國立交通大學

電機與控制工程學系

碩　士　論　文

里德所羅門軟體解碼器之硬體加速器設計

Hardware Accelerator Design for Processor-based Reed-Solomon Decoder

研 究 生：簡嘉宏

指導教授：董蘭榮 博士

中 華 民 國 九 十 七 年 十 月

里德所羅門軟體解碼器之硬體加速器設計

Hardware Accelerator Design for Processor-based Reed-Solomon Decoder

研 究 生：簡嘉宏　　Student：Chia-Hung Chien

指導教授：董蘭榮 博士　　Advisor：Lan-Rong Dung

國 立 交 通 大 學

電 機 與 控 制 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Department of Electrical and Control Engineering

College of Electrical Engineering and Computer Science

National Chaio-Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

in

Eletrical and Control Engineering

October 2008

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 七 年 十 月

里德所羅門軟體解碼器之硬體加速器設計

研 究 生：簡嘉宏　　　　　　　　指導教授：董蘭榮 博士

國立交通大學電機與控制工程學系

摘要

本篇論文提出了一套里德所羅門解碼器以處理器為基礎外掛硬體加速器的架構，主要是為了適應於各種不同的里德所羅門碼的規格，也是為了解決使用處理器運算里德所羅門碼裡的伽羅瓦場乘法花費太多的時間。因為處理器的處理速度越來越快，使得一些不需要高處理效率(Throughput)的功能可以由處理器來做運算，也可以達到要求。也因此以前需要使用許多的ASIC才能做許多不同的工作，而處理器卻只需一個就可以做不同的工作。所以現在的趨勢也慢慢的將許多功能交由處理器來運算。在里德所羅門碼中所有的運算都是建立在伽羅瓦場中，然而處理器對於伽羅瓦場的乘法運算沒有特別的方法可以使用，因此處理器在做伽羅瓦場的乘法運算相當耗時。因為這個原因，所以將在里德所羅門解碼中使用到最多伽羅瓦場乘法運算的方塊以硬體加速器處理以增加運算速度。由於硬體有一套特殊的伽羅瓦場乘法運算，因此加快了運算的速度。在里德所羅門解碼中的尋找錯誤症狀和尋找錯誤位置這兩部份使用到了許多伽羅瓦場的乘法運算，因此將這兩部份以硬體加速器的方式實現，又因為這兩部份的運算很類似，所以將這兩部份以同一套硬體來實現，並且使用摺疊硬體架構，減少外掛硬體加速器的面積，也讓外掛硬體加速器更具有擴充性。

Hardware Accelerator Design for Processor-based Reed-Solomon Decoder

Student：Chia-Hung Chien　　Advisor：Dr. Lan-Rong Dung

Department of Electrical and Control Engineering
National Chiao Tung University

ABSTRACT

This thesis presents a processor-based with hardware accelerator for Reed-Solomon decoder. The main reason doing this is to adapt different types of Reed-Solomon code, also to solve the problem of using too much time do the multiply of Galois field in Reed-Solomon code. Now with one processor could complete many works which needed many ASIC before. But processor operate multiplication of Galois field is very slow. So we need hardware accelerator speed up multiplication of Galois field. The reconfigurable Reed-Solomon decoder is targeted on xDSL applications. Under the requirement of the throughput rate, we fold the Reed-Solomon decoding with the minimal number of processing elements (PEs) while the complexity of scheduler is low. The folded architecture is suitable for array processors whose processing rate is not necessary to be optimal. The proposed reconfigurable decoder is highly scalable as the application parameters change.

誌謝

本篇論文得以順利完成，首先要感謝的是我的指導教授──董蘭榮教授，在碩士班的兩年間，董教授不厭其煩地指導我，當我陷入瓶頸時，董教授亦適時地指點我正確的方向，讓我不至於常常失焦，並且及時做出修正，並且提供非常豐富的資源，讓我能好好潛心於學習研究，讓我在這兩年間獲益良多。

感謝整個實驗室研究團隊的支持與幫助，無論是精神上的鼓勵，或是研究上的協助，都讓我在這段研究的日子裡得以堅持。謝謝學長們的指導，謝謝同學們的陪伴，謝謝學弟們的加油打氣，這些都是我心靈上最溫暖的寄託，研究上最堅強的後盾。最後要感謝我的家人的支持，有了你們的鼓勵，使我無後顧之憂，才能夠安心地完成碩士班學業。

謹將此論文獻給所有關心我的人，在此致上最深的謝意。

章節目錄

中文摘要..i

英文摘要..ii

誌謝..iii

章節目錄..iv

圖目錄..vi

表目錄..viii

第一章　簡介..1

1-1　研究動機..1

1-2　章節規劃..2

第二章　研究背景..3

2-1　伽羅瓦場(Galois Field)定義介紹..3

2-1.1 本質多項式與伽羅瓦場的建立..3

2-1.2 伽羅瓦場的乘加運算..5

2-2　里德所羅門(Reed-Solomon)碼定義介紹..5

2-3　里德所羅門(Reed-Solomon)編碼演算法..6

2-4　里德所羅門(Reed-Solomon)編碼演算法..7

2-4.1 收到信號的錯誤症狀(Syndrome)計算..10

2-4.2 尋找錯誤位置多項式之方法..10

2-4.3 尋找錯誤位置之方法(Chien Search)..15

2-4.4 尋找錯誤大小的佛尼(Forney)演算法..16

第三章　摺疊演算法之推導與設計及 Xilinx EDK 整合系統環境介紹..17

3-1　摺疊演算法的硬體架構介紹..17

3-2　摺疊演算法實現硬體之方法與特色..22

3-3　Xilinx EDK 整合系統背景......23

3-4　MicroBlaze 架構......24

3-5　On-chip peripheral Bus(OPB)......25

第四章　　硬體實現架構......29

4-1　整合里德所羅門解碼器與完整電路架構......29

4-2　伽羅瓦場乘法器......30

4-3　處理錯誤症狀(Syndrome)之硬體摺疊架構......33

4-3.1 乘加運算器及暫存器檔案......34

4-3.2 位址產生器的設計......38

4-4　尋找錯誤位置之硬體摺疊架構......40

4-4.1 一般方法之尋找錯誤位置硬體摺疊架構......40

4-4.2 一般方法之尋找錯誤位置硬體摺疊架構之乘加運算器及暫存器規劃......41

4-4.3 變形之尋找錯誤位置硬體摺疊架構......44

4-4.4 變形之尋找錯誤位置硬體摺疊架構之乘加運算器及暫存器規劃......45

4-5　尋找錯誤症狀硬體架構與尋找錯誤位置硬體架構的合併......50

4-5.1 尋找錯誤症狀-位置(Syndrome-Chien search)的硬體架構......50

4-5.2 尋找錯誤症狀-位置(Syndrome-Chien search)硬體架構之乘加運算器與暫存器規劃......51

4-6　尋找錯誤位置多項式及錯誤大小評估多項式之軟體實現......54

4-7　尋找錯誤大小之軟體實現......56

4-8　里德所羅門解碼之軟、硬體系統整合......56

4-9　實現結果......58

第五章　　結論與未來發展......62

參考文獻：......64

圖目錄

圖 1 非二進位環狀碼（Cyclic code）的編碼（Encoding）電路 7

圖 2 里德所羅門解碼流程圖 9

圖 3 錯誤症狀運算電路 10

圖 4 LFSR 架構迴圈形式.. 11

圖 5 基本運算單元(Processing Element)............................. 18

圖 6 基本運算單元陣列 19

圖 7 基本運算單元陣列之時序規劃.................................... 20

圖 8 基本運算單元陣列之摺疊架構 21

圖 9 摺疊架構之乘加器與暫存器規劃 21

圖 10 摺疊架構之暫存器定址 22

圖 11 MicroBlaze 處理器架構圖...................................... 25

圖 12 MicroBlaze 系統的 OPB 仲裁器示意圖........................... 26

圖 13 OPB 讀取時序圖.. 28

圖 14 OPB 寫入時序圖.. 28

圖 15 里德所羅門解碼器軟、硬體概略圖 30

圖 16 兩種互斥樹狀架構 32

圖 17 建立在 GF(2^4)之非規則全平行乘法器硬體架構 33

圖 18 錯誤症狀計算之陣列硬體架構 34

圖 19 更正能力 t=8 之錯誤症狀計算陣列 35

圖 20 錯誤症狀計算陣列之時序規劃 36

圖 21 錯誤症狀計算陣列之摺疊架構 37

圖 22 錯誤症狀計算硬體之乘加器與暫存器規劃 38

圖 23 錯誤症狀計算硬體之位址產生器................................39
圖 24 更正錯誤能力 t 之尋找錯誤位置的硬體架構........................40
圖 25 更正錯誤能力 t=8 之尋找錯誤位置的硬體架構......................41
圖 26 八次尋找錯誤位置器陣列之時序規劃..............................42
圖 27 八次尋找錯誤位置器之摺疊架構..................................43
圖 28 尋找錯誤位置器硬體之乘加器與暫存器規劃........................44
圖 29 變形之尋找錯誤位置陣列硬體架構................................45
圖 30 更正錯誤能力 t 之變形尋找錯誤位置的硬體架構....................46
圖 31 更正錯誤能力 t 之變形尋找錯誤位置陣列之時序規劃................47
圖 32 更正錯誤能力 t 之變形尋找錯誤位置之摺疊架構....................48
圖 33 變形尋找錯誤位置硬體之乘加運算器與暫存器規劃....................48
圖 34 變形尋找錯誤位置之位址產生器..................................50
圖 35 尋找錯誤症狀-位置陣列之摺疊架構................................51
圖 36 尋找錯誤症狀-位置硬體之乘加運算器與暫存器規劃....................52
圖 37 位址產生器..54
圖 38 BM 與 ME 演算法運算時間比較.....................................55
圖 39 控制暫存器..57
圖 40 狀態暫存器..57
圖 41 控制&狀態暫存器的連接..58
圖 42 尋找錯誤症狀之 RTL 模擬..59
圖 43 尋找錯誤位置模擬..59
圖 44 尋找錯誤位置模擬..59
圖45 利用狀態暫存器和處理器溝通.....................................60

表目錄

表 1　利用本質多項式 $h(x)=1+x^2+x^3+x^4+x^8$ 建立之 $GF(2^8)$ 4

表 2　OPB 介面訊號一覽表 27

表 3　三種硬體使用的邏輯閘個數 61

表 4　里德所羅門解碼的各功能方塊所花時間表 61

第一章　簡介

1-1 研究動機

隨著時代不斷的改變與進步，科技越來越發達，無線通訊應用到的地方也越來越多，不僅在是在音訊方面的傳輸，甚至在視訊方面的傳輸也越來越多，例如數位電視廣播(DVB)及視訊電話等等。

數位訊號處理器(DSP Processor)大部分用作於處理影像資料，但是隨著科技的快速發展，處理器的速度一直提升，就有人提出利用數位訊號處裡器來做通訊方面的運算，而以往通訊方面的運算都是以 ASIC 來實現，假設一個系統需要很多個功能方塊，就必須要有不同功能的硬體，如果有些時候只會有其中一部分的功能方塊在使用，則在這個時間其他沒運作的硬體就佔了空間。因此有人提出利用數位訊號處理器，當要運算怎樣的功能時，只要把該功能的程式送到數位訊號處理器就可以達到要的功能，這樣一個數位訊號處理器就可以做許多種的功能，在硬體的使用上可以減少很多。

由於使用軟體做某些運算會花許多時間去處理，而讓系統的大部份的時間都浪費在這些運算，大大的降低了系統整體的效能。像是本篇論文主要討論的里德所羅門解碼用到的伽羅瓦場乘法，必須先建立兩個表，當兩數要做伽羅瓦場乘法時，則要先去查表轉換成指數形式後，做相加動作，這樣才完成一個伽羅瓦場的乘法；而當是做伽羅瓦場的乘累加運算，要先查表轉換成指數形式，做相加動作，接著還要轉回原本的形式，然後做 XOR，這樣才完成一個乘累加的動作。光是計算一次乘累加就要來來回回轉換，浪費許多運算時間。

因此我們希望將這部分用硬體的方式來實現，提高軟體的效能，但是又不需使用太多空間的硬體，因此有了本篇論文的研究，利用摺疊的方法以及修改硬

體架構讓硬體可以共同使用，降低硬體的空間。

1-2 章節規劃

在此小節中先對本篇整個論文架構作個概略性的介紹。

第一章 緒論

提出論文主題、想要解決的問題及其主要應用所在。

第二章 研究背景

介紹里德所羅門碼（Reed-Solomon Code）之基本運作原理：包含了編碼與解碼，但焦點在於解碼過程中的定義與不同演算法之間的比較，並提出一些相關硬體實現時所要考慮的問題。

第三章 摺疊演算法之推演與建立及 Xilinx EDK 整合系統環境介紹

提出一套摺疊演算法之理論，經由圖論上的推導，讓人擁有明確的流程可以遵循，並指出採用摺疊演算法後的架構，展現了可重複使用的特性。

第四章 硬體實現

先架構我們要的里德所羅門解碼器，包然軟體、硬體部分，接著介紹解碼器中硬體部分的架構，及軟體部份的架構，再來是整合軟、硬體使得解碼器動作無誤，最後將模擬結果呈現出來。

第五章 結論

本篇論文之結語與未來展望。

第二章　研究背景

在深入探討本篇研究論文之前，本章先介紹一些基本的里德所羅門（Reed-Solomon）的編解碼方式，還有一些將會運用到的基本數學理論。首先，第一節提到的是里德所羅門編解碼所建立在伽羅瓦場（Galois Field）其構成的原理，以及在場中的乘加運算；第二節則開始介紹里德所羅門編解碼的基本定義，進而分別在第三節和第四節中介紹其編碼與解碼的演算法，其中尤其以解碼較為複雜，必須由許多步驟接續處理完成。

2-1 伽羅瓦場(Galois Field)定義介紹

[1]若有一個場，其中所存在的元素數目是有限值，則稱之為有限場或伽羅瓦場。有限場是整個錯誤更正碼（Error-Control Coding）最有用、最基本且最重要的觀念。若一個場 F 為有限，則將 F 場的元素個數定義為 F 的階數。一個 q 階的有限場表示為 GF(q)。

在這節中主要探討伽羅瓦場的構成方法，這將運用一些實數系中較不會用到的數學定義及定理。另外，在此場中乘加運算的動作原理，也會一併作個簡介。

2-1.1 本質多項式與伽羅瓦場的建立

定義　2.1　不可化簡多項式（Irreducible polynomial）

假設一個多項式 f(x)除了 1 和本身之外沒有其他因式，則稱這種多項式為不可化簡的多項式。

定義　2.2　本質多項式（Primitive polynomial）

一個 n 大於一階的不可化簡多項式，假若符合 $m < 2^n - 1$ 且其並不是 $1 + x^m$ 之因式，則將其稱之為本質多項式。

表 1　　利用本質多項式$h(x)=1+x^2+x^3+x^4+x^8$建立之$GF(2^8)$

Word	Polynomial in x (modulo h(x))	Power of α
00000000	0	---
10000000	1	α^0
01000000	x	α^1
00100000	x^2	α^2
00010000	x^3	α^3
00001000	x^4	α^4
00000100	x^5	α^5
⋮	⋮	⋮
00110000	$x^2+x^3\equiv x^{27}$	α^{27}
00011000	$x^3+x^4\equiv x^{28}$	α^{28}
00001100	$x^4+x^5\equiv x^{29}$	α^{29}
⋮	⋮	⋮
00011011	$x^3+x^4+x^6+x^7\equiv x^{251}$	α^{251}
10110101	$1+x^2+x^3+x^5+x^7\equiv x^{252}$	α^{252}
11100010	$1+x+x^2+x^6\equiv x^{253}$	α^{253}
01110001	$x+x^2+x^3+x^7\equiv x^{254}$	α^{254}

[1]利用本質多項式（Primitive polynomial）去建立一個$GF(2^n)$伽羅瓦場比利用非本質多項式（Non-primitive polynomial）來的容易許多。令α代表x對h(x)取餘數結果的字元（i.e. x mod h(x)），其中h(x)必須是n階的本質多項式，則在同一數系下所有非零的字元都可以被表示成α次方的形式：$\alpha^i \leftrightarrow x^i \bmod h(x)$，而$\alpha$稱為伽羅瓦場$GF(2^n)$之本質元素，這個特性讓伽羅瓦場的乘法運算變的更為簡單。除此之外，在這個有限的伽羅瓦場中一共會有2^n-1個截然不同的非零元素，可以表示為2^n-1個α的連續次方，即為{1, α, α^2, ⋯ ,

