*操作碼的一般例子

一般對+*操作碼的說明為其執行有條件相加並且在移位之後當1或0填入T的高位元(HB)位置時將T中的所有位元向低階位元方向移位。

第6圖顯示本發明移位加法運算系統100的方塊圖，其中顯示與執行單一+*操作碼相關的所有可能動作。+*操作碼具有兩種主要子處理程序，包括移位子處理程序102以及有條件相加子處理程序104。移位相加運算系統100具體化為+*運算碼，開始於步驟106，步驟108係判斷T的LB位置內容。首先討論移位子處理程序102(當T的LSB為0時)，在步驟110中判斷T的HB位置內容。若T的HB位置為0，則在步驟112將T的內容向右移位，在步驟114中將0填入T的HB位置，且在步驟116中T包含其新的數值。另外，若T的HB位置為1，則在步驟118中將T的內容向右移位，在步驟120中將1填入T的HB位置，且在步驟116中T包含其新的數值。

現在來討論有條件相加子處理程序104(當T的LSB

為 1 時），在步驟 122 中將 T 與 S 的內容相加，且在步驟 124 中判斷是否產生進位。若沒有產生進位，則進入移位子處理程序 102 的步驟 110(如圖所示)。另外，若產生進位(進位位元為 1)，則進入移位子處理程序 102 的步驟 118(如圖所示)。接下來，+*操作碼的處理程序(移位加法運算系統 100)繼續執行移位子處理程序至步驟 116，T 將會包含新的數值。

由於定義與+*操作碼相關的動作很容易，因此第 6 圖顯示執行+*操作碼在概念上並不簡單。第 7 圖的圖表顯示在執行+*操作碼之前 T(在此稱為舊的 T 暫存器)的 LB 位置與 HB 位置之間的關係，當 T 與 S 之值相加時(若有發生此動作)的中間進位，以及在執行+*操作碼之後所產生 T(在此稱為新的 T 暫存器)的 HB 與倒數第二位元(HB-1)。

一種+*虛擬碼演算法

藉由使用虛擬碼演算法來說明+*操作碼的一般例子。假設+*操作碼於 n 位元機器上執行，其中 n_t 位元寬的數值初始設置於 T 而 n_s 位元寬的數值初始設置於 S。另外，即使+*操作碼產生理論上至少需要一位元來表示之進位，仍假設只有一額外位元能用來表示進位。對於 n_t 與 n_s 的長度並沒有限制，然其個別的位元長度應該小於或等於 n 的位元寬度。虛擬碼如下：

1. 若 T 的 LB 位置為 1：

1a. 將 T 中之值 t 與 S 中之值 s 相加，並且將其總

和 t+s(t')取代目前 T 中之值 t，且 S 維持不變。

1a1.若 T 的 HB 位置為 1：

1a1a.若 t 與 s 相加產生進位：

1a1a1.將 T 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的部分係填入數值 1，其中移位後的位元 m(m<n)包括移位前位元 m+1 的內容。當有效的破壞移位前的 t' 之位元 0 時，此處理步驟不具有位元 n。移位後的 t' 之位元 n(數值 1)將會以數值 1 填入。

1a1b.若 t 與 s 相加不產生進位：

1a1b1.將 T 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的部分係填入數值 1，其中移位後的位元 m(m<n)包括移位前位元 m+1 的內容。當有效的破壞移位前的 t' 之位元 0 時，此處理步驟不具有位元 n。移位後的 t' 之位元 n(數值 1)將會以數值 1 填入。

1a2.若 T 的 HB 位置為 0：

1a2a.若 t 與 s 相加產生進位：

1a2a1.將 T 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的部分係填入數值 1，其中移位後的位元 m(m<n)包括移

位前位元 m+1 的內容。當有效的破壞移位前的 t' 之位元 0 時，此處理步驟不具有位元 n。移位後的 t' 之位元 n(數值 1)將會以數值 1 填入。

1a2b.若 t 與 s 相加產生進位：

1a2b1.將 t 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的部分係填入數值 1，其中移位後的位元 m(m<n)包括移位前位元 m+1 的內容。當有效的破壞移位前的 t' 之位元 0 時，此處理步驟不具有位元 n。移位後的 t' 之位元 n(數值 1)將會以數值 0 填入。

2.若 T 的 LB 位置為 0：

2a.若 T 的 HB 位置為 1：

2a1.將 T 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的部分係填入數值 1，其中移位後的位元 m(m<n)包括移位前位元 m+1 的內容。當有效的破壞移位前的 t' 之位元 0 時，此處理步驟不具有位元 n。移位後的 t' 之位元 n(數值 1)將會以數值 1 填入。

2b.若 T 的 HB 位置為 0：

2b1.將 T 中的所有位元向右移位 1 位元。移位後的 t' 之位元 0 包括移位前位元 1 的內容。移位後的 t' 之位元 1 包括移位前位元 2 的內容。同樣的，t' 剩下的

部分係填入數值1，其中移位後的位元m(m<n)包括移位前位元m+1的內容。當有效的破壞移位前的t'之位元0時，此處理步驟不具有位元n。移位後的t'之位元n(數值1)將會以數值0填入。

值得注意的是，先前的+*操作碼總是包括在T中所有位元之向右移位的一位元(向低階位元方向)。位元移位並不是執行+*操作碼之前、期間或是之後之任何事件的結果。位元移位永遠是與+*操作碼有關的執行事件。

使用+*操作碼執行乘法運算

此處意味著本發明移位加法運算系統100可用來執行乘法運算。本發明實施例說明使用+*操作碼來執行完整且正確的乘法運算之一般例子。

假設將數值9與數值7相乘，T代表數值9初始設置的8位元記憶體位置，S代表數值7初始設置的8位元記憶體位置。為了方便說明，儘管下述的概念可延伸用於任何位元寬度，在此不使用18位元暫存器寬度的SEAforth™ 24a裝置。