α^{2^n-2} }。**表 1** 即是一個利用本質多項式 $h(x)=1+x^2+x^3+x^4+x^8$ 所建立出 $GF(2^8)$ 的對照表。

2-1.2 伽羅瓦場的乘加運算

伽羅瓦場的乘法運算，簡單來說，就是直接將伽羅瓦場裡的代表字元作一般實數系相乘的動作，所得到的字元再依照二進位與伽羅瓦場的轉換對照表，找出其相對應的二進位值。例：$\alpha \cdot \alpha^2 = \alpha^3 = 00010000$。

反之，伽羅瓦場的加法運算就沒有像乘法運算這麼單純，可以直接利用該字元進行類似實數系的運算，而是必須先找出運算字元所對應到的二進位表示法的數，然後將兩個二進位表示法的數作 XOR 的運算，再利用所得到的值去尋找其所對應的伽羅瓦場的字元。例：$\alpha^3+\alpha^4 = 00010000 \oplus 00001000 = 00011000 = \alpha^{28}$。

2-2 里德所羅門(Reed-Solomon)碼定義介紹

里德所羅門碼是在西元 1960 年由 I. Reed 以及 G. Solomon 在麻省理工學院（M.I.T.）實驗室所共同發明的，這是一個建立在伽羅瓦場的編碼方式，並可以將其視為另一種 BCH 編碼的特殊情況，自從里德所羅門碼被發明之後，近幾年來已經被廣泛的應用在各種方面。

定義　2.3　里德所羅門碼（建立於 $GF(2^m)$ 之元素）

假設 α 為 $GF(2^m)$ 之本質元素，對任意一個正整數 $t \le 2^m-1$，必定存在一組建立於伽羅瓦場 $GF(2^m)$ 可去除 t 個錯誤符號（Symbol）的里德所羅門碼[2]，其各項參數如下：

$$n = 2^m - 1 \quad \text{（式 2-1）}$$

$$n-k = 2t \quad \text{（式 2-2）}$$

$$d_{\min} - 1 = 2t = n-k \quad \text{（式 2-3）}$$

其中 m 表示字碼長度，n 為編碼後字碼數目，k 則是來源字碼數目。

以 RS (255, 223)這組里德所羅門碼為例：經由 n = 255、k = 223 進而可得知 m = 8、t = 16、d_{min} = 33。同時，這一組規格的里德所羅門碼也是美國太空總署（NASA）給衛星和太空通訊所採用的標準編碼。

2-3 里德所羅門(Reed-Solomon)編碼演算法

當開始進行編碼之前，首先介紹字碼產生多項式（Codeword generator polynomial)，表示如下：

$$g(x)=\prod_{i=0}^{2t-1}(x+\alpha^{m_0+i})=g_0+g_1x+\cdots+g_{2t-1}x^{2t-1}+x^{2t}$$

$$where\ \ g_i\in GF(2^m) \qquad \text{（式 2-4）}$$

一般來說，m_0 的典型值為 0 或 1。此外，值得一提的是連續 2t 個α的次方α^{m_0}、α^{m_0+1}、…、α^{m_0+2t-1}均為字碼產生多項式 g(x)的根。

假設需要編碼的 k 個 m 位元的來源訊息的多項式表示為：

$$m(x)=m_0+m_1x+\cdots+m_{k-1}x^{k-1}\quad where\ \ m_i\in GF(2^m) \qquad (2.5)$$

然而這其中的 k 就是訊息長度，並且根據前小節里德所羅門碼的定義：$k=n-2t$，即原先的訊息長度 k 等於編碼後長度 n 扣掉兩倍的除錯能力 t。

接著，將訊息多項式乘上x^{2t}，再除上里德所羅門碼其專屬的字碼產生多項式 g(x)，則可得到一個餘式 b(x)如下：

$$x^{2t}m(x)=a(x)g(x)+b(x)$$

$$where\ \ b(x)=b_0+b_1x+\cdots+b_{2t-1}x^{2t-1} \qquad \text{（式 2-6）}$$

其中$x^{2t}m(x)$的最低次方為 2t，而 b(x)的最高次方為 2t-1，則$x^{2t}m(x)+b(x)$所組成的 2t +k-1=n-1 次方多項式，就是編碼完成的字碼多項式 c(x)，且編碼後

長度即為 n。此外，由於$c(x)=x^{2t}m(x)+b(x)=a(x)g(x)$，為 g(x)的因式，所以連續 2t 個$\alpha$的次方$\alpha^{m_0}$、$\alpha^{m_0+1}$、…、$\alpha^{m_0+2t-1}$也均為 c(x)的根。編碼電路的電路圖如圖 1 所示[2]。

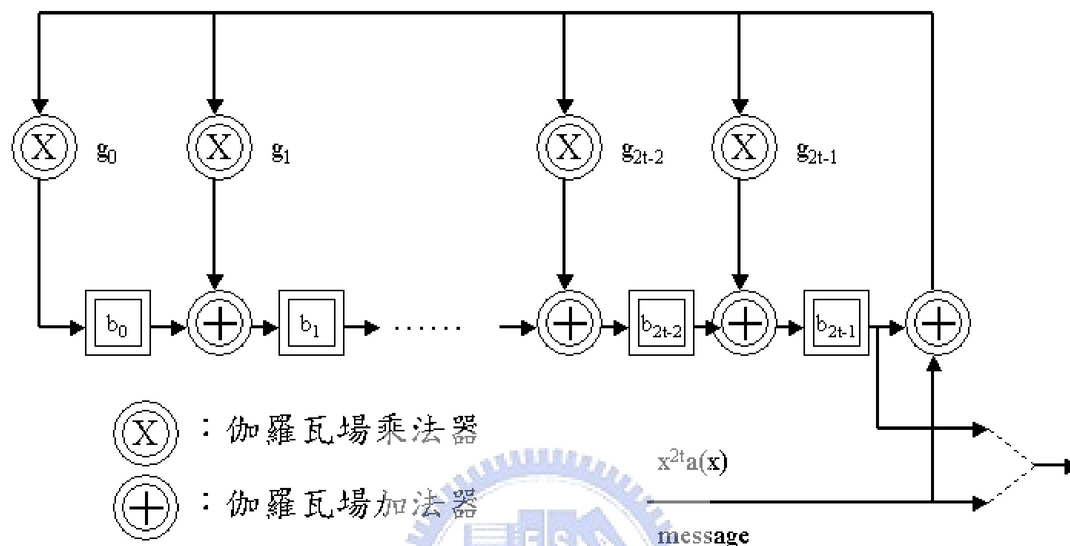


圖 1 非二進位環狀碼（Cyclic code）的編碼（Encoding）電路

2-4 里德所羅門(Reed-Solomon)編碼演算法

整個里德所羅門碼比較煩瑣的步驟及主要的問題都是出現在解碼的過程，再加上本篇論文所提出的主要是解碼器架構，故在此小節作個較為詳盡的解碼演算法介紹。

一開始先簡介一下整個解碼的過程。令傳送的碼和接收到的碼分別表示為：

$$c(x)=c_0+c_1x+\cdots+c_{n-1}x^{n-1}, c_i\in GF(2^m) \qquad \text{（式 2-7）}$$

$$r(x)=r_0+r_1x+\cdots+r_{n-1}x^{n-1}, r_i\in GF(2^m) \qquad \text{（式 2-8）}$$

故因為傳輸而產生的雜訊就可以表示為：

$$e(x)=r(x)-c(x)=e_0+e_1x+e_2x^2+\cdots+e_{n-1}x^{n-1} \qquad \text{（式 2-9）}$$

其中 $e_i=r_i-c_i$ 也是屬於 GF(2^m)中的元素。假設這個錯誤多項式有 v 個根，即

代表存在有 v 個錯誤位置的訊息，其錯誤位置和相關大小的表示法定義如下：

Error Values $Y_l = e_{j_l} \quad for \quad l = 1,2,\ldots,v$ （式 2-10）

Error Locators $X_l = \alpha^{j_l} \quad for \quad l = 1,2,\ldots,v$ （式 2-11）

也就是說，在錯誤位置X_l處出現錯誤值Y_l，而以錯誤位置為根所形成的錯誤位置多項式可以表示成：

$$\begin{aligned}\Lambda(x) &= (1 - X_1x)(1 - X_2x)\cdots(1 - X_vx) \\ &= \prod_{i=1}^{v}(1 - X_ix) \\ &= \Lambda_0 + \Lambda_1x + \Lambda_2x^2 + \cdots + \Lambda_vx^v\end{aligned} \quad \text{（式 2-12）}$$

有一種解碼 BCH Code 或是 RS Code 的方式[1]，其整個解碼過程就好比去解出兩個關鍵的方程式：

$$\begin{bmatrix} S_1 & S_2 & \ldots & S_{t-1} & S_t \\ S_2 & S_3 & \ldots & S_t & S_{t+1} \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ S_t & S_{t+1} & \ldots & S_{2t-2} & S_{2t-1} \end{bmatrix} \cdot \begin{bmatrix} \Lambda_t \\ \Lambda_{t-1} \\ \vdots \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ \vdots \\ -S_{2t} \end{bmatrix} \quad \text{（式 2-13）}$$

$$\begin{bmatrix} X_1 & X_2 & \ldots & X_{v-1} & X_v \\ X_1^2 & X_2^2 & \ldots & X_{v-1}^2 & X_v^2 \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ X_1^v & X_2^v & \ldots & X_{v-1}^v & X_v^v \end{bmatrix} \cdot \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ \vdots \\ Y_v \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ \vdots \\ S_v \end{bmatrix} \quad \text{（式 2-14）}$$

從方程式（2.13）中可以清楚看出，假若能夠從接受到的信號中找出錯誤的症狀（Syndrome），便可以利用方程式（2.13）求出剛提到的錯誤位置多項式（Error-Locator Polynomial）的係數。有了這些係數，利用土法煉鋼的方式，把所有伽羅瓦場裡的元素都代入檢查，找出能夠使其為零的元素，便為其根，這

也就是 Chien Search 的演算法。利用 Chien Search 找出錯誤位置(X_l)之後，便可以利用方程式 (2.14) 找出錯誤的大小(Y_l)，而後再把找出來錯誤的大小和位置加回原先收到信號上，整個解碼過程才算完整結束。其實這整個觀念就是熟知的 Peterson-Gorenstein-Zierler Decoding Algorithm (P-G-Z 解碼演算法)，整個解碼流程圖如**圖 2** 所示。

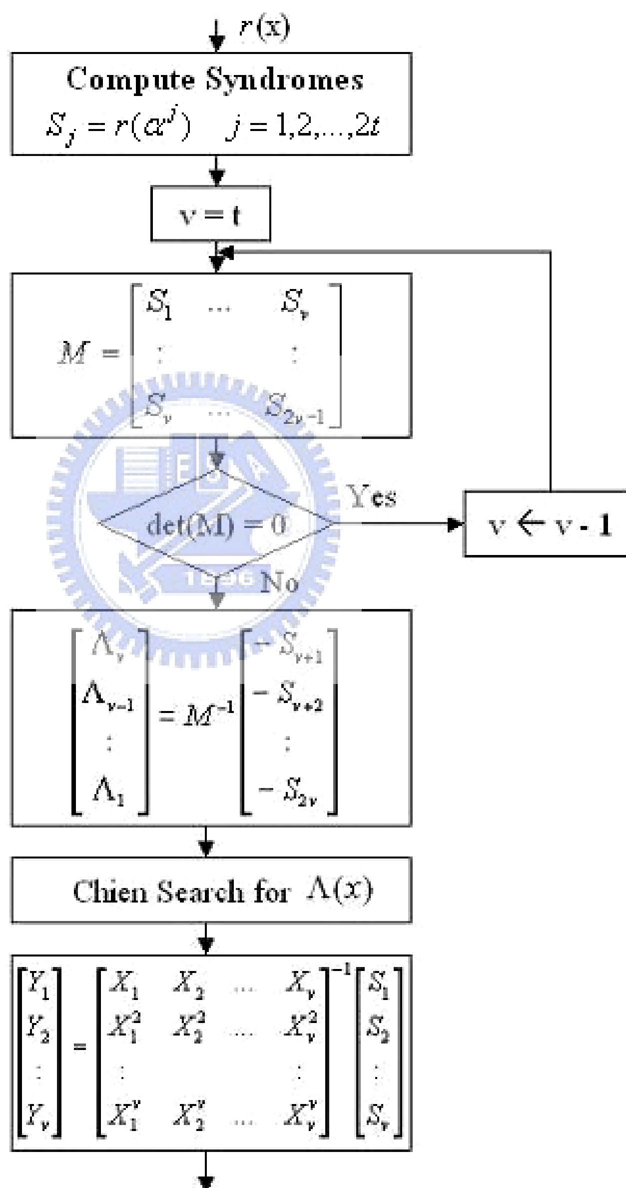


圖 2 里德所羅門解碼流程圖

2-4.1 收到信號的錯誤症狀(Syndrome)計算

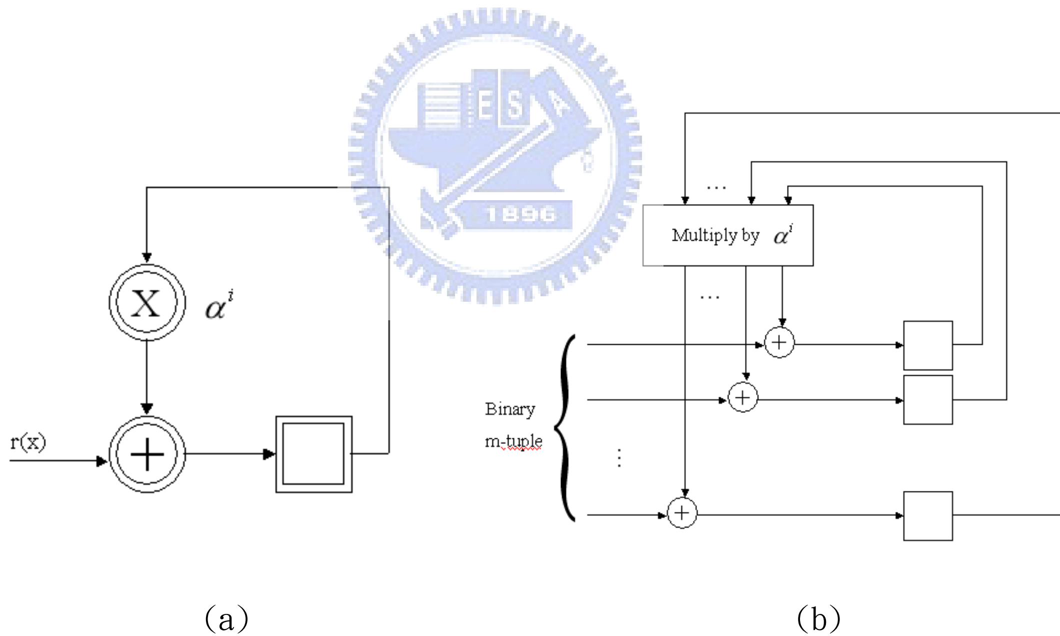
在解碼的過程中，錯誤症狀的計算是所要進行的第一步驟。由前面的章節得知，接收到的信號可表示為 $r(x)=r_0+r_1x+\cdots+r_{n-1}x^{n-1}$，從 2.2 小節得知字碼產生多項式（Codeword Generator Polynomial）的根為 $\alpha^{m_0}\sim\alpha^{m_0+2t-1}$，其中 t 代表除錯的能力，又因為 $c(x)=m(x)g(x)$，所以便可以得到以下的關係：

$$r(\alpha^{m_0+i})=c(\alpha^{m_0+i})+e(\alpha^{m_0+i})=e(\alpha^{m_0+i})=\sum_{j=0}^{n-1}e_j\alpha^{(m_0+i)j},\ i=0,2,\ldots,2t-1 \quad \text{（式 2-15）}$$

因此，所收信號的錯誤症狀即為 $S_i=r(\alpha^{m_0+i})$，其多項式表示法如下：

$$S(x)=\sum_{j=0}^{2t-1}S_j\cdot x^j \quad where \quad S_j=\sum_{i=1}^{n}r_{n-i}\cdot(\alpha^{m_0+j})^{n-i} \qquad \text{（式 2-16）}$$

整個過程就如同乘累加運算一樣，運算電路如圖 3 所示。



（a）　　　　　　　　　　（b）

圖 3　錯誤症狀運算電路

（a） 在 GF(2^m)表示法 （b）二進位表示法

2-4.2 尋找錯誤位置多項式之方法

由於前述之 P-G-Z 解碼演算法，在求錯誤位置多項式的運算過程關係到一些逆矩陣的運算，所以較沒效率，速度也會減慢，相對變的比較複雜，其複雜度

是跟 t^3 成正比，故這種方法只比較適用於較小的除錯能力架構中。而後人想了幾種加快這部分速度的辦法，包括：Berlekamp-Massey 疊代演算法，以及最大公因式演算法（Euclidean Algorithm)。