TBL. 5-10顯示在一組假設的+*乘法運算範例中T暫存器與S暫存器之值。如TBL. 5所示，數值9初始設置於T暫存器，而數值7初始設置於S暫存器。接下來，將T中的值向右對齊至8位元欄位寬度，使得四個前導位元(leading bit)由0填滿。相反的，S中的值向左對齊至8位元欄位寬度，使得四個後置位元(trailing bit)由0填

滿。TBL. 6顯示執行這些對齊後的結果。

在此，正確的乘法運算需要連續執行四個+*操作碼。第一次+*操作具有下列影響。T的LB位置為1(參照TBL. 6)，使得T與S中的值相加，並將結果設置於T中(參照TBL. 7的左半部)。接下來，透過上述1a2b1的方式將T中的值向右移位1位元。經過第一次+*操作後的值顯示於TBL. 7的右半部。

由於T的LB位置為0，因此第二次+*操作相當簡單。T中所有的位元皆透過上述2b1的方式向右移位。經過第二次+*操作後的值顯示於TBL. 8。

由於T的LB位置也是0，因此第三次+*操作與第二次+*操作相似。T中所有的位元皆再次透過上述2b1的方式向右移位。經過第三次+*操作後的值顯示於TBL. 9。

第四次與最後一次+*操作與第一次+*操作相似。T的LB位置為1(參照TBL. 9)，使得T與S中的值相加，並將結果設置於T中(參照TBL. 10的左半部)。接下來，透過上述1a2b1的方式將T中的值向右移位1位元。經過第四次+*操作後的值顯示於TBL. 10的右半部。

TBL. 10中顯示T的結果為十進位數值63，也就是將數值9與數值7執行乘法運算的期望結果。

用於乘法運算的+*虛擬碼演算法

在執行此虛擬碼之前，若T與S中的有效位元總和小於或等於16位元，則具有正值之正值乘法運算會產生正

確的乘積。而在執行此虛擬碼之前，若 T 與 S 中的有效位元總和小於或等於 17 位元，則具有負值之正值乘法運算會產生正確的乘積。值得注意的是，在執行此虛擬碼之前，S 應該包括 S 中期望負值的二補數(two's complement)

1.若期望之乘法運算為具有正值之正值：

1a.對 T 之 n 位元欄位寬度中的 t 進行向右對齊。

1a1.將 T 中 t 的最高有效位元之後的所有前導位元填入 0。填入 0 的前導位元數量為 $n-n_t$。

1b.調整 S 之 n 位元欄位寬度中的 s，使得 s 的最低有效位元的設置比 T 中 t 的最高有效位元高 1 位元。

1b1.以 0 填入 S 中所有的前導與後置位元。填入 0 的位元數量為 $n-n_s$。

1c.執行乘法運算。

1c1.完成索引從 1 至 n_t 的 for 迴圈。

1c1a.執行用於上述一般例子的+*虛擬碼。

2.若期望之乘法運算為具有負值之正值：

2a.對 T 之 n 位元欄位寬度中的 t 進行向右對齊。

2a1.將 T 中 t 的最高有效位元之後的所有前導位元填入 0。填入 0 的前導位元數量為 $n-n_t$。

2b.對 S 中的數值 s 執行二補數運算。

2b1.藉由有效位元 n_t 的數量對 S 中的數值 s 向 S 的 HB 進行位元移位

2c.執行乘法運算。

2c1.完成索引從 1 至 n_t 的 for 迴圈。

2c1a.執行用於上述一般例子的+*虛擬碼。

3.若期望之乘法運算為具有負值之負值：

3a.對 T 中的數值 t 執行二補數運算。

3b.對 S 中的數值 s 執行二補數運算。

3c.執行 1a-1c。

當然，只要負值在 T 中且正值在 S 中，具有正值之負值的乘法運算與上述 2 的乘法運算相同。

第 8 圖顯示本發明實施例以移位加法為基礎之乘法運算處理程序 200。步驟 202 開始以移位加法為基礎之乘法運算處理程序 200。在步驟 204 中，若 T 為第一記憶體位置，則將第一值設置於第一記憶體位置中，也就是上述 1 的向右對齊方法。在步驟 206 中，若 S 為第二記憶體位置，則將第二值設置於第二記憶體位置中，也就是上述 2 的向左對齊方法。熟悉此技藝者皆瞭解可使用與下列計數比較減少(count-compare-work-decrement)方法不同的其他程式控制機制。在步驟 208 中判斷+*操作碼的疊代(iteration)次數。此次數實質上必須與第一值(在 T 中)中的有效位元數量相同。在步驟 210 中判斷是否已執行所有必要的+*操作碼疊代。若否，則在步驟 212 中執行+*操作碼疊代，並在步驟 214 中減少仍須執行之+*操作碼疊代的計數值。另外，若步驟 210 判斷所有需要的+*操作碼疊代皆已執行，則在步驟 216 中，則目前的第一記憶體(在 T 中)中為第一與第二值的乘積。

使用+*操作碼產生亂數

移位加法機制 100 亦可用來產生虛擬亂數(pseudo random number)。接下來說明利用+*操作碼產生亂數的例子。

假設被問到下列問題：僅透過使用+*操作碼來執行完整且正確的乘法運算或是產生虛擬亂數比較有效率？假設透過+*操作碼來執行複雜的乘法運算比產生亂數更有效率看似合乎邏輯的假設。實際上，除了對僅具有幾位元長度的兩數執行乘法運算之外，否則上述假設不成立。另外，令許多資深程式設計師意外的是，對 SEAforth™ 24a 裝置來說產生虛擬亂數是實際上最短的程式。因此，透過執行乘法運算相同的+*操作碼來產生虛擬亂數更容易完成。

如同乘法運算，亂數產生器需要兩值，分別是 T 暫存器中 n_t 位元寬的數值 t 以及 S 暫存器中 n_s 位元寬的數值 s，其中 n_t 與 n_s 皆大於零。這代表 S 與 T 暫存器中之值具有至少一有效位元。接下來顯示虛擬亂數演算法，該演算法為虛擬亂數產生器的概要。

產生隨機亂數的虛擬碼+*演算法

以下不是錯誤。此處的虛擬碼+*演算法僅具有一行文字。

1. 當某些循環條件成立時：

1a. 執行+*虛擬碼演算法。

一種 FORTH 碼的+*亂數產生器

第 9 圖顯示亂數產生器的碼列表 300(行編號做為參考用，且此處的程式碼具有 ANS Forth 與 Venture Forth™ 的元件)。T 暫存器中的種子(seed)與 S 暫存器中之值可以在命名為”random.log” 的個別檔案中產生約 2^{18}(132,000)個虛擬亂數。接下來針對每個行編號作說明。

行說明

1.註解：檔案名稱。

2.載入編譯器/模擬器。

3.作為編碼風格的虛格(white space)。

4.註解：此區段為在主機上執行之程式碼的開始直到在目標機器上執行的行編號 32-37 為止。接下來的冒號定義是用來協助檔案處理。

5.對主機(非目標機器)進行編譯。

6.作為編碼風格的虛格。

7.產生如變數之值，並回傳取代其位址之一值來開啟檔案，fid 為檔案名稱的縮寫。

8.開始產生，命名主機 Forth 目錄內的位置。對主機 Forth 目錄內的位置命名並且傳送回車(carriage return)與換行(line feed)字元至檔案。

9.作為編碼風格的虛格。

10.以標準 Forth 方式開啟檔案。

11.產生檔案”random.log”，其中 r/w 代表可讀取/寫入，create-file 透過暫用覆寫任何現存檔案來產生檔

案，若成功則回傳錯誤碼 0，若不成功則回傳錯誤碼 1 並且將該錯誤碼丟棄。

12.作為編碼風格的虛格。

13.以標準 Forth 方式關閉檔案。

14.將已開啟的檔案關閉，傳送錯誤碼(0 或 1)，將 0 儲存至 fid，因此之後不會不小心使用錯誤碼 1。

15.作為編碼風格的虛格。

16.以標準 Forth 方式寫入檔案，將數值轉換為字串並寫入檔案。

17.寫入檔案”ramdom.log”的數值以十進位為基礎。

18.使用標準 Forth 圖片數值輸出運算子對該值即將寫入檔案的欄位寬度格式化，將該值寫入該檔案，並再次回傳錯誤碼(0 或 1)。

19.將字串置放於包含回車與換行字元的堆疊並寫入檔案，因此下一值會寫入新的一行並且擺脫錯誤碼。

20.作為編碼風格的虛格。

21.以標準 Forth 方式從目標機器的 T 暫存器中擷取數值，並使用先前的冒號定義將 T 暫存器中的數值寫入檔案 random.log 中。

22.表示以十六進位的方式處理數值。這實際上開啟檔案 random.log 並將 0 置放於主機堆疊中。

23.具有從 0、1 至 2 的計數值，包括 0、1 與 2。接下來執行兩個迴圈，分別是 do 迴圈以及 begin 迴圈。步驟

接著進行至一循環，其中目標機器資料堆疊中的T值具有主機上的變數名稱t。此程式碼將會循環至變數t之值不同於主機堆疊上之值為止。

24.交換主機堆疊上的兩個數值，將0置放於主機堆疊中並接著開始另一迴圈。

25.繼續執行迴圈直到t改變為止，步驟為來自模擬器用來模擬一循環的字元。

26.一旦取得的t值不同於主機堆疊之值，則複製該值並將該值寫入檔案random.log。

27.繼續執行迴圈直到完成標示於行42的131058次操作。該迴圈對應至行27的do迴圈，並且會增加迴圈計數值。

28.此處的drop代表主機堆疊中僅剩下最後一個項目，將此最後的項目排出並將檔案random.log關閉。

29.作為編碼風格的虛格。

30.註解：代表結束檔案處理。接下來的程式碼(除了行42之外)皆執行於目標機器。

31.作為編碼風格的虛格。

32.指出接下來程式碼將會執行節點(節點 0)的位置。接下來的程式碼將會於目標機器上執行。

33.執行節點0內的程式碼的冒號定義。

34.將兩個十六進位數值置放於資料堆疊中。在此程式碼結束處，數值$iff3係設置於S暫存器中而數值$5係設置於T暫存器中。

35.接下來的指令開始執行迴圈。

36.第一指令+*以段落 3.1 與子段落 3.1 的方式來執行。第二指令再次使迴圈返回行 38。

37.此行結束將於行 35 指定節點內執行的程式碼，也就是在節點 0 執行的程式碼。另外，此行控制行 36 於節點 0 內執行的冒號定義。

38.作為編碼風格的虛格。

39.131058 為將數值寫入輸出檔案 random.log 的總次數。

使用+*產生虛擬亂數的一些注意事項

上述用來產生虛擬亂數的虛擬碼演算法比先前用來產生乘法運算的演算法更簡單。儘管此演算法很簡單，但仍有些重要的變數值得進一步討論。

在用於本發明亂數產生器處理程序 300 的硬體中(也就是 SEAforth™ 24a 裝置)需要兩個週期來確保+*操作碼的正確執行。只執行單一週期的+*操作將會無法預測 T 暫存器中 HB 的變化。因此需要兩週期的+*操作來產生正確/期望的結果，第一週期用來執行加法而第二週期用來執行移位。另外，單一週期的+*操作的例子中將會使 T 暫存器的 HB 具有確定的變化。然而，透過在每個+*操作碼前加上·(nop)前導操作即可解決此問題。