2-4.2.1 Berlekamp-Massey 疊代演算法

西元 1967 年 E. Berlekamp 提出一種極有效率之演算法，同時適合使用於解碼 BCH code 及 RS code，再加上後來 Massey 進一步的改良，其複雜度僅隨著 t^2 成長[1]，所以是一個求取錯誤位置多項式比較有效率的演算法，因此適宜用來處理較大除錯能力的里德所羅門解碼架構。

根據 2.4 節方程式（2.13)，可以把其表示成此種迴圈形式：

$$S_j = -\sum_{i=1}^{v} \Lambda_i S_{j-i} = -(\Lambda_v S_{j-v} + \Lambda_{v-1} S_{j-v+1} + \cdots + \Lambda_1 S_{j-1}),\quad \text{for } j = v+1, v+2, \ldots, 2t \qquad (式 2\text{-}17)$$

這個迴圈形式在實際硬體上，可以表示成線性迴授移位暫存器(Linear Feedback Shift Register)，如圖 4 所示。故原先求錯誤位置多項式之係數的問題，可以將其轉變成找出最短長度的線性迴授移位暫存器，因而使其輸出的前 2t 個值就是之前錯誤症狀多項式的係數，此時乘法器上所乘的值(tap)，就是所要求的錯誤位置多項式之係數。

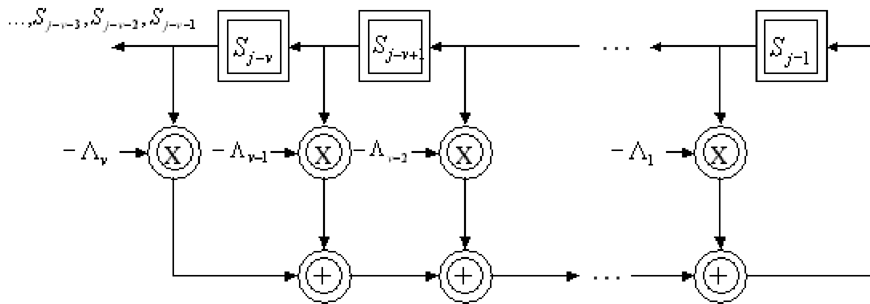


圖 4 LFSR 架構迴圈形式

整個演算法的流程是以遞迴的形式，其最主要的精神就在於讓 $\Lambda^{(k)}(x)=\Lambda_k x^k+\Lambda_{k-1}x^{k-1}+\cdots+\Lambda_1 x+1$ 成為一個最短長度為 k 的連接多項式（Connection Polynomial），且可滿足 2t 個錯誤症狀多項式的係數。剛開始先找出 $\Lambda^{(1)}$，其相對應的輸出即為第一個錯誤症狀。之後再拿第二個輸出來和第二個錯誤症狀做比較，假如兩者中間有差異值（Discrepancy），就用這個差異值來建立一個新的連接多項式；相反的，假如兩者中間沒有差異，就繼續拿此連接多項式去產生第三個輸出值，再與第三個錯誤症狀做比較。如此循環運作，直到此線性迴授移位暫存器可以產生 2t 個之前所得到的錯誤症狀。

此演算法有幾個重要的變數符號定義，以及整個演算法的歸納，將敘述如下[2]：

1. 從收到的信號求出 2t 個錯誤症狀。
2. 對演算法中的各項變數初始化：

$$k=0,\Lambda^{(0)}(x)=1,L=0,T(x)=x$$

3. 令 k = k + 1，計算差異值 $\Delta^{(k)}$：

$$\Delta^{(k)}=S_k-\sum_{i=1}^{L}\Lambda_i^{(k-1)}S_{k-i}$$

4. 假若 $\Delta^{(k)}=0$，直接跳到 8.。
5. 更新先前之連接方程式：

$$\Lambda^{(k)}(x)=\Lambda^{(k-1)}(x)-\Delta^{(k)}T(x)$$

6. 假若 $2L\geq k$，直接跳到 8.。
7. 令 L= k – L 且 $T(x)=\Lambda^{(k-1)}(x)/\Delta^{(k)}$。
8. 令 $T(x)=x\cdot T(x)$。
9. 假若 k < 2t，直接跳到 3.。
10. 找出 $\Lambda(x)=\Lambda^{(2t)}(x)$ 的根。

- 連接多項式（Connection Polynomial）$\Lambda^{(k)}(x)$
- 更正多項式（Correction Polynomial）T(x)
- 差異值（Discrepancy）$\Delta^{(k)}$
- 線性迴授移位暫存器長度 L

由上述步驟，可以清楚的了解里德所羅門解碼器是利用錯誤症狀多項式 $S(x)=S_0+S_1x+\cdots+S_{2t-1}x^{2t-1}$ 去計算錯誤位置。根據已知的錯誤位置多項式，可以進一步計算出錯誤大小評估多項式（Error-Evaluator Polynomial）而求得該錯誤位置相對應之錯誤大小。定義一組錯誤位置多項式 $\Lambda(x)=\Lambda_0+\Lambda_1x+\cdots+\Lambda_tx^t$，則錯誤大小評估多項式可得知定義如下：

$$\omega(x)=\omega_0+\omega_1x+\cdots+\omega_{t-1}x^{t-1} \qquad \text{（式 2-18）}$$

$$\Lambda(x)S(x)\equiv\omega(x)\bmod x^{2t} \qquad \text{（式 2.19）}$$

方程式（2.19）稱為關鍵方程式（key equation）。後面的章節裡將會提到的佛尼（Forney）演算法，就是利用錯誤大小評估多項式以及找到的錯誤位置去求出相關位置之錯誤大小。

2-4.2.2 最高公因式演算法(Euclidean Algorithm)

這個演算法主要是應用到數學裡求最高公因式的輾轉相除法，只是額外作一些改變而已，其方法比較淺顯易懂，但過程較為複雜，運算量也較 Berlekamp 的方法大。

假設有兩個多項式 $a(x)$ 和 $b(x)$，其最高公因式（GCD）為 $r_n(x)$，則必定存在兩組多項式 $u(x)$ 和 $v(x)$，符合 $r_n(x)=u(x)a(x)+v(x)b(x)$。一般所看到的輾轉相除法必定以下面的形式呈現：

$$r_1(x)=a(x)-q_1(x)\cdot b(x)$$

$$r_2(x)=b(x)-q_2(x)\cdot r_1(x)$$

$$r_3(x) = r_1(x) - q_3(x) \cdot r_2(x) \quad （式 2-20）$$

$$\vdots$$

$$0 = r_{n-1}(x) - q_{n+1}(x) \cdot r_n(x)$$

值得注意的是此遞迴方法不僅找出$r_n(x)$，同時也會找到$u(x)$及$v(x)$。

根據上述演算法的流程，可以定義一些變數，並延續這些定義，將演算法歸納出一個遞迴的形式：

$$\begin{aligned} & r_{-1}(x) = a(x),\ \ r_0 = b(x) \\ & q_{k+1}(x) = \left\lfloor r_{k-1}(x) \middle/ r_k(x) \right\rfloor \\ & r_k(x) = u_k(x)a(x) + v_k(x)b(x) \\ & u_{-1}(x) = 1,\ \ u_0(x) = 0 \\ & v_{-1}(x) = 0,\ \ v_0(x) = 1 \end{aligned} \quad （式 2-21）$$

$$\begin{aligned} r_{k+1}(x) &= u_{k+1}(x)a(x) + v_{k+1}(x)b(x) \\ &= r_{k-1}(x) - q_{k+1}(x)r_k(x) \\ &= (u_{k-1}(x)a(x) + v_{k-1}(x)b(x)) \\ &\quad - q_{k+1}(x)(u_k(x)a(x) + v_k(x)b(x)) \\ &= (u_{k-1}(x) - q_{k+1}(x)u_k(x))a(x) \\ &\quad + (v_{k-1}(x) - q_{k+1}(x)v_k(x))b(x) \end{aligned} \quad （式 2-22）$$

因此，藉由這些定義及推導，可以統整出一些關係式：

$$r_{k+1}(x) = r_{k-1}(x) - q_{k+1}(x)r_k(x) \quad （式 2-23）$$

$$u_{k+1}(x) = u_{k-1}(x) - q_{k+1}(x)u_k(x) \quad （式 2-24）$$

$$v_{k+1}(x) = v_{k-1}(x) - q_{k+1}(x)v_k(x) \quad （式 2-25）$$

$$q_{k+1}(x) = \left\lfloor r_{k-1}(x) \middle/ r_k(x) \right\rfloor \quad （式 2-26）$$

只要一直重複計算（2.23）（2.24）（2.25）和（2.26）四式，直到$r_{n+1}(x) = 0$，就可以找到最大公因式 $r_n(x) = u_n(x)a(x) + v_n(x)b(x)$，以及相對應的$u(x)$及$v(x)$兩個多項式。

Euclidean 演算法就是要用這種精神去計算前面所提到的關鍵方程式（Key Equation）：$\omega(x)=\Lambda(x)S(x) \bmod x^{2t}$，將這個關鍵方程式轉變為另一種類似最高公因式的描述方式：$\omega(x)=\Lambda(x)\cdot S(x)+u(x)\cdot x^{2t}$，由此可見，只要擁有錯誤症狀多項式$S(x)$，接下來就可以採用 Euclidean 演算法來求出最高公因式$\omega(x)$，也就是錯誤大小評估多項式，而且還可以一併得到錯誤位置多項式$\Lambda(x)$。

最後針對用於里德所羅門解碼器的 Euclidean 演算法作一個總結，令$x^{2t}\equiv a(x)$且$S(x)\equiv b(x)$，則$\omega(x)\equiv r_k(x)=GCD(a(x),b(x))$，以及 $\Lambda(x)\equiv v(x)$，終止的條件為$\deg(r_k(x))<t$ [1]，完整演算法的歸納如下。

1. 從收到的信號求出 2t 個錯誤症狀。
2. 對變數做初始化：

$$\Lambda_{-1}(x)=0,\ \ r_{-1}(x)=x^{2t}$$

$$\Lambda_0(x)=1,\ \ r_0(x)=S(x)$$

3. 遞迴方式從 i = 1 到$\deg(r_i(x))<t$：

$$\Lambda_i(x)=\Lambda_{i-2}(x)-Q_{i-1}(x)\Lambda_{i-1}(x)$$

$$r_i(x)=r_{i-2}(x)-Q_{i-1}(x)r_{i-1}(x)$$

$$Q_{i-1}(x)=\left\lfloor r_{i-2}(x)/r_{i-1}(x)\right\rfloor$$

2-4.3 尋找錯誤位置之方法(Chien Search)

Chien 尋根演算法說穿了其實很容易，可以說是很系統化的一種方法，也可以形容為土法煉鋼法，其精神就是把建在的伽羅瓦場中所有的元素代入錯誤位置多項式測試，假如有一個元素使得$\Lambda(\alpha^{-j})=0$，代表α^j即為錯誤的位置，也就是接收信號中的r_j發生錯誤，找到錯誤的位置之後，就可以利用下一小節所要介紹的佛尼（Forney）演算法去找出其相關位置之錯誤大小，然後加以修正。

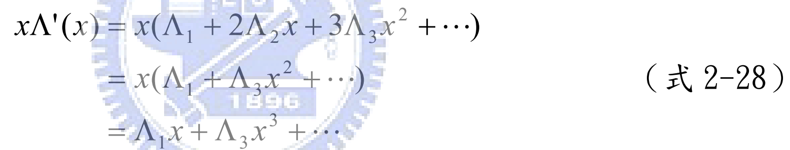
2-4.4 尋找錯誤大小的佛尼(Forney)演算法

此演算法主要也是從錯誤症狀（Syndrome）和關鍵方程式（Key Equation）推導得來[2]：

$$Y_i = \frac{X_i^{-(m_0-1)}\omega(X_i^{-1})}{X_i \prod_{j=1, j\neq i}^{v}(1 - X_j X_i^{-1})} = -\frac{X_i^{-(m_0-1)}\omega(X_i^{-1})}{\Lambda'(X_i^{-1})} = -\frac{x^{m_0}\omega(x)}{x\Lambda'(x)}\bigg|_{x=X_i^{-1}} \quad （式 2-27）$$

其中 Y_i 是錯誤大小、X_i 是錯誤位置，$\Lambda'(x)$ 是錯誤位置多項式 $\Lambda(x)$ 的微分，所以整個演算法要配合著 Chien 尋根演算法所找出的錯誤位置後，再代入求出錯誤大小。

這中間有一個比較值得注意的地方是，伽羅瓦場裡的微分，會使多項式裡的二次項等於零，因為伽羅瓦場裡的加法相當於 XOR 的動作，才會產生這種效果。



$$\begin{aligned} x\Lambda'(x) &= x(\Lambda_1 + 2\Lambda_2 x + 3\Lambda_3 x^2 + \cdots) \\ &= x(\Lambda_1 + \Lambda_3 x^2 + \cdots) \\ &= \Lambda_1 x + \Lambda_3 x^3 + \cdots \end{aligned} \quad （式 2-28）$$

所以 $x\Lambda'(x)$ 相當於 $\Lambda(x)$ 奇數次方項的總和。

第三章　摺疊演算法之推導與設計及 Xilinx EDK 整合系統環境介紹

3-1 摺疊演算法的硬體架構介紹

現今許多的硬體架構設計為了達到簡單化和規律性，大多採用運算單元(PE)陣列的形式，Systolic 陣列是其中最著名並且被廣泛運用的。Systolic 陣列是由 Kung[3]在 1980 年所提出來的一種硬體架構，這種架構是由一些簡單的基本運算單元所連接組成，所以具有簡單且規則的特性，適合用在超大型積體電路的設計上。此外，Systolic 陣列用到了大量的管線化(Pipelining)以及多重運作(Multiprocessing)，因此，它可以達到高速的運算效能以及維持很高的處理速率(Throughput)。

在 1995 年 Keiichi Iwamura[4]提出一篇利用 Systolic 陣列架構為基礎的里德所羅門解碼器硬體，其主要的特色就是只使用一種相同基本運算單元來實現解碼器架構，並且利用了管線化的技巧，不僅讓整個硬體非常的簡單且有規律，也可以有很高的處理速率。

從另外一個角度來看，假如在整個系統中，在里德所羅門解碼區塊的下一級並不需要很高的處理速率，那麼這個高處理速率的優點則可以拿來交換一些其他的好處。摺疊硬體架構[14]就是一個以降低處理速率而換取減少硬體面積的一個方法。這個方法是將一連串基本運算單元的陣列經由摺疊方法，使用較少的運算單元，分時完成原本的所需的運算量，雖然整體的運算時間變長，但運算單元卻可以減少。這樣對於不必要很高處理速率的系統，不僅速度能夠達到要求，還可以減少硬體面積，不失為一個好方法，而其最大的好處除了以時間上的優勢取得空間上的改善外，摺疊後的架構也有可重複使用性，增加了硬體的彈性。

一般傳統運算單元陣列，對於訊號傳遞的路徑，可分為兩種：向前路徑(Forward Path)以及回授路徑(Feedback Path)。向前路徑上的訊號，在兩個相鄰的運算單元之間互有關係，而回授路徑上的訊號，則只有在本體運算單元中有影響。在陣列中的每個基本運算單元，主要區分成兩部份：乘加器(MAC)與暫存器(Register)，圖 5 即為一個常見的基本運算單元。為了在演算法上的推導方便，在這裡將乘加器和暫存器分開表示，而在往後運算單元的表示上，也皆以此方式呈現。

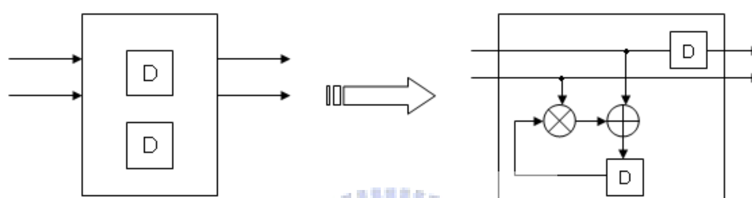


圖 5 基本運算單元(Processing Element)

現在就以一個例子來說明摺疊演算法的推導。假設有一個串接 12 個基本運算單元的陣列，首先要決定所需要使用的運算單元個數，至於要採用幾個基本運算單元，則取決於系統的需求。這裡我們採用三個基本運算單元來做說明，因此可以將 12 個基本運算單元分成四列，每一列的基本運算單元必須完整的排列對齊，這樣的目的是打算將原本在一個回合內就可以完成的資料計算，分成三個時間點來運算，每一列為進行一個時間點。圖 6 所示為基本運算單元的排列方式。