產生虛擬亂數產生的特定方法對可產生序列的長度沒有任何限制。實際上，唯一的限制為可用來描述儲存於

T與S暫存器之值的位元數量。若該機器可用於此方法，則其他暫存器可用來輔助T與S暫存器數值的位元長度，則此方法產生虛擬亂數的位元長度僅受限於描述T的位元數量。

值的注意的是，設置於T暫存器的初始值必須為非零值，在此不詳細說明。為了瞭解原因，假設在執行產生虛擬亂數的虛擬碼演算法之前的T暫存器包含0。在此實施例中S暫存器的內容不重要。T暫存器的初始值為0代表T暫存器的每個位元皆為0。此時，一次+*疊代在T暫存器中產生其他0值，由於執行前T的LSB為0，因此T暫存器中所有的位元向右移位，其中T的最高位元為0。其他次+*疊代將會產生相同的結果。因此，設置於T暫存器的初始值必須為非零是很重要的，否則將會產生非常令人不感興趣的序列。值得注意的是，在執行乘法虛擬碼演算法之前設置於T暫存器中的零值將不會產生任何+*疊代，但是即使不改變T暫存器中的值，乘法運算的正確值會在T暫存器中。

值得注意的是，設置於S暫存器中的初始值必須為非零值，在此不詳細說明。為了瞭解原因，假設在執行產生虛擬亂數的虛擬碼演算法之前的T包含非零值且S包含零值。+*操作對T暫存器的作用僅將位元向右移位。若在任何+*疊代之前T中的最高位元為1，則最終的T值將會與所有未設定的位元相關。

在執行虛擬碼虛擬亂數產生演算法之前設置於T暫存

器中之值為該演算法的種子。假設S暫存器中的值在下列期間不會改變。T暫存器中的初始值 t_1 產生序列 Q_1。使用T暫存器中的初始值 t_2 產生序列 Q_2，其中 t_1 不等於 t_2。也就是每個 t_i 產生一序列 Q_i，其中每個 Q_i 為序列Q的子集合，Q為所有可產生序列的超集合(superset)。換句話說，不同的種子產生不同的序列，且使用相同的種子兩次將會產生相同的序列。因此，可能產生的序列主要取決於設置在T暫存器中的初始值。

每個連續+*疊代期間產生於T暫存器之值主要取決於S暫存器之值。由於在+*疊代期間S暫存器之值是固定的，因此選取適當的值是很重要的。所謂的”適當”取決於對所產生序列的期望特性。目前最適合S暫存器之值仍有待討論，因此選取S暫存器之值為程式設計師的設計選擇。然而，就發明人的觀點而論，可以透過目標裝置(例如 SEAforth™ 24a 裝置)上的蠻力(brute force)測試來選取S暫存器的最佳值。

改善序列品質

程式設計師考慮所有上述注意事項仍無法產生虛擬亂數序列。關於產生虛擬亂數方法的難題在於判斷虛擬亂數序列產生的時機。第一虛擬亂數是否為第一+*疊代的結果？或者第一虛擬亂數是否作為之後疊代的結果？甚至，此方法是否真的產生虛擬亂數序列？

第9圖顯示本發明產生虛擬亂數的實施例，就發明人

的觀點而論，基於各種NIST測試的結果，如18位元機器(例如SEAforth™ 24a)的NIST測試的定義，此方法無法單獨產生虛擬亂數序列。這不代表所產生的序列將無法通過較低信心水準處之NIST測試或是大於18位元的機器將無法產生較佳結果。就發明人的觀點而論，本發明較佳實施例將產生一序列，該序列幾乎產生從0至約2^{n-1}的每個值，而不是期望的虛擬亂數序列。然而，所產生的序列仍然有用，並且可改善所產生序列的品質。另外，這樣的改善將會改善大於18位元每個項目的序列品質。

許多技術可用來改善由+*操作碼所產生序列的品質。例如，最明顯的改善大概為增加T與S暫存器的位元寬度。這會增加所產生序列的長度，並且增加序列中所產生每個項目的位元寬度。當使用NIST測試進行分析時，此方法可單獨產生更好的結果。另外，當使用任何下列技術時將會改善序列品質。

第9圖之移位加法機制100在每個+*疊代之後將T暫存器之值視為序列中的虛擬亂數。一種對所產生序列的簡單改進為遮蔽每個項目的某些位元，使得在每次+*操作碼疊代之後，虛擬亂數序列中每個項目的位元長度具有比設置於T暫存器之值更少的位元長度。透過遮罩產生之值可以改進整體的序列品質。

改進所產生虛擬亂數序列的另一種方法與遮蔽輸出中的某些位元類似。可以將具有靜態或動態值之互斥或(exclusive or, XOR)應用至序列中的每個輸出值。這並

不會減少所產生數值的長度。

當然，仍具有許多此處未提及的技術可用來改進所產生序列品質。本發明雖以較佳實施例揭露如上，然其並非用以限定本發明的範圍，任何熟習此項技藝者，在不脫離本發明之精神和範圍內，當可做些許的更動與潤飾，因此本發明之保護範圍當視後附之申請專利範圍所界定者為準。

【圖式簡單說明】

TBL. 1-4 顯示在一組假設的+*(移位加法運算系統)範例中，SEAforth™ 24a 裝置中顯示之 T 暫存器與 S 暫存器值。

TBL. 5-10 顯示在一組假設的+*(移位加法運算系統)乘法運算範例中，SEAforth™ 24a 裝置中顯示之 T 暫存器與 S 暫存器值。

第 1 圖(先前技術)顯示在 Venture Forth™ 程式語言中的 32 種操作碼之圖表。

第 2 圖(先前技術)顯示在 SEAforth™ 24a 裝置中的每個核心之一般架構的方塊圖。

第 3a-b 圖(先前技術)顯示 SEAforth™ 24a 裝置中 18 位元寬的暫存器的方塊圖，其中第 3a 圖顯示實際位元配置，而第 3b 圖顯示概念位元配置。

第 4a-b 圖(先前技術)顯示暫存器內容的方塊圖，第 4a 圖顯示填滿四個·(nop)操作碼的擴充槽，而第 4b 圖顯

示填滿數字 236775(無號二進位(unsigned binary))的暫存器。

第 5a-b 圖(先前技術)分別顯示 SEAforth™ 24a 裝置中每個核心之返回與資料堆疊元件的示意圖，其中第 5a 圖顯示返回堆疊區域中的元件，而第 5b 圖顯示資料堆疊區域中的元件。

第 6 圖為本發明移位加法運算系統 100 的方塊圖，顯示與執行單一+*操作碼相關的所有可能動作。

第 7 圖顯示與第 6 圖有關的位元關係圖。

第 8 圖顯示本發明實施例基於移位加法運算之乘法處理程序 200 的流程圖。

第 9 圖顯示本發明實施例之亂數產生器的程式碼列表。

【主要元件符號說明】

100~移位相加運算系統

102~移位子處理程序

104~有條件相加子處理程序

200~基於移位加法運算之乘法處理程序

300~亂數產生處理程序

五、中文發明摘要：

一種虛擬亂數產生系統，包括處理器以及一邏輯。該處理器具有包含第一值之第一記憶體以及包含第二值之第二記憶體。該邏輯於循環條件成立時執行+*操作。

六、英文發明摘要：

A system for pseudorandom number generation. A processor is provided that has a first memory to hold a first value and a second memory to hold a second value. Then a logic performs a +* operation while a looping condition is true.

十、申請專利範圍：

1.一種虛擬亂數產生系統，包括：

一處理器，具有包含一第一值的一第一記憶體以及包含一第二值的一第二記憶體；以及

一邏輯，當一循環條件(loop condition)成立時執行一+*操作。

2.如申請專利範圍第1項之虛擬亂數產生系統，其中上述+*操作為上述處理器之一操作碼。

3.如申請專利範圍第1項之虛擬亂數產生系統，其中上述第一記憶體包括一第一組多個記憶體在上述處理器中，以及/或上述第二記憶體包括一第二組多個記憶體在上述處理器中。

4.如申請專利範圍第1項之虛擬亂數產生系統，更包括：一邏輯，用來遮蔽上述第一值之至少一位元。

5.如申請專利範圍第1項之虛擬亂數產生系統，更包括：一邏輯，對上述第一值與一第三值進行互斥或(XOR)運算。

6.如申請專利範圍第5項之虛擬亂數產生系統，更包括：一邏輯，動態改變執行上述+*操作之不同例子之間的上述第三值。

7.如申請專利範圍第1項之虛擬亂數產生系統，其中上述邏輯用來在每個上述+*操作之前執行·(nop)操作。

8.一種虛擬亂數產生方法，設置於具有一第一與第二記憶體之一處理器中，包括：

將一第一值設置於上述第一記憶體中；

將一第二值設置於上述第二記憶體中；以及

當一循環條件成立時執行一+*操作。

9.如申請專利範圍第8項之虛擬亂數產生方法，其中上述+*操作為上述處理器之一操作碼。

10.如申請專利範圍第8項之虛擬亂數產生方法，其中上述第一記憶體包括一第一組多個記憶體在上述處理器中，以及/或上述第二記憶體包括一第二組多個記憶體在上述處理器中。