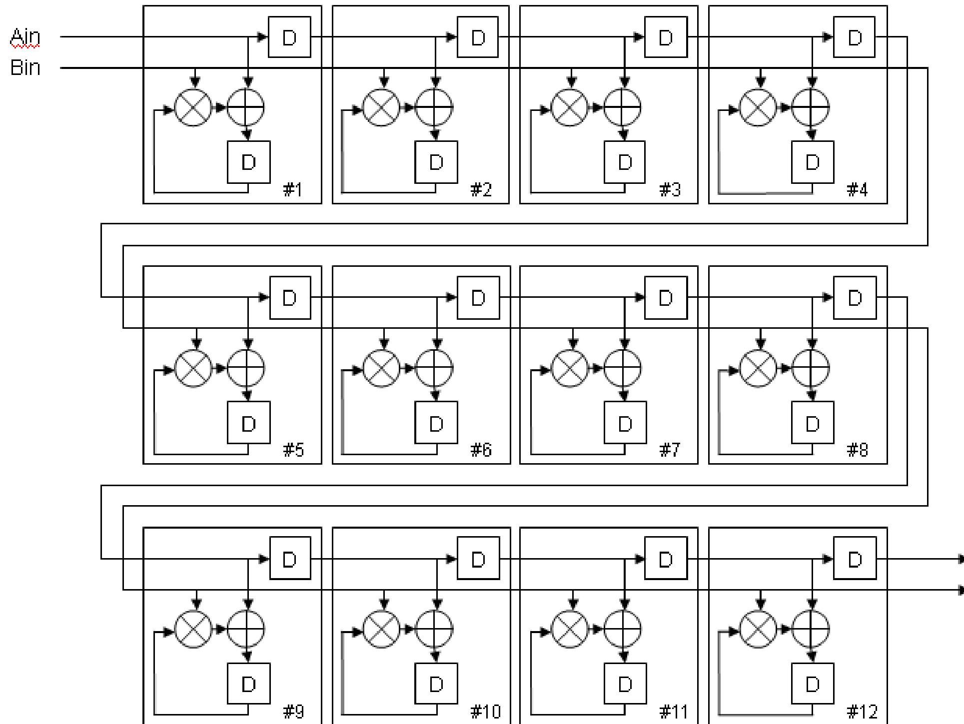


圖 6 基本運算單元陣列

決定好需要幾個基本運算單元後，我們開始對整體的架構做一些改變。首先將每個基本運算器裡的暫存器變成原本的三倍大小，接著在列與列之間的每個向前路徑上加入資料銜接的暫存器，也就是每個時間點最後運算的值暫存起來以傳遞給下個一時間點處理，如此一來，在每回合的第一個時間點時，Ain 與 Bin 送入資料，僅第一列的四個乘加器開始動作，且將計算後的結果存入該列各暫存器的第一個位置，在此時，第四個向前路徑上暫存器的值，也會傳遞至第一列與第二列之間的銜接暫存器中，而第二、三列的基本運算單元則沒有動作；到了第二個時間點，銜接暫存器的值開始送入第二列的乘加器，第二列的基本運算單元開始運作，且將計算後的結果存入該列各暫存器的第二個位置，同時第四個向前路徑上暫存器的值傳遞至第二列與第三列之間的銜接暫存器中，其餘二列的基本運算單元均不動作；到了第三個時間點，銜接暫存器的值開始送入第三列的乘加器，第三列的基本運算單元則開始運作，並將計算後的結果存入該列各暫存器的第三個位置，其他二列的基本運算單元也均不動作；到此一個回合的運算就結

束，之後進入下一個新的回合，如此一直反覆循環。圖 7 顯示各暫存器的變動以及銜接暫存器連接的位置。

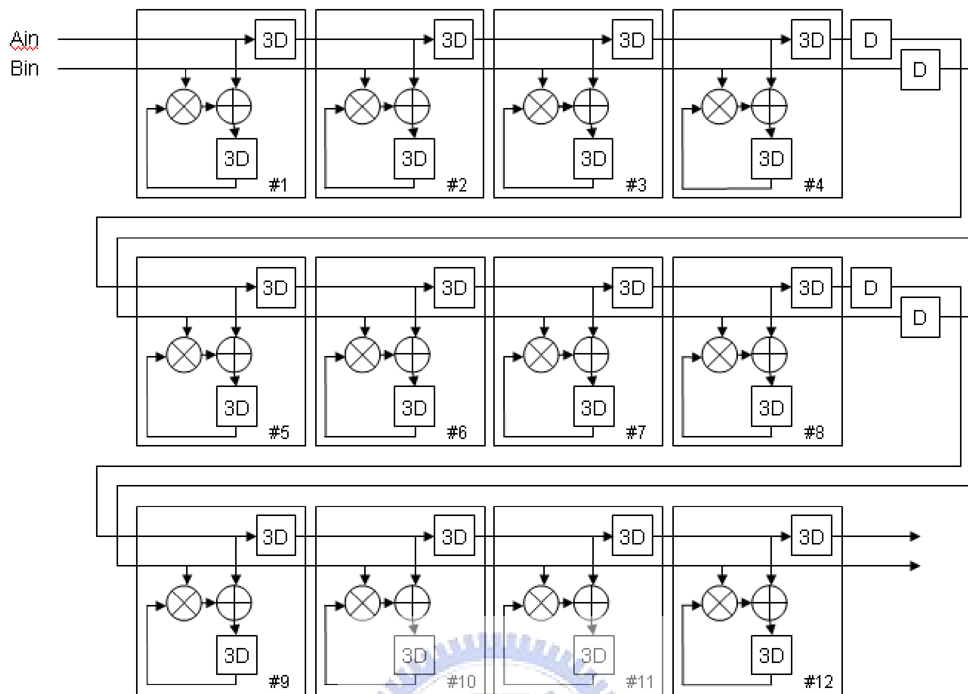


圖 7 基本運算單元陣列之時序規劃

由此可以知道，每一列中的第一個乘加器，在每回合的三個動作時間點是不重疊的，也就是每一列的第一個乘加器不會在同一個時間點同時動作，而在每一列的第二、三、四個乘加器也是如此。由於每一列中相對位置的乘加器在不同工作時間點是互相不衝突的，所以可以將三列的基本運算單元摺疊成一列，如圖 8 所顯示。於是，只要在一個新的回合中，送入 Ain 和 Bin 資料，並且在這一個回合裡，依照不同的時間點，調整每個乘加器各自所需輸入的來源暫存器，以及將每個乘加器輸出所需儲存的計算結果存入正確的暫存器位置，就可以得到正確的運算，和未摺疊前的硬體架構所得到的運算結果是一樣的。經過摺疊後的硬體架構雖然把整體的運算時間拉長，但是卻減少了三分之二的乘加器數目，這也達到了我們摺疊所期望的效果。

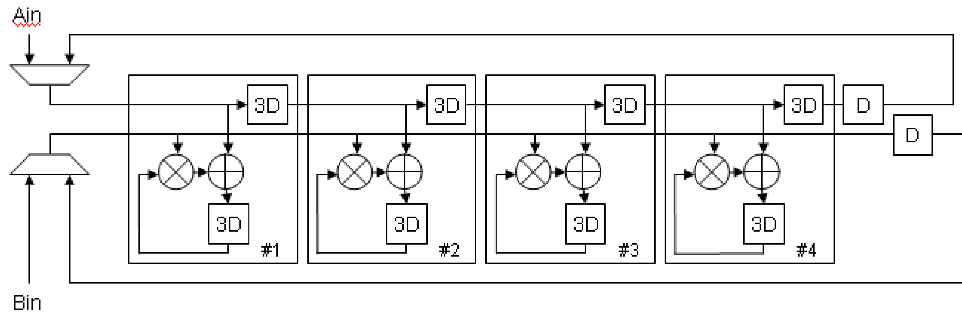


圖 8 基本運算單元陣列之摺疊架構

原先 12 個基本運算單元的陣列，經過摺疊演算法的調整後，可以清楚的分出乘加器群(MAC Group)及暫存器檔案(Register File)，要使整個運算流程正確的動作，需要在每個時間點調整暫存器檔案所需要輸出及儲存的資料，以及正確的切換在最前端的多工器(MUX)，使其在 Addr=00 時，也就是新的一回合開始的時候，由外部 Ain 及 Bin 當輸入，其餘時間點則由銜接暫存器當輸入，就可以完成我們要求達到的工作，至於乘加器群只要不停的運算接收到的資料就可以了。圖 9 表示經過摺疊演算法所產生的架構圖。

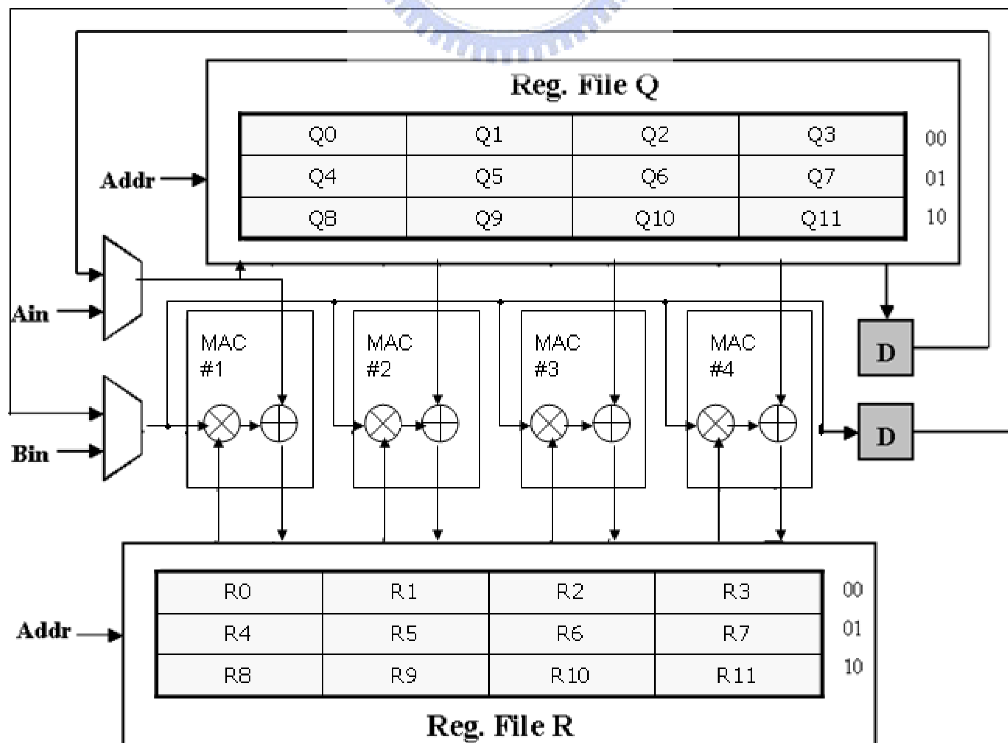


圖 9 摺疊架構之乘加器與暫存器規劃

從圖 9 的摺疊架構可以知道，在暫存器檔案中的每一行(Column)共有三個暫存器，也就是先前推導過程中三倍暫存器(3D)大小所代表的意思。另外，暫存器檔案每一列(Row)中也有四個暫存器，同一列的四個暫存器是需要在同一個時間點送資料到乘加器群做運算，因此將每一列的四個暫存器設定為一組，給予同一個暫存器定址名稱，到時只需要正確的控制暫存器位址，整個架構就會正常運作，圖 10 即為暫存器組合及定址的示意圖。

摺疊演算法最只要的精神就是在於時脈上的規劃，將一個回合的動作分時完成，而依此演算法所推演出的硬體架構，擁有可重複使用的特性。在下一章節中會更詳細的說明套用摺疊演算法時基本運算單元個數的選擇，以及摺疊硬體架構的好處。

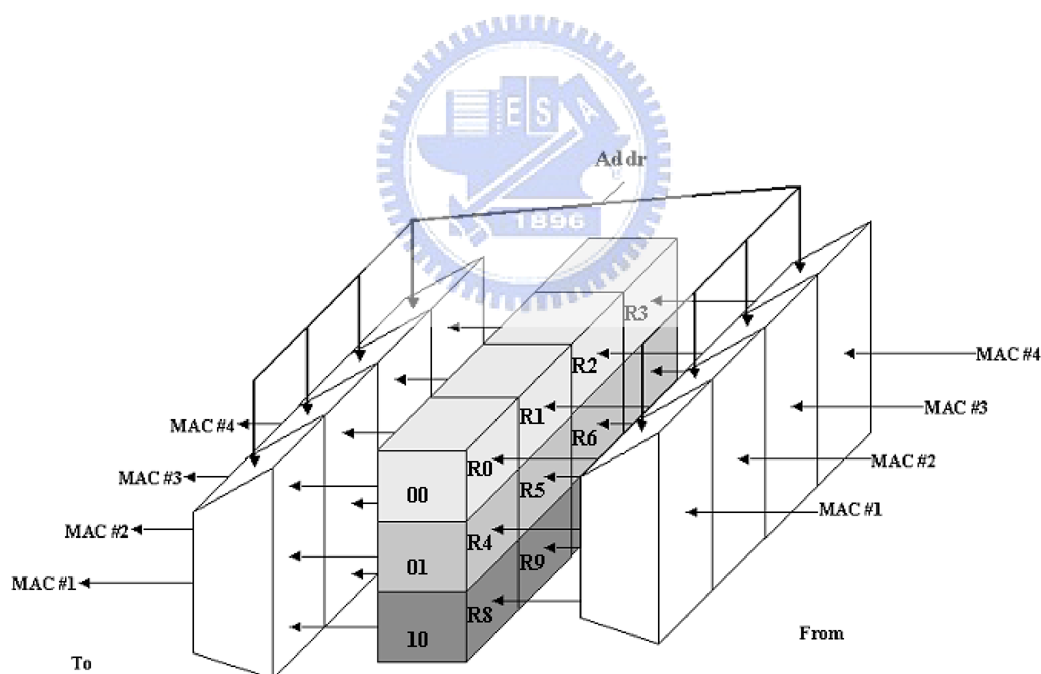


圖 10 摺疊架構之暫存器定址

3-2 摺疊演算法實現硬體之方法與特色

在上一節中有提到，在使用摺疊演算法之前，首要的工作就是決定要使用的乘加器個數及暫存器檔案的大小。至於如何決定乘加器個數及暫存器檔案大小，

可以依據系統的需求然後按照以下的數學式子簡單的得到，接著就來說明這些數學式子如何產生。假設現在系統中有一個方塊，其硬體架構由N個基本運算單元所組成，且延遲時間(Latency)為 t_{mac} ，而這個方塊的規格所需的處理速率(Throughput)為 X_{put} ，於是重複週期(Iteration Period)IP 及所採用的乘加器數目m各為：

$$IP = X_{put}^{-1} / t_{mac} \quad \text{(式 3-1)}$$

$$m = \lceil N/IP \rceil \quad \text{(式 3-2)}$$

藉由上列的式子計算，可以估計出新摺疊架構採用的乘加器數目m，且將原本一回合中的計算，分為 $\lfloor IP \rfloor$，個時間點來達成，而摺疊後的架構不僅符合規格，也比原先乘加器的數目少。

摺疊演算法非常適合於處理速率不需要很高的陣列硬體，尤其是對於陣列的基本運算單元個數會因規格不同而有數目上變化的架構。以里德所羅門解碼器架構為例，解碼器方塊的內部大多數的構成部分是由基本運算單元陣列所組成，而且陣列的長度是根據不同的解碼器更正能力規格來改變，更正能力越大，基本運算單元個數需要越多。當對於這種硬體架構來實行摺疊時，則以最差狀況的處理速率及陣列長度來估計所採用的基本運算單元個數，摺疊後的架構對於更正能力較小的規格，只需要減短重複週期，調整每個時刻所指定的輸出及輸入暫存器檔案即可，這樣的硬體架構展現了硬體可重複使用的特性。此外，摺疊後的硬體架構在處理速率規格符合的前提下，若要提升最大更正能力，加長陣列長度，只需要擴充暫存器檔案大小，增加重複週期，如此也使得摺疊後的硬體更具有彈性。

3-3 Xilinx EDK 整合系統背景

隨著科技進步，嵌入式系統廣泛的被應用在各類的產品中，嵌入式系統的研究也不再只是純軟體或純硬體的設計，而以軟、硬體整合的趨勢為主。早期的系統都是先做好硬體的部份，下線實作成晶片後，才交由軟體設計者做軟體的設計，而軟體設計者若是在設計過程中，硬體部分有問題，則需要跟硬體設計者做