11.如申請專利範圍第8項之虛擬亂數產生方法，更包括：在執行上述至少一疊代之後遮蔽上述第一值之至少一位元。

12.如申請專利範圍第8項之虛擬亂數產生方法，更包括在執行上述至少一疊代之後對上述第一值與一第三值進行互斥或運算。

13.如申請專利範圍第12項之虛擬亂數產生方法，更包括：動態改變不同上述疊代之間的上述第三值。

14.如申請專利範圍第8項之虛擬亂數產生方法，更包括：在每個上述+*操作之前執行·(nop)操作。

T → 0 0 0 1 → 1
S → 0 0 1 1 → 3

TBL. 1

T → 0 1 0 0 → 4
S → 0 0 1 1 → 3

TBL. 2

T → 0 0 1 0 → 2
S → 0 0 1 1 → 3

TBL. 3

T → 0 0 0 1 → 1
S → 0 0 1 1 → 3

TBL. 4

T → 0 0 0 0 1 0 0 1 → 9

S → 0 0 0 0 0 1 1 1 → 7

TBL. 5

T → 0 0 0 0 1 0 0 1

S → 0 1 1 1 0 0 0 0

TBL. 6

T → 0 1 1 1 1 0 0 1

S → 0 1 1 1 0 0 0 0

→

T → 0 0 1 1 1 1 0 0

S → 0 1 1 1 0 0 0 0

TBL. 7

T → 0 0 0 1 1 1 1 0

S → 0 1 1 1 0 0 0 0

TBL. 8

T → 0 0 0 0 1 1 1 1

S → 0 1 1 1 0 0 0 0

TBL. 9

T → 0 1 1 1 1 1 1 1

S → 0 1 1 1 0 0 0 0

→

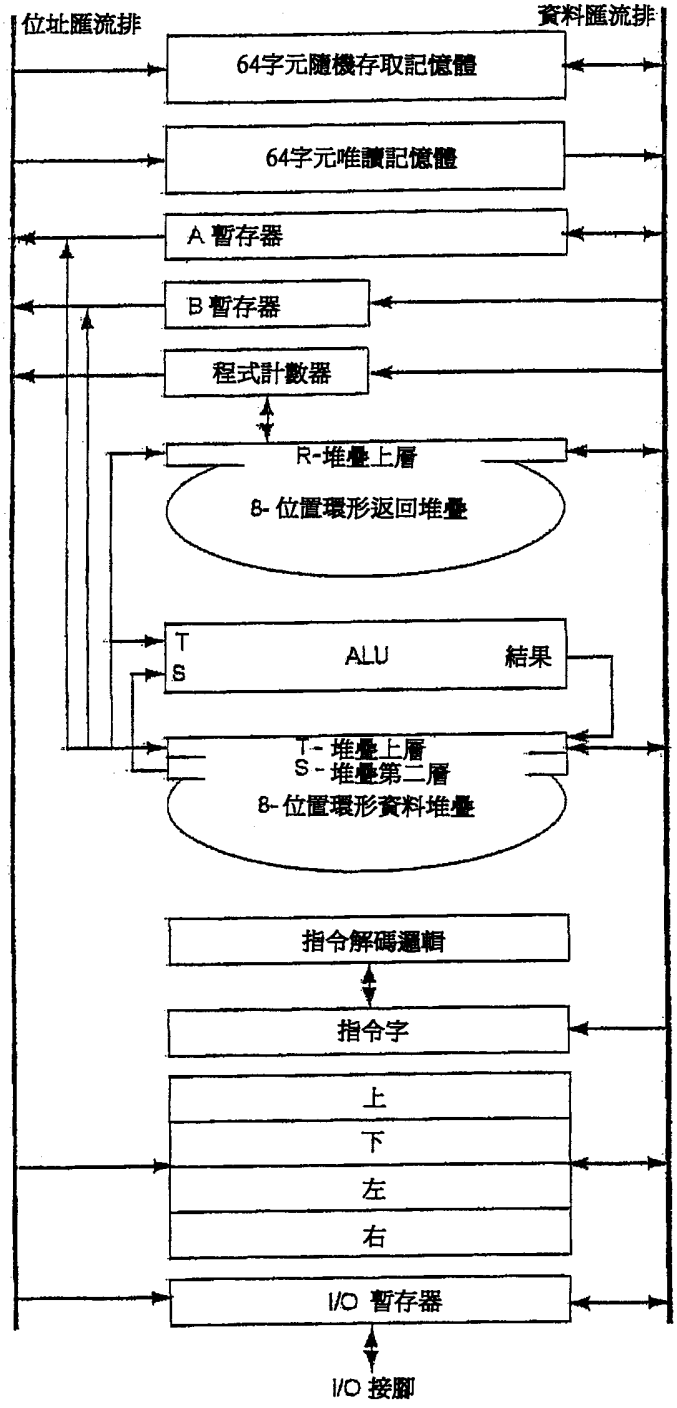
T → 0 0 1 1 1 1 1 1

S → 0 1 1 1 0 0 0 0

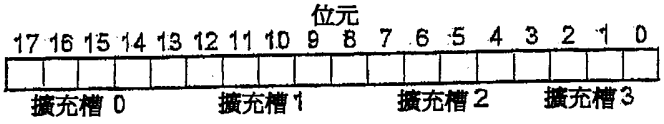
TBL. 10

十六進位（位元） 1 0		二進位（位元） 4 3 2 1 0	十六進位（位元） 1 0		二進位（位元） 4 3 2 1 0
0 0	; (return)	0 0 0 0 0	1 0	+*	1 0 0 0 0
0 1	::	0 0 0 0 1	1 1	2*	1 0 0 0 1
0 2	jump	0 0 0 1 0	1 2	2/	1 0 0 1 0
0 3	call	0 0 0 1 1	1 3	not	1 0 0 1 1
0 4	unext	0 0 1 0 0	1 4	+	1 0 1 0 0
0 5	next	0 0 1 0 1	1 5	and	1 0 1 0 1
0 6	If	0 0 1 1 0	1 6	xor	1 0 1 1 0
0 7	-If	0 0 1 1 1	1 7	drop	1 0 1 1 1
0 8	@p+	0 1 0 0 0	1 8	dup	1 1 0 0 0
0 9	@a+	0 1 0 0 1	1 9	pop	1 1 0 0 1
0 a	@b	0 1 0 1 0	1 a	over	1 1 0 1 0
0 b	@a+	0 1 0 1 1	1 b	a@	1 1 0 1 1
0 c	!p+	0 1 1 0 0	1 c	· (nop)	1 1 1 0 0
0 d	!a+	0 1 1 0 1	1 d	push	1 1 1 0 1
0 e	!b	0 1 1 1 0	1 e	b!	1 1 1 1 0
0 f	!a	0 1 1 1 1	1 f	a!	1 1 1 1 1

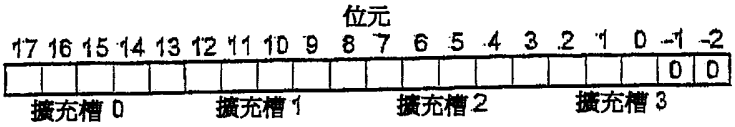
第1圖


位址匯流排
資料匯流排
64字元隨機存取記憶體
64字元唯讀記憶體
A 暫存器
B 暫存器
程式計數器
R-堆疊上層
8- 位置環形返回堆疊
T
S
ALU
結果
T - 堆疊上層
S - 堆疊第二層
8- 位置環形資料堆疊
指令解碼邏輯
指令字
上
下
左
右
I/O 暫存器
I/O 接腳