溝通，這樣才能設計出一套系統。由於軟體設計者必須等到硬體設計者完全設計好硬體才能開始工作，並且在軟體設計者與硬體設計者之間的溝通又會浪費許多時間，導致產品生產速度減慢。所以就有人提出軟、硬體共同設計的作法，加快產品的生產速度。

基於軟硬體共同設計的想法 Xilinx EDK 整合系統，提供一個讓使用者快速開發嵌入式系統的介面。EDK 整合系統提供了設計者硬體設計元件，其中包括了 MicroBlaze 處理器和 PowerPC 處理器[5][6]，和可程式化晶片(Field Programmable Gate Arrays，FPGA)，以及記憶體和一些外部 I/O 周邊設備。由於 FPGA 可重複程式化的優點，方便用來當做測試驗證平台。另外此系統還提供了軟體設計的編譯器(compiler)，提供了部份標準 C 的支援，以及 Xilinx 特殊的嵌入式 C[7]的支援，可以讓使用者快速的自我設計一套軟體驅動硬體。

3-4 MicroBlaze 架構

MicroBlaze 處理器是屬於標準的 32 位元哈佛精簡指令集處理器(Harvard Reduced Instruction Set Computer, Harvard RISC)的架構。因此特性為：將資料與指令匯流排分開；可以再一個時序週期內同時對資料及指令作存取；執行一道指令只需要一個時序即可完成。而此處理器所有的指令和資料都是以位元(bit)為處理的最小單元，寬度為 32 位元，也支援 word、half word、byte access。

MicroBlaze 處理器擁有高效能的可規劃性，可依照使用者設計需求而提供了許多特色，包含如下：

1. 擁有 32 個 32 位元的一般用途暫存器(general purpose register)以及 5 個 32 位元的特殊用途的暫存器(special purpose register)。
2. 提供了 32 位元的指令集，擁有三個運算元和兩種位址模式。
3. 分別提供資料流和指令流技術，彼此之間是各自獨立作業的。
4. 擁有 32 位元的位址線

5. 提供重置(reset)、中斷(interrupts)、例外處理(expections)等服務

6. 管線化

MicroBlaze 處理器包括了運算單元、浮點數運算單元、暫存器、快速緩衝儲存體(cache)、以及匯流排(bus)，如圖 11 為 MicroBlaze 處理器架構圖。

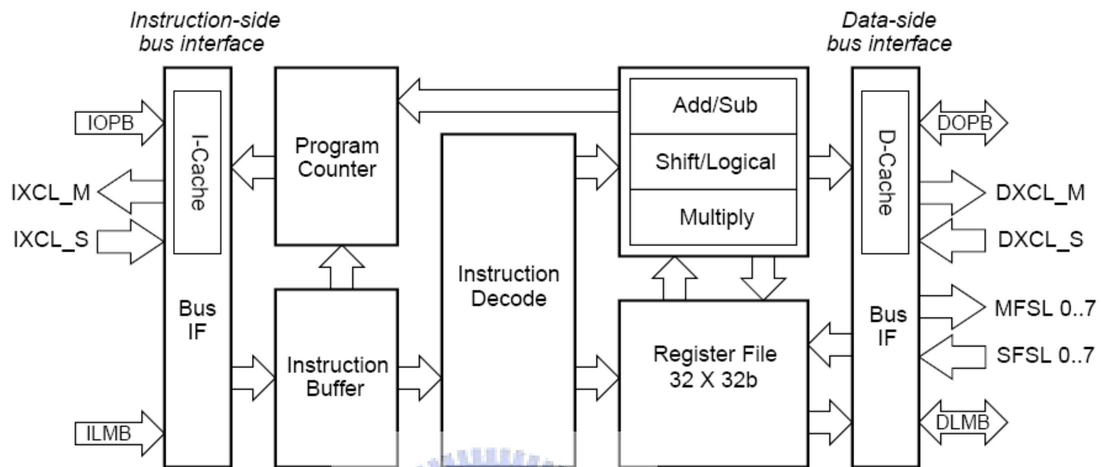


圖 11 MicroBlaze 處理器架構圖

在傳輸方面，MicroBlaze 提供了三種匯流排，用於 MicroBlaze 和硬體電路的溝通及資料的傳遞，分別為 On-chip Peripheral Bus(OPB)、Fast Simplex Link(FSL)和 Processor Local Bus(PLB)。在此我們只介紹 OPB 的協定。

3-5 On-chip peripheral Bus(OPB)

On-chip peripheral Bus[8]是 Xilinx 嵌入式系統中最主要的一條匯流排，提供了外部 I/O 周邊設備、中斷控制、DMA、以及處理器和使用者設計硬體電路的溝通。目前 OPB 是採用 IBM 的匯流排標準 CoreConnect 架構，且利用 Xilinx 的 CoreGen 提供了很方便的設計流程來驅動 OPB，所以使用者必須符合其匯流排協定，即可以輕易的連接在晶片裡的硬體。圖 12 為 MicroBlaze 系統的 OPB 示意圖。此匯流排擁有幾項特色：

1. 完全的同步單一時脈

2. 提供 32 位元位址線和 32 位元資料線

3. 一個週期時間就可以完成在 OPB master 與 OPB slave 之間的資料傳送

4. 擁有仲裁機制，用來控制處理器與周邊硬體電路的溝通

5. 提供 Master byte enables 和 Slave 的 timeout supress、request retry

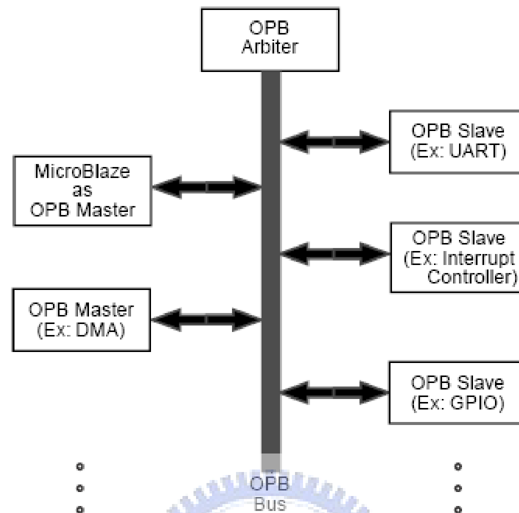


圖 12 MicroBlaze 系統的 OPB 仲裁器示意圖

在 OPB 仲裁器中，外部 I/O 周邊設備皆為 Slave，唯讀 DMA 可以是 Master，所以訊號線有分 Master 和 Slave 的不同，下面是 OPB 介面訊號線的描述，表 3.1 為 OPB 介面訊號一覽表：

OPB_Clk：由外界加入 OPB 的工作時脈。

OPB_Rst：由 OPB 送出去的重置 Reset 訊號。

OPB_ABus：由 OPB 送出的位址匯流排，來選擇 OPB 匯流排上的裝置，寬度為 0~31。

OPB_DBus：由 OPB 送出的資料匯流排，寬度為 0~31。

OPB_BE：由 OPB 送出的位元組致能訊號，用來控制 OPB 傳輸是以 byte 來傳輸。

OPB_RNW：由 OPB 送出的讀、寫控制訊號，用來控制 OPB Master 和 OPB Slave 之間的關係，當此訊號為高位準時，即是 OPB Master 可以讀取 OPB Slave 資料而不能寫入，當訊號為低位準時則相反。

OPB_select：由 OPB 送出的選擇訊號，表示選到要的裝置。

OPB_seqAddr：由 OPB 送出的順序位址訊號。

表 2　OPB 介面訊號一覽表

訊號名稱	寬度(位元)	方向
OPB_Clk	1	輸入
OPB_Rst	1	輸入
OPB_ABus	32	輸入
OPB_DBus	32	輸入
OPB_BE	5	輸入
OPB_RNW	1	輸入
OPB_select	1	輸入
OPB_seqAddr	1	輸入
<Sln>_DBus	32	輸出
<Sln>_xferAck	1	輸出
<Sln>_retry	1	輸出

由 OPB Slave 裝置送到 OPB 仲裁器的訊號如下，訊號前面的<Sln>為硬體電路名稱：

<Sln>_DBus：讀取資料的資料線，寬度為 0~31。

<Sln>_xferAck：Slave 裝置的轉換認可訊號，當高位準時代表傳送或是接收的動作已經完成，當為低位準時則相反。

<Sln>_retry：Slave 裝置的重試訊號。

有了上述的 OPB 訊號線，接著我們來說明 OPB 是如何做讀取和寫入的動作。當要讀取時，OPB Master 會拉起 OPB_select 訊號，並且選擇有效的位址線 OPB_ABus，且 OPB_RNW、OPB_BE 均拉起來維持穩定狀態，Slave 裝置將資料準備好要傳送並且拉起 OPB_xferAck 告訴 Master 可以從匯流排取資料，然後將 OPB_select 訊號拉下來。圖 13 為 OPB 讀取時序圖。

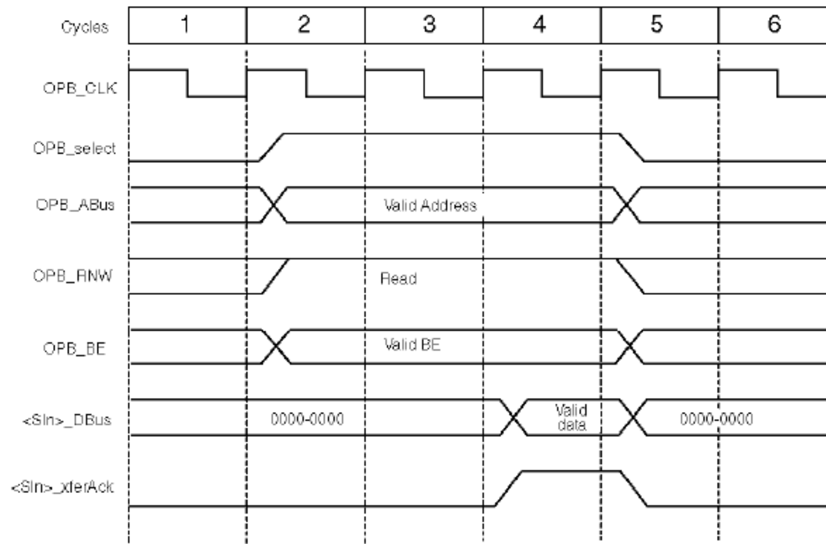


圖 13 OPB 讀取時序圖

當要寫入時，OPB_Select 會被拉起來，並且選擇有效的位址線 OPB_ABus，且 OPB_BE 均會拉起來維持穩定狀態，而 OPB_RNW 在低位準才可以將資料寫入，直到資料寫入完畢 OPB_xferAck 會發送一訊號告知。圖 14 為 OPB 寫入時序圖。

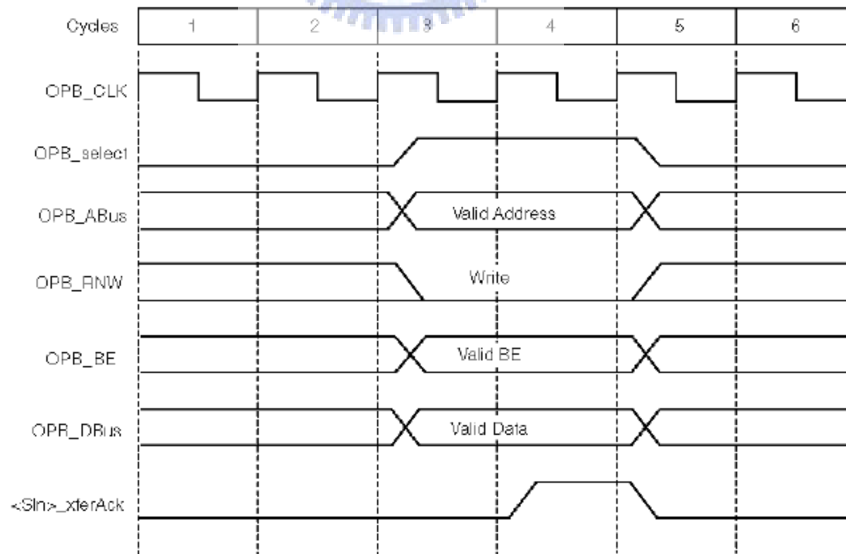


圖 14 OPB 寫入時序圖

第四章　硬體實現架構

在介紹這麼多里德所羅門相關演算法以及摺疊理論推導之後，再來就是要考慮里德所羅門解碼硬體實現的部分。有些在演算法上看似簡單的問題，在硬體實現上不見得相對單純，在接下來的內容之中，會作較為詳盡的探討。主要焦點會集中在應用摺疊演算法所架構的硬體部分，以及將尋找錯誤症狀和尋找錯誤位置的硬體架構合併。

首先，第一節中會概括的提到整個里德所羅門解碼的完整電路架構。第二節會提出一種低功率消耗及低延遲的伽羅瓦場乘法器。接下來的幾個小節會分別介紹尋找錯誤症狀及尋找錯誤位置的摺疊硬體架構，以及如何將這兩塊硬體架構合併，還有軟、硬體之間如何溝通，都會做仔細的介紹。最後就是將模擬的結果呈現出來。

4-1　整合里德所羅門解碼器與完整電路架構

由第二章的里德所羅門解碼基本原理中，可以知道整個解碼過程是經由：

（一）尋找錯誤症狀

（二）尋找錯誤位置多項式及錯誤大小評估多項式

（三）尋找錯誤位置

（四）尋找錯誤大小

由於(二)跟(四)的部分由軟體處理器來完成，因為使用的是不同演算法，因此還是分為兩個方塊，而(一)跟(三)我們會做在同一套硬體裡面，因此整個系統定義成五個區塊：

（一）尋找錯誤症狀及錯誤位置方塊(Syndrome_Chine Search Block)

（二）尋找錯誤位置多項式及錯誤大小評估多項式方塊(BM Block)

（三）尋找錯誤大小方塊(Forney Block)

（四）控制暫存器(Control register)

（五）狀態暫存器(Status register)

收到的信號先經由尋找錯誤症狀及錯誤位置方塊找到錯誤症狀，接著再將錯誤症狀送入尋找錯誤位置多項式及錯誤大小評估多項式方塊，找出相對應的錯誤位置多項式以及錯誤大小評估多項式。接著再送入尋找錯誤症狀及錯誤位置方塊找錯誤位置，最後將錯誤位置送回尋找錯誤大小方塊即可找到錯誤的值。整體電路概略圖如圖 15 所示。

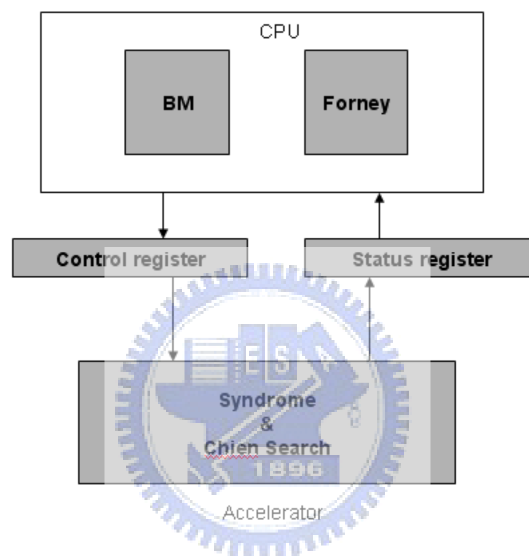


圖 15 里德所羅門解碼器軟、硬體概略圖

4-2 伽羅瓦場乘法器

在許多的參考文獻之中，提出了很多不同架構的伽羅瓦場乘法器，而由 K. K. Parhi[9]在 2001 年所提出的一篇專門為了里德所羅門碼而設計的有限場乘法器具有低功率且低延遲時間(Latency)的特性，因此被廣為使用。以下就來介紹此有限場乘法器的架構。

假設現有二多項式 $A, B \in GF(2^m)$，則 A 與 B 可以表示為基底形式：

$$A = \sum_{k=0}^{m-1} a_k \alpha^k,\ B = \sum_{k=0}^{m-1} b_k \alpha^k \qquad \text{（式 4-1）}$$

若 C 為 A 與 B 之乘積，則：

$$C = A \cdot B = \sum_{k=m}^{2m-2} d_k \alpha^k + \sum_{k=0}^{m-1} d_k \alpha^k \quad where \quad d_k = \sum_{i=0}^{k} a_i b_{k-i} \qquad \text{（式 4-2）}$$

其中α^k是 GF(2^m)中的元素，且一定含有獨特的$g_i^{(k)}$使得：

$$\alpha^k = \sum_{i=0}^{m-1} g_i^{(k)} \alpha^i$$

$$where \quad 0 \le i \le m-1 \quad and \quad m \le k \le 2m-2 \qquad \text{（式 4-3）}$$

這些$g_i^{(k)}$都是可以事先算好的。

因此 C 可以經由推導化簡為：

$$C = \sum_{k=m}^{2m-2} d_k \alpha^k + \sum_{k=0}^{m-1} d_k \alpha^k = \sum_{k=m}^{2m-2} d_k (\sum_{i=0}^{m-1} g_i^{(k)} \alpha^i) + \sum_{k=0}^{m-1} d_k \alpha^k$$

$$= \sum_{k=0}^{m-1} \alpha^k (d_k + \sum_{j=m}^{2m-2} d_j g_k^{(j)}) = \sum_{k=0}^{m-1} \alpha^k c_k \qquad \text{（式 4-4）}$$

$$where \quad c_k = d_k + \sum_{j=m}^{2m-2} d_j g_k^{(j)}$$

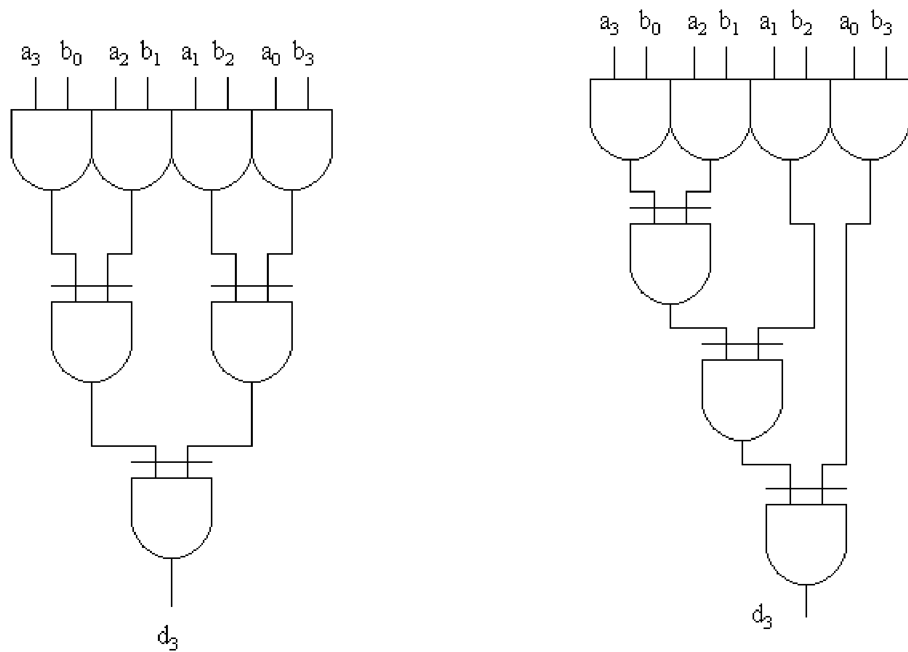
經由上面的推導我們不難發現整個乘法的過程包含了兩個步驟：

（一）兩數相乘(Multiplication)（方程式(式 4-2)得知）

（二）模數的化簡(Modular Reduction)（方程式(式 4-4)得知）

這兩個步驟之間沒有所謂資料依賴(Data Dependence)的關係，所以兩者之間可以平行處理。值得注意的是，在這兩部分之中，平衡的互斥樹狀架構(Balanced XOR Tree)可以達到最短之邏輯路徑效果。例如：

$$(a_3 \ AND \ b_0) \ XOR \ (a_2 \ AND \ b_1) \ XOR \ (a_1 \ AND \ b_2) \ XOR \ (a_0 \ AND \ b_3)$$



(a) 平衡的互斥樹狀架構　　(b) 非平衡的互斥樹狀架構

圖 16 兩種互斥樹狀架構

從圖 16 中可以看出假如使用(a)平衡的互斥樹狀架構，邏輯路徑只需要經過一個 AND 和兩個 XOR 邏輯閘的時間；相反地，假如使用(b)非平衡的互斥樹狀架構，邏輯路徑就需要經過一個 AND 和三個 XOR 邏輯閘的時間，整體時間較(a)多出了一個邏輯閘的時間，所以我們當然採用平衡的互斥樹狀架構。

圖 17 是當非規則全平行乘法器建立在 GF(2^4)時之硬體架構，我們可以看到上半部就是前面所提到的第一個步驟：兩數相乘；下半部就是第二個步驟：模數的化簡。其中 $g_i^{(k)}$ $(0 \le i \le 3, 4 \le k \le 6)$ 是事先就算好的。

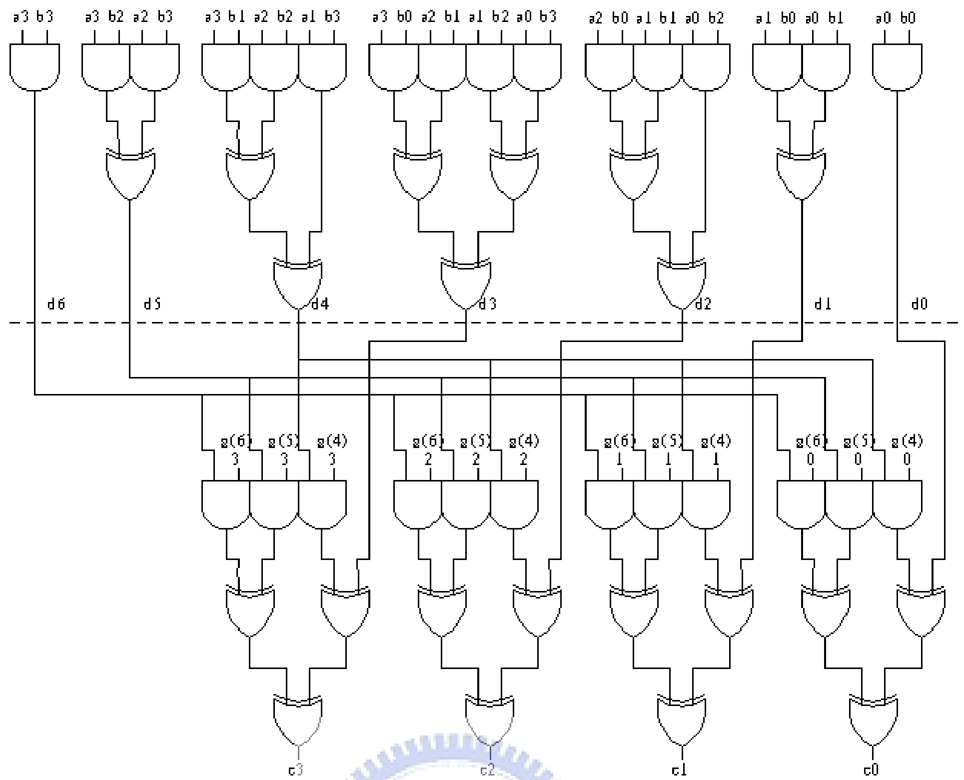


圖 17 建立在 GF(2^4)之非規則全平行乘法器硬體架構

4-3 處理錯誤症狀(Syndrome)之硬體摺疊架構

由 2-4.1 小節我們可知道錯誤症狀的計算可以被視為是一連串伽羅瓦場的乘加運算，根據定義：

$$S(x)=\sum_{j=0}^{2t-1}S_j\cdot x^j \quad where \quad S_j=\sum_{i=1}^{n}R_{n-i}\cdot(\alpha^{m_0+j})^{n-i} \qquad (式 4\text{-}5)$$

為了使得在硬體設計上有規律性，所以把 S_j 化成下面的形式：

$$S_j=(\cdots((R_{n-1}\cdot\alpha^{m_0+j}+R_{n-2})\cdot\alpha^{m_0+j}+R_{n-3})\cdots)\alpha^{m_0+j}+R_0 \qquad (式 4\text{-}6)$$

這一連串的乘累加運算，進而以演算法之 Pseudo-code 表示如下：

```
begin
  for j := 0 to 2t-1 do
    begin
      z[0, j] := 0;
      for i := 1 to n do
        z[i, j] := z[i-1, j]·α^(m0+j) + R(n-i)
    end;
  Sj := z[n, j]
end
```

由許多陣列處理器的參考書中，不難發現到只要能夠將類似上述之Pseudo-code找出來，就能夠有辦法依照變數將其硬體化成一維或二維之陣列的形式。故依照上述關於錯誤症狀之Pseudo-code，整理出其相對應之硬體架構，如圖18所示，每一個基本運算單元負責處理一個錯誤症狀的計算。

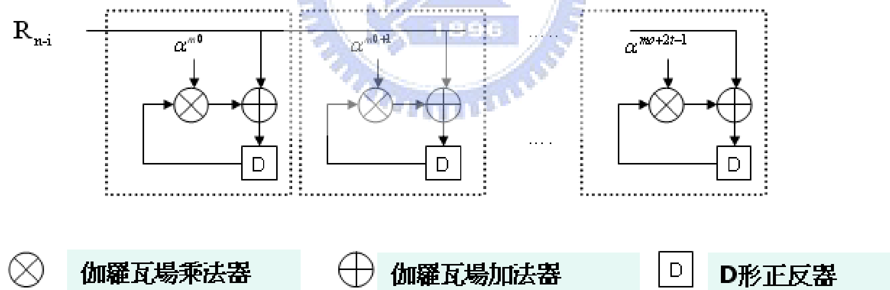


圖 18 錯誤症狀計算之陣列硬體架構

4-3.1 乘加運算器及暫存器檔案

這一節就以上述的陣列硬體架構為基礎，開始設計出錯誤症狀的硬體架構，並且利用第三章所推導的摺疊演算法，將硬體架構做摺疊處理，讓摺疊硬體架構套用在里德所羅門解碼器上。

在開始摺疊錯誤症狀的硬體架構之前，首先必須考慮要採用的基本運算單

元個數。根據錯誤症狀演算法的定義，假設更正能力為 t，則必須計算出 2t 個錯誤症狀，因此處理錯誤症狀的硬體為 2t 個基本運算單元形成的陣列。我們現在就以更正能力最大為 8 作例子，希望使用 4 個基本運算單元來架構處理錯誤症狀的硬體。由於更正能力最大為 8，所以原先由基本運算單元形成的陣列長度為 16。因為我們剛剛假設希望用 4 個基本運算單元來得到更正能力最大為 8 的錯誤症狀的值，所以先將原本 16 個基本運算單元分成 4 列，每列有 4 個基本運算單元，如圖 19 所示。

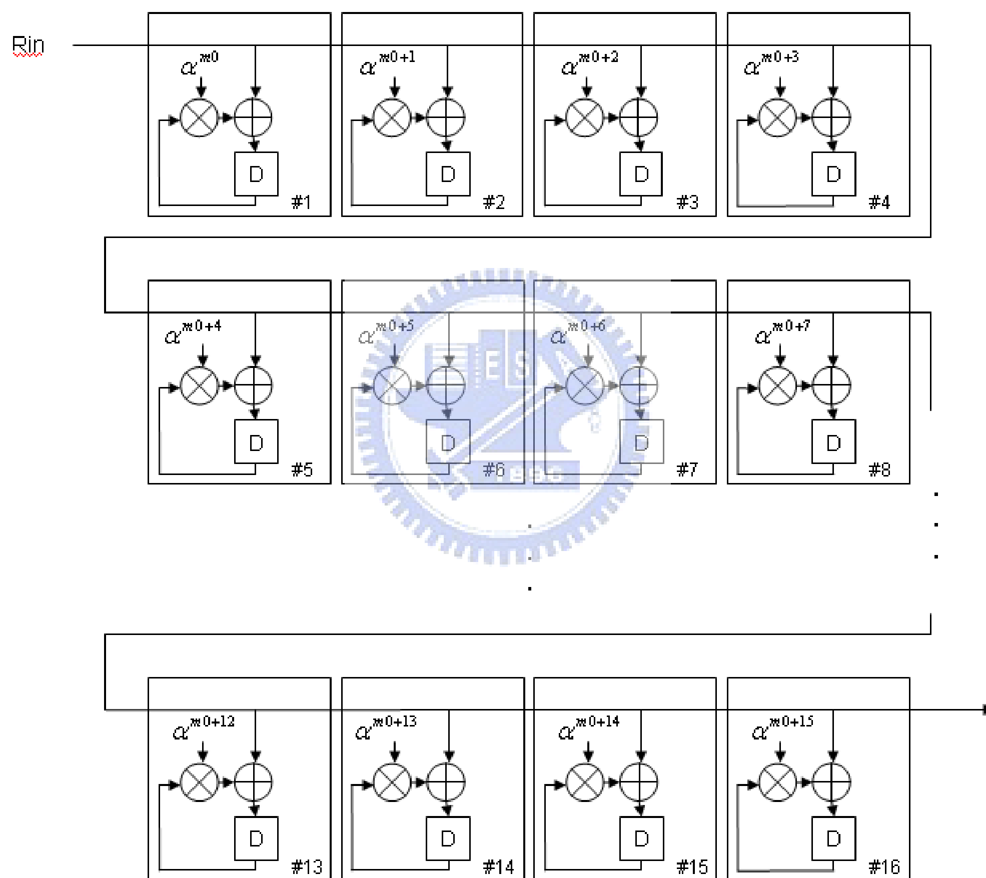


圖 19 更正能力 t=8 之錯誤症狀計算陣列

決定好要使用幾個基本運算單元後，接著要對暫存器的大小做改變，以及加入銜接暫存器使得電路正常運作。由於打算將一個回合的資料分為 4 個時間點來運算，每一列各在一個時間點完成。因此將陣列中每個暫存器變成 4 倍大小，接著每一列在向前路徑上的連接處加入一個資料銜接的暫存器。做了適當的改變

後，當在第一個時間點時，送入資料 R_{n-i}，僅第一列的 4 個乘加器開始動作，且將計算後的結果存入該列各暫存器的第一個位置中，而此筆 R_{n-i} 資料也會傳送至第一列與第二列的銜接暫存器中，而其他第二、三、四列的各個乘加器均沒有動作；到了第二個時間點，讀入銜接暫存器中的 R_{n-i} 值，第二列開始動作，並將計算結果存入該列各暫存器的第二個位置，R_{n-i} 也繼續傳送至第二列與第三列的銜接暫存器中，而其餘第一、三、四列的各個乘加器也是均沒有動作；到了第三個時間點，讀入銜接暫存器的 R_{n-i} 值，第三列開始動作，並將計算結果存入該列各暫存器的第三個位置，R_{n-i} 也繼續傳送至第三列與第四列的銜接暫存器中，而其餘第一、二、四列的各個乘加器也是均沒有動作；到了最後第四個時間點也是如此，讀入銜接暫存器中的 R_{n-i} 值，第四列開始動作，並將計算結果存入該列各暫存器的第四個位置，到此，結束了一個回合的運算。圖 20 說明了上述的這些動作。