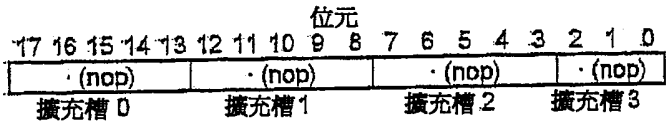
第2圖



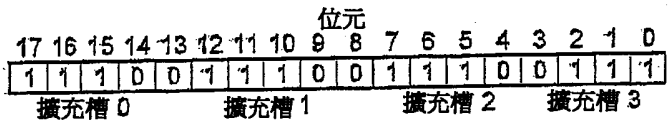
第3a圖



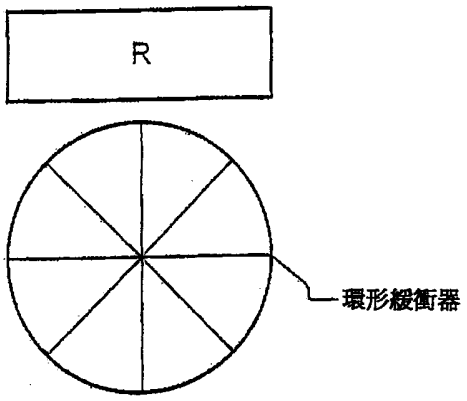
第3b圖



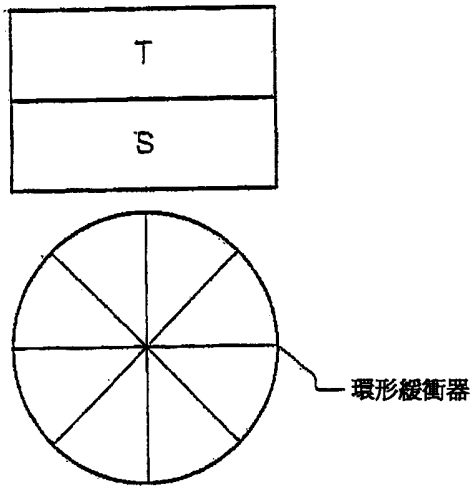
第4a圖



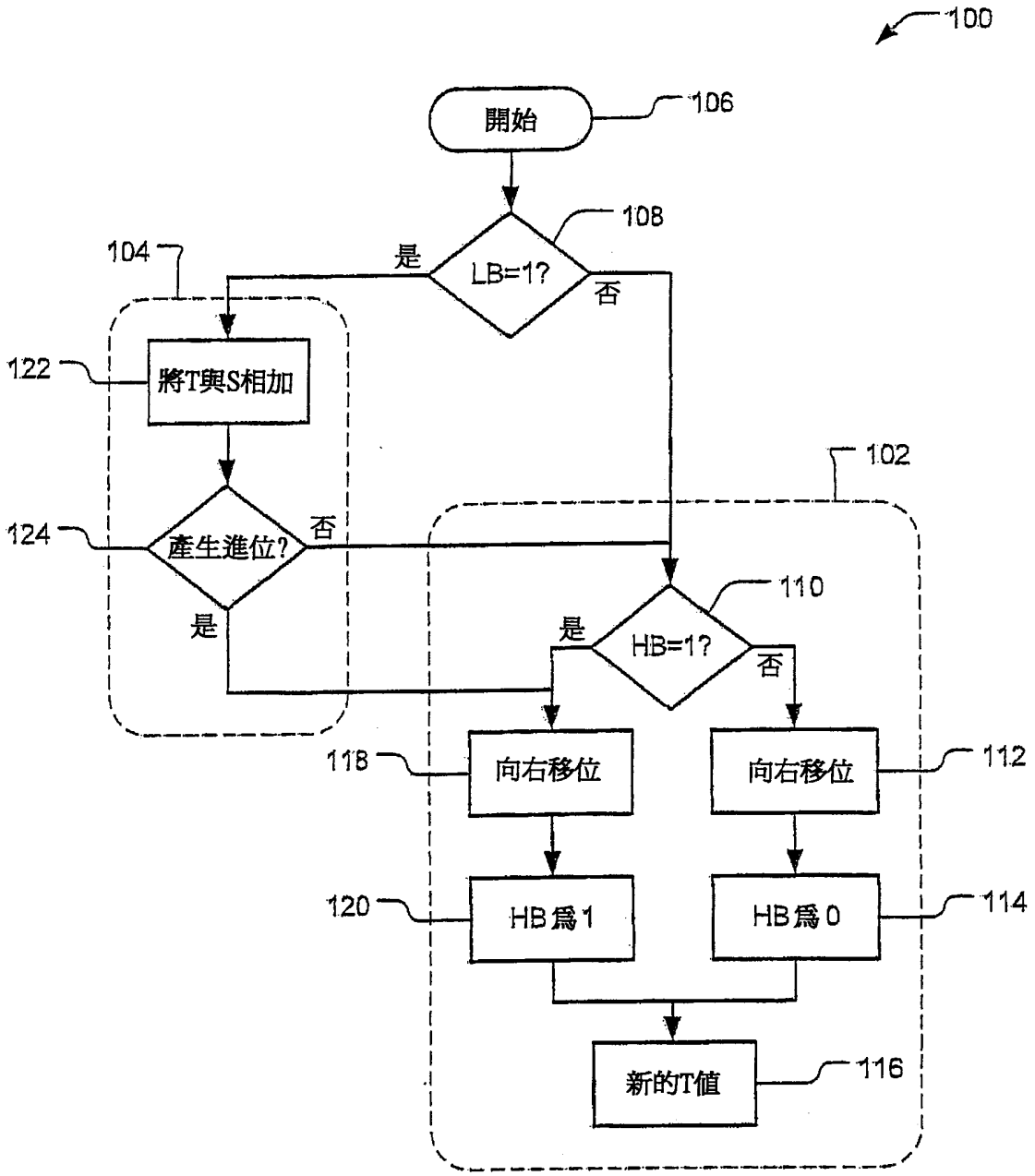
第4b圖



第5a圖



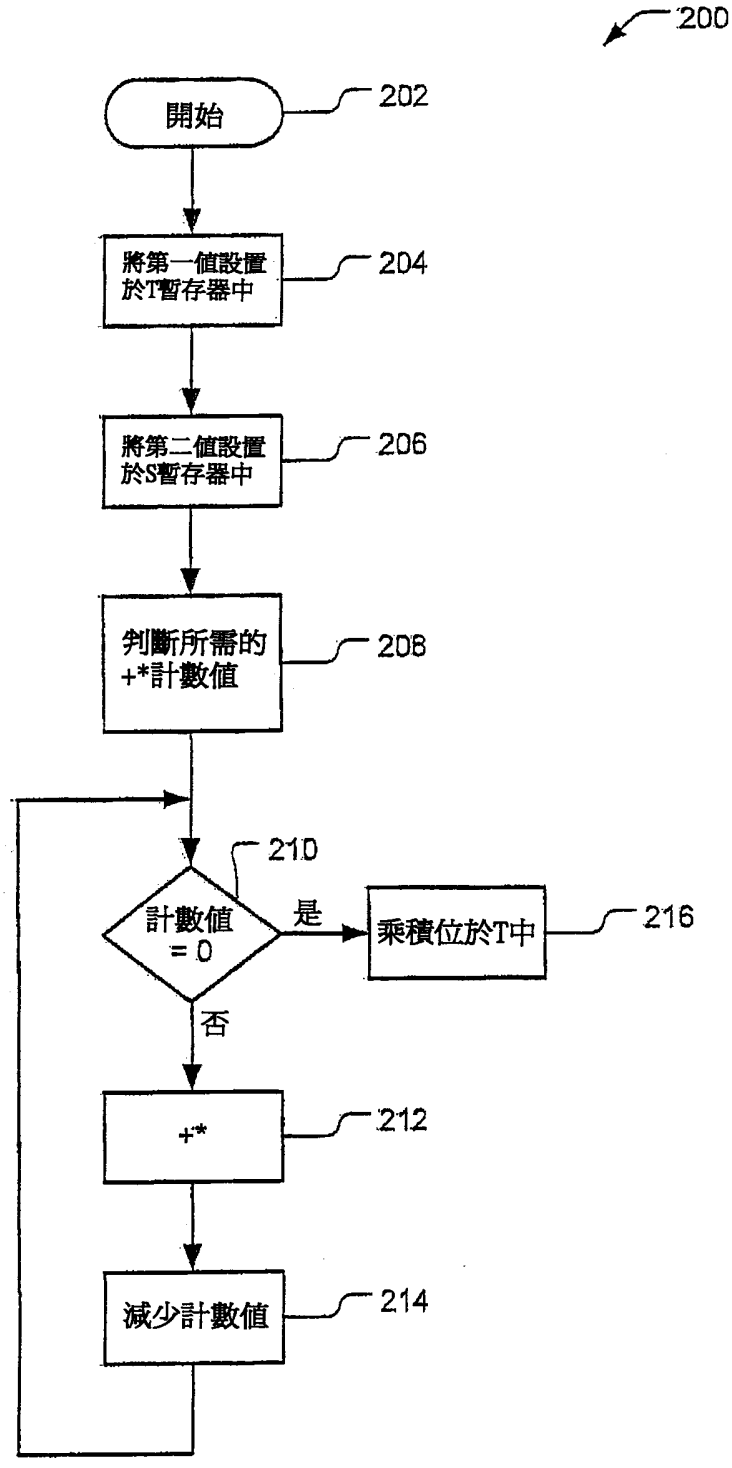
第5b圖


100
開始
106
108
LB=1?
是
否
104
122
將T與S相加
124
產生進位?
否
是
102
110
HB=1?
是
否
118
向右移位
112
向右移位
120
HB爲1
114
HB爲0
新的T值
116

第6圖

舊的T值			T+S			新的T值	
LB	HB		HB	進位位元		HB	HB-1
0	0		-	-		0	0
0	1		-	-		1	1
1	-		0	0		0	0
1	-		0	1		1	0
1	-		1	0		1	1
1	-		1	1		1	1

第7圖


200
開始
202
將第一值設置於T暫存器中
204
將第二值設置於S暫存器中
206
判斷所需的+*計數值
208
210
計數值 = 0
是
乘積位於T中
216
否
+*
212
減少計數值
214

第8圖

300

```
1  \ mytest.mf
2  include seaforth.f
3
4  \ ----- begin file handling -----
5  host
6
7  0 value fid
8  create "crlf 13 c, 10 c,
9
10 : open-log
11   s" random.log" r/w create-file throw to fid ;
12
13 : close-log
14   fid close-file throw  0 to fid ;
15
16 : log ( n - )
17   decimal
18   0 <# # # # # # # #> fid write-file throw
19   "crlf 2 fid write-file throw ;
20
21 : run ( n)
22   hex  0 node!  open-log  0
23   3 0 do  begin  step t @ tuck - until  loop
24   swap 0 do
25     begin  step t @ tuck - until
26     dup log
27   loop
28   drop close-log ;
29
30 \ ----- end file handling -----
31
32 0 {node
33 : test
34   $1fff3 $5
35   begin
36     +* again
37 node} runs test
38
39 131058 run bye
```

第9圖

七、指定代表圖：

（一）本案指定代表圖為：第（9）圖。

（二）本代表圖之元件符號簡單說明：無

八、本案若有化學式時，請揭示最能顯示發明特徵的化學式：

無