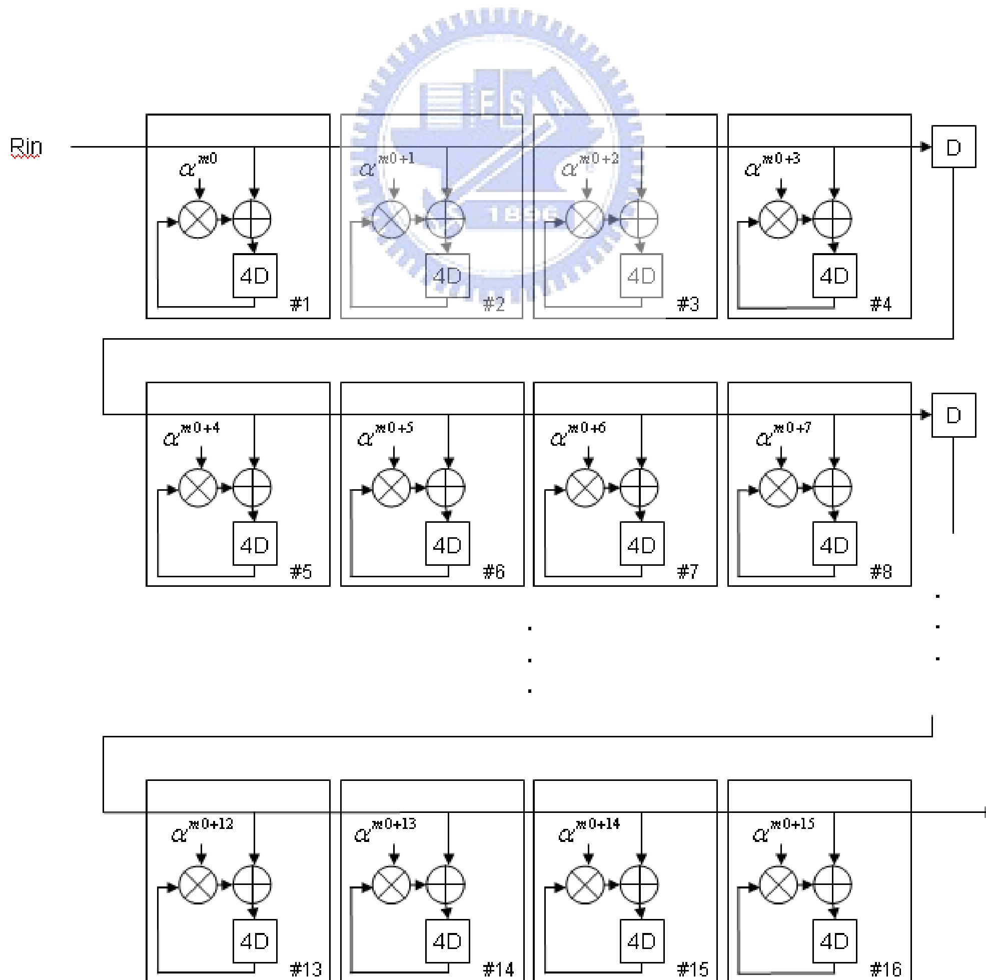


圖 20 錯誤症狀計算陣列之時序規劃

經由上述的推導我們可以清楚了解，每列中相對應位置的乘加運算器，在四個動作時間點中是兩兩互斥的，也就是列與列中相對位置的乘加器不會再同一個時間點同時有動作；而每列中所使用的暫存器，因為變成原本的 4 倍大小，所以在四個動作時間點是存入不同的位置，互相不干擾，也就是在任一時間點時不會讀取或寫入其他時間點的值；最後，銜接暫存器所存的值，都是該回合一開始輸入的值，由於這三個重要的元素在各個時間點都互不干擾，所以可以將這四列摺疊成一列。於是只要在一個新的回合中送入一個 R_{n-i} 資料，且在這一回合裡將四個時間點各自所需的計算結果存入正確的暫存器就可以得到正確的運算。因此接下來的工作就是根據不同的時間點，調整每個時間點乘加器所需的輸入來源及輸出目的地，而讓整體架構正確工作，達到和未摺疊前一樣的功能。圖 21 所示為摺疊後的硬體架構。

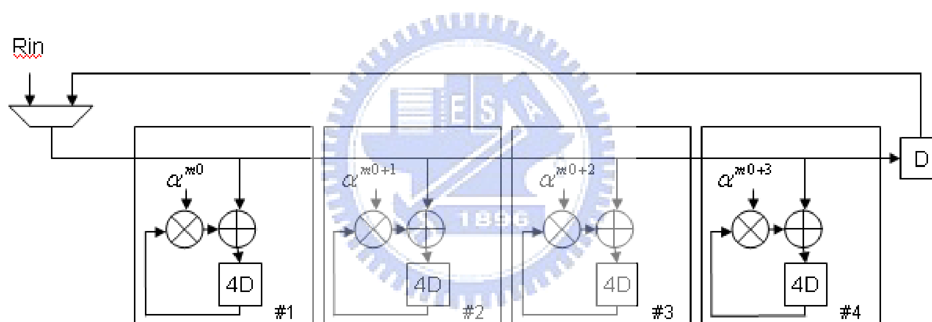


圖 21 錯誤症狀計算陣列之摺疊架構

將經過摺疊演算法調整後的硬體架構之乘加器群及暫存器檔案所連接的輸入端以及輸出端明顯的表示出來，其中每個乘加器擁有三個輸入端及兩個輸出端，從各自的暫存器中讀取上一個回合所乘累加的結果，乘上該運算器在該時間點各自對應的 α 值，再加上此回合外部輸入的 R_{n-i} 後，再存回暫存器。因此三個輸入來源分別為累加暫存器、相對應的 α 值以及 R_{n-i}；輸出則是存入累加暫存器，並將 R_{n-i} 送入銜接暫存器。如圖 22 所顯示。

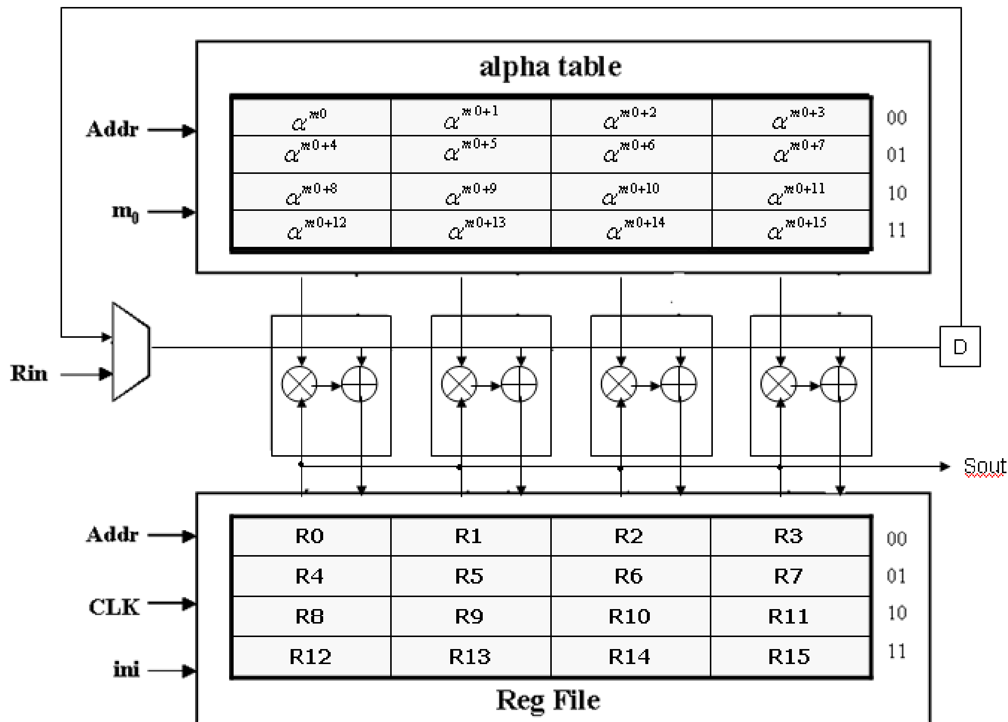


圖 22 錯誤症狀計算硬體之乘加器與暫存器規劃

由圖 22 中可以看出，α 資料表及暫存器檔案在每個時間點各會輸出四筆資料分別送入四個乘加器中，因此在α資料表及暫存器檔案中可以將同一時間點的四筆資料設定為同一組，由 Addr 來給予定址命名。舉例來說，當第一個時間點時，Addr 送入 00 值，因此α資料表會送出$\alpha^{m_0}\sim\alpha^{m_0+3}$到四個乘加器；同理，暫存器檔案也會送出上一回合儲存在 R0~R3 中的乘累加結果到乘加器中做運算；然而在第二、三、四個時間點時亦是如此，只要控制不同的 Addr 值就可以完成想要達到的工作。然後如此經過數回合的計算後，Sout 所輸出的值即為錯誤症狀，依照 Addr 的變換，一個時脈可以輸出四筆錯誤症狀，連續四個時脈即可得到全部 16 筆的錯誤症狀。另外，一般典型的參數 m_0 值為 0 或 1，所以α資料表可經由 m_0 值的輸入，選擇輸出$\alpha^0\sim\alpha^{15}$或$\alpha^1\sim\alpha^{16}$兩種模式。

4-3.2 位址產生器的設計

位址產生器(Address Generator)是用來規劃每個時間點所應送入資料表位

址的一個控制機制，依據時脈(Clock)對位址訊號做遞增的動作，其作用就如同一個有計數上限的計數器。每當一個發生，位址訊號就會向上加 1，而達到摺疊係數(Fold Factor)這個計數上限時則歸零。以第三章推導的架構為例子來看，當更正能力 t 為 6 時，原先應該使用 12 個基本運算單元，現在只想使用 4 個基本運算單元來完成，所以必須摺成 3 摺，因此摺疊係數為 3。當知道了摺疊係數是多少後，可以來了解位址計數是如何動作。位址訊號會從 00 開始計數，隨著時脈的觸發，經過 01 到 10，而達到了摺疊係數這個上限，然後回歸到 00。如果是 4-3.1 的架構，更正能力為 8 時，則摺疊係數等於 4，也就是位址計數器會從 00 開始計數，經過 01、10、11 達到摺疊係數上限後又回歸到 00。

圖 23 是位址產生器的方塊圖，其內部有兩個計數器，一個用作位址訊號遞增的計數，另一個則是紀錄硬體運作的回合數，進而了解錯誤症狀計算的完成時間。在圖中的輸入訊號 ini 是一個具重置功能的訊號，能夠讓位址訊號歸零，此外這個訊號也會送給暫存器檔案，將累加暫存器做重置歸零的動作。CLK 則為整個系統所使用的時脈。Fold_factor 訊號是用來規定計數的上限。輸出的 Done 訊號則是告知錯誤症狀的計算已經完成，當 Done 訊號升為高位準時，就是告知此時的資料是可以截取的，其餘時間 Done 訊號均為低位準，也就是錯誤症狀的計算還未完成，資料是不可取的。

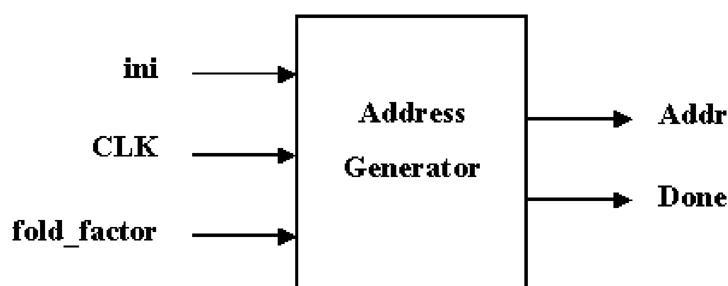


圖 23 錯誤症狀計算硬體之位址產生器

4-4 尋找錯誤位置之硬體摺疊架構

從演算法及數學運算式子可以發現，計算錯誤症狀(Syndrome)和尋找錯誤位置(Chien Search)的計算很類似，所以希望利用這個共同點把兩塊硬體合併成一塊硬體，進而減少硬體的使用。在此先介紹一般尋找錯誤位置的作法以及其硬體架構，然後在往後的章節再介紹如何改變尋找錯誤位置的硬體，使得其硬體架構能夠和計算錯誤症狀共用。

4-4.1 一般方法之尋找錯誤位置硬體摺疊架構

由 2-4.3 小節可知，錯誤位置的尋找是將伽羅瓦場中所有的元素代入由 BM 演算法所計算出的錯誤位置多項式而得到。假設我們的更正錯誤能力為 t，則由 BM 演算法求出的錯誤位置多項式如下：

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_t x^t \qquad \text{(式 4-7)}$$

然而為了使硬體規則化，我們把 $f(x)$ 改變成下面的形式：

$$f(x) = (((f_t x + f_{t-1})x + \cdots)x + f_1)x + f_0 \qquad \text{(式 4-8)}$$

接著將伽羅瓦場中所有的元素 $\alpha^0 \sim \alpha^{254}$ 代入 x，每個元素 α^i 從輸入端開始經過 t+1 個係數的乘累加，存到最後一個暫存器的值就是 $f(\alpha^i)$。這整體的架構是以運算陣列的形式管線化(Pipeline)進行，所以輸入端可以連續的輸入，而當此時刻最後一個暫存器得到某一元素計算結果的值，下一個元素的計算結果將會在下一個時刻就得到。上述的硬體架構如圖 24。

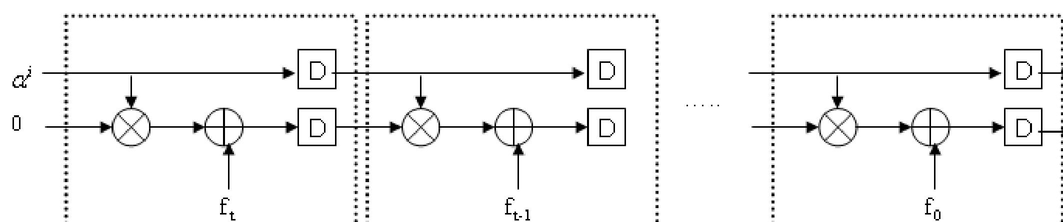


圖 24 更正錯誤能力 t 之尋找錯誤位置的硬體架構

4-4.2 一般方法之尋找錯誤位置硬體摺疊架構之乘加運算器及暫存器規劃

以 4-4.1 所介紹的硬體架構為基礎，開始建立出尋找錯誤位置的硬體架構，接著再利用摺疊演算法的推導，把原本的硬體架構做摺疊處理，接著再將修改過後的摺疊架構取代原本尋找錯誤位置的硬體架構。

在開始摺疊尋找錯誤位置的硬體架構之前，首先一樣必須考慮採用的基本運算單元個數。由上述的介紹可以得知，如果更正錯誤的個數為 t，則需要 t+1 個基本運算單元的陣列才能夠完成我們要的工作。我們現在以更正錯誤能力最大為 8 來說明，所以必須要有 9 個基本運算單完。要做摺疊之前，首先決定要使用幾個基本運算單元來完成，我們也是像計算錯誤症狀一樣採用四個基本運算單元為一組，也就是希望摺疊後的硬體，只使用四個基本運算單元。於是可以將 9 個乘加器分成三列，每列都具有四個基本運算器，而第三列卻只需要用到一個基本運算器就足夠了。如圖 25 所示。

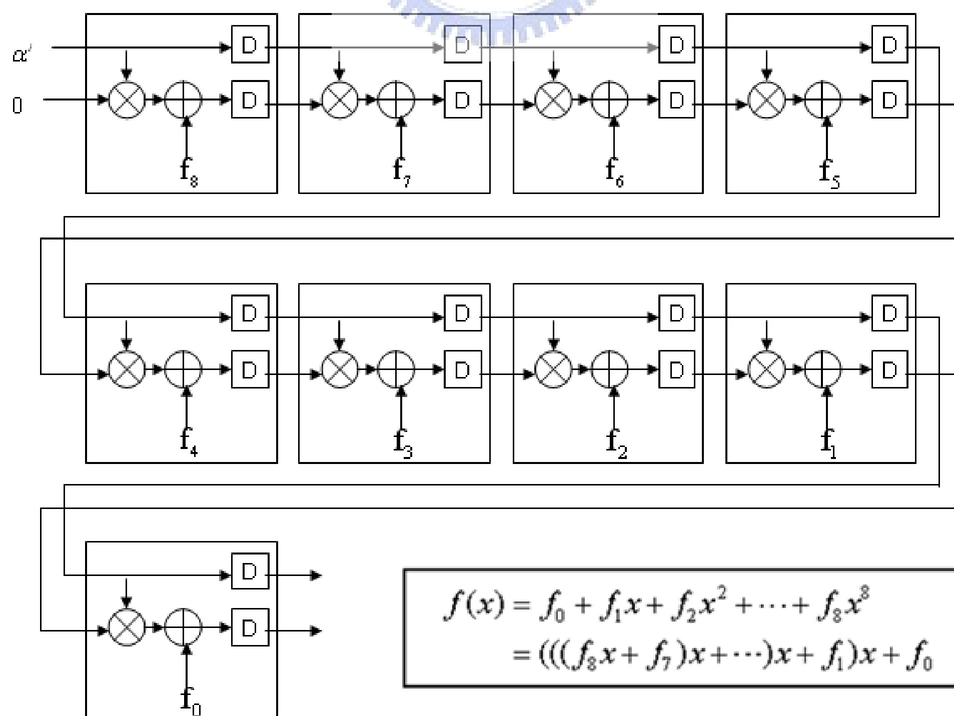


圖 25 更正錯誤能力 t=8 之尋找錯誤位置的硬體架構

現在將 9 個基本運算單元分成三列，也就是打算將原本一個回合內必須完成的動作分成三個時間點來運算，每一列各在一個時間點完成。因此每個暫存器必須變成三倍大小，接著在列與列之間的每個向前路徑上加入資料銜接的暫存器，儲存此刻時間點最後運算的值，等到下個時間點使用。當第一個時間點時，僅有第一列的四個乘加器會動作，並且將運算後的值存入該列各暫存器的第一個位置中，而第二、三列均不動作；當第二個時間點時，只有第二列的四個乘加器會動作，並且將運算後的值存入該列各暫存器的第二個位置中，而第一、三列均不動作；當第三個時間點時，只有第三列的乘加器會動作，並且將運算後的值存入該列暫存器的第三個位置中，而第一、二列均不動作。由於只要經過九個乘累加運算就可以得到我們要的值，也就是在第三列的第一個基本運算單元計算存入的值，所以在第三個時間點時，只需要一個基本運算單元，經由這個基本運算單元所算出來的值就是我們要的結果。因為摺疊的關係，在第三個時間點時，還是有四個基本運算單元可以使用，但是只會使用到一個，我們只要注意控制這個基本運算單元即可。圖 26 表示了上面的敘述。

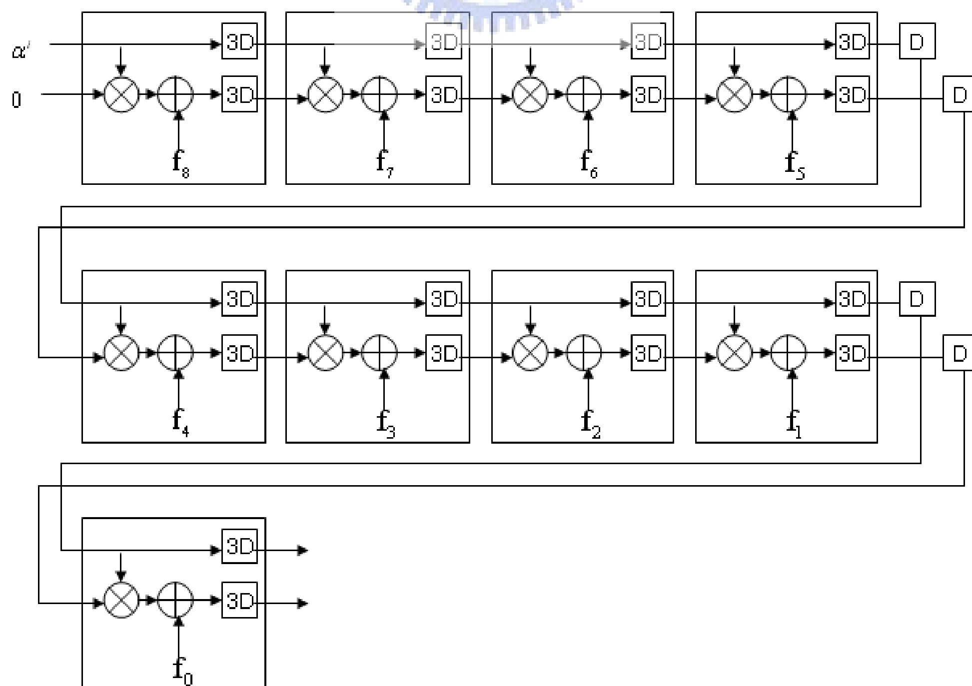


圖 26 八次尋找錯誤位置器陣列之時序規劃

如此一來，在不同工作時間點，這三列各個相對應位置的乘加器與暫存器均不互相衝突，所以可以將這三列摺疊成一列，如圖 27。圖中的四個乘加器在一個回合中的三個不同時間點分別處理原先三列乘加器所應該處理的資料。此外，兩個多工器只有在每回合的第一個時間點才會選擇α^i或是 0 來當作輸入，其餘的時間點將會選擇銜接暫存器來當作輸入來源。

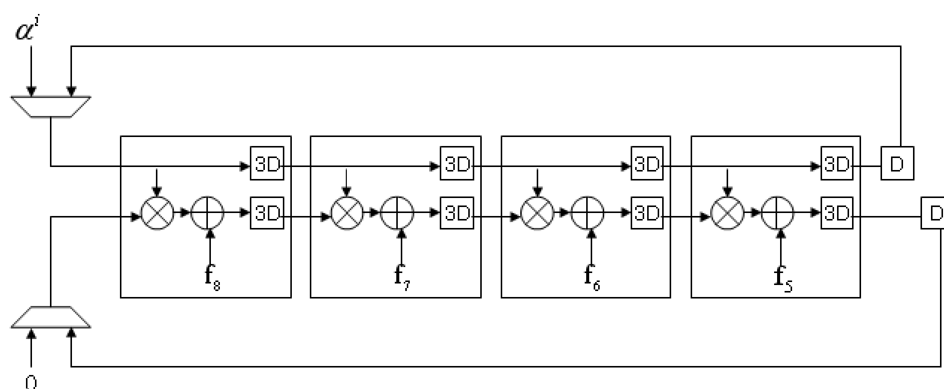


圖 27 八次尋找錯誤位置器之摺疊架構

根據摺疊後的架構，每個基本運算器在不同時間點的輸入輸出必須仔細的安排，圖 28 列出這四個乘加運算器在不同時間點暫存器檔案存取的規劃。在動作開始進行之前，ini 訊號會先讓三個暫存器檔案重置，暫存器檔案 A 與 D 會被歸為 0，而儲存係數的暫存器則會將要計算的多項式係數載入。接下來在每一個回合的第一、二、三個時間點，三個檔案根據 00、01、10 位址所指示的暫存器送出資料至乘加運算器中計算，當然，除了係數暫存器外，其餘兩套暫存器也會將結果存入相對應的位置。至於多工器的選擇，當暫存器檔案的位址指定為 00 時，才切換為α^i或是 0，其餘時間皆以銜接暫存器當做輸入來源。此外，暫存器檔案 D 的輸入 t 值是用來選擇多項式計算結果所存入的暫存器位置。假設 t=8，錯誤位置多項式為一個八次方多項式，因此 Q 所送出的資料應為 D8 的值；若 t=5 時，則為 D5 的值。

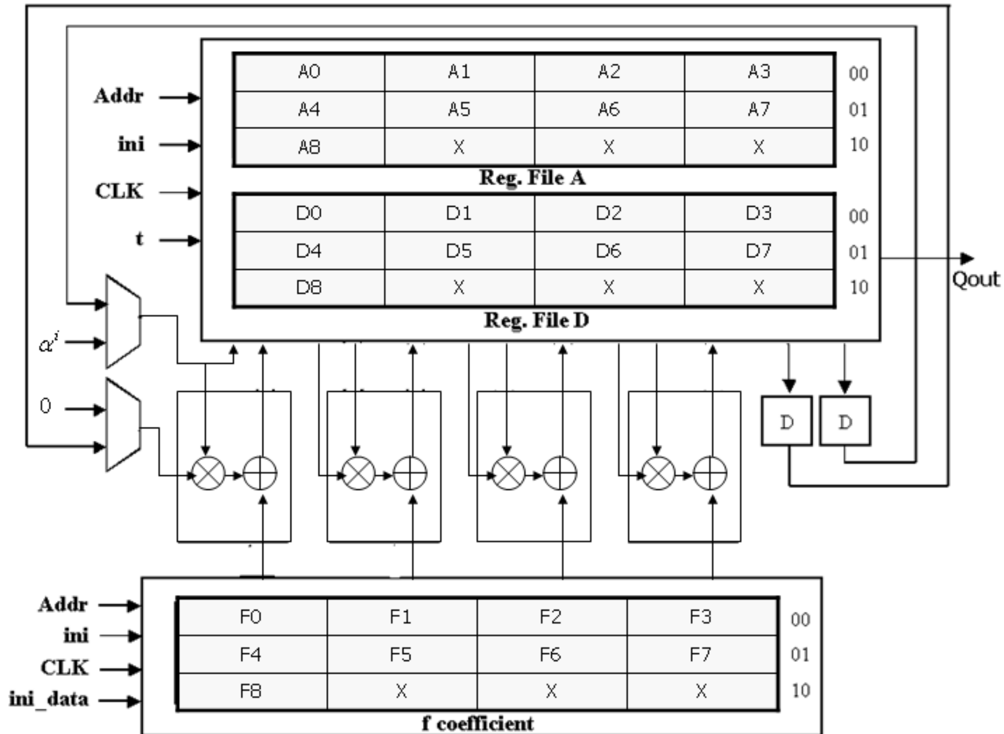


圖 28 尋找錯誤位置器硬體之乘加器與暫存器規劃

4-4.3 變形之尋找錯誤位置硬體摺疊架構

要得到錯誤位置的方法必須對從 BM 演算法運算出的錯誤位置多項式做尋根的動作，這部份在 4-4.1 小節已經有介紹過。我們還是依照(式 4-7)及(式 4-8)的數學式子做硬體架構修改，把基本運算單元修改成像尋找錯誤症狀的形式，當然也必須改變資料流的方式。4-4.1 小節一般的方法是讓伽羅瓦場中的元素α^i當作輸入，錯誤位置多項式的係數固定在每個基本運算單元內送同樣的值做乘累加，假設更正錯誤能力為 t，則經過 t+1 個基本運算單元計算後，可得到一個元素計算的結果，如圖 24 所示，這個方法是每個基本運算單元間都有關聯，上一個基本運算單元計算過後的值存到暫存器，再傳給下一個基本運算單元做計算，一直如此傳遞下去，直到經過 t+1 個基本運算單元才結束；接著是說明修改過的尋找錯誤位置之方法，由於希望能夠和尋找錯誤症狀一樣的硬體架構，所以朝著這方向做修改，組成尋找錯誤症狀硬體架構的每個基本運算單元之間是沒有關聯的，並不是由上一個基本運算單元的暫存器傳送過來做運算，而是由輸入的值以

及各自暫存器的儲存值做乘累加的計算，因此將原先的輸入值α^i換成錯誤位置多項式的係數 f_i，而將α^i值放置在與尋找錯誤症狀的基本運算單元內的α^i值相同的位置，如圖 29 所示。

如此修改硬體架構也可以和一般方法的尋找錯誤位置硬體架構有相同的運算結果，而且連續輸入完一次錯誤多項式的係數後即可以得到所有所有伽羅瓦場中元素代入的值，但是這樣卻需要龐大的基本運算單元來做計算，因此摺疊硬體的架構可以再一次的發揮功用，由下個小節再加以敘述。

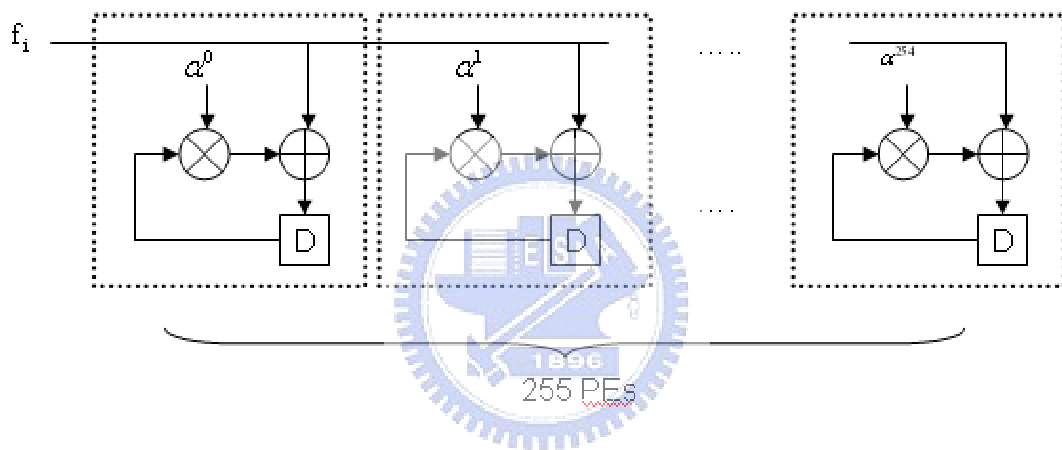


圖 29 變形之尋找錯誤位置陣列硬體架構

4-4.4 變形之尋找錯誤位置硬體摺疊架構之乘加運算器及暫存器規劃

由上一小節構想的陣列架構為基礎，來建構出變形之尋找錯誤位置的硬體，接著利用摺疊演算法讓硬體做摺疊動作，以達到我們所要的需求。由於希望能夠和尋找錯誤症狀共用相同的硬體，因此就朝著這個方向做摺疊修改。因為尋找錯誤症狀的陣列架構是使用四個基本運算單元，所以這裡的陣列架構也決定使用四個基本運算單元。根據演算法的定義，將伽羅瓦場中所有的元素代入錯誤多項式中可以尋找錯誤位置，假設我們在此的運算都是建立在$GF(2^8)$伽羅瓦場中，也就是需要 255 個基本運算單元。由於採用 4 個基本運算單元，所以將 255

個基本運算單元分為 64 列，如圖 30。

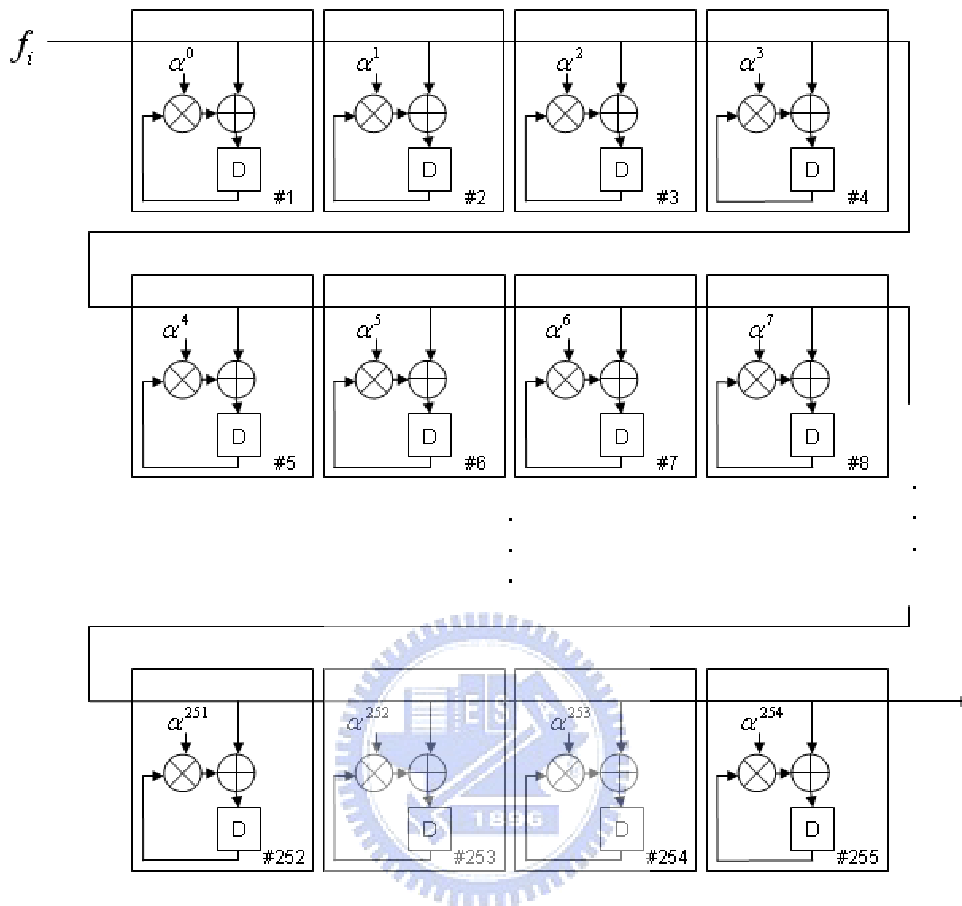


圖 30 更正錯誤能力 t 之變形尋找錯誤位置的硬體架構

為了能夠和尋找錯誤症狀共用相同的硬體，所以改變了資料流的方式，也因此在修改成摺疊架構時，資料流的方式也要有稍稍的改變。如果依照原本的資料流方式，一筆資料的輸入，也就是一個回合的開始，需要經過 64 個時間點的運算，並且在每個時間點都存到各自的暫存器，如此需要原本 64 倍的暫存器大小，這樣不僅沒有達到節省硬體的效果，反而更浪費硬體，也違背了摺疊架構的原則。因此在資料流的方式上做了些改變，先把錯誤多項式的係數都先暫存起來當作輸入，並且在各列送入四個伽羅瓦場中不同的元素至乘加器做乘累加。第一個係數輸入只在第一列與該列的四個不同伽羅瓦場元素做乘累加，並且將計算後的值存入暫存器，然後第二個係數接著輸入，也是只在第一列與該列的四個不同

伽羅瓦場元素做乘累加，且將計算後的值存入暫存器，第三個到第九個係數也是如此，然後重回輸入第一個係數，而此時才會開始在第二列與這列的四個不同伽羅瓦元素做乘累加，一直重複運算，直到 64 列都運算完才結束。由於每一列有四個基本運算單元，所以每一列在計算時，當輸入九個錯誤多項式係數後，可以得到四個元素代入多項式的值，在此時也就可以將值從暫存器中取出，而當下一列在計算時就可以重複的再使用這個暫存器。由於每一列的輸入值都是由一開始就存好的暫存器中取值，也因此不需要銜接暫存器。這也算是變形的摺疊架構，如同圖 31 所示。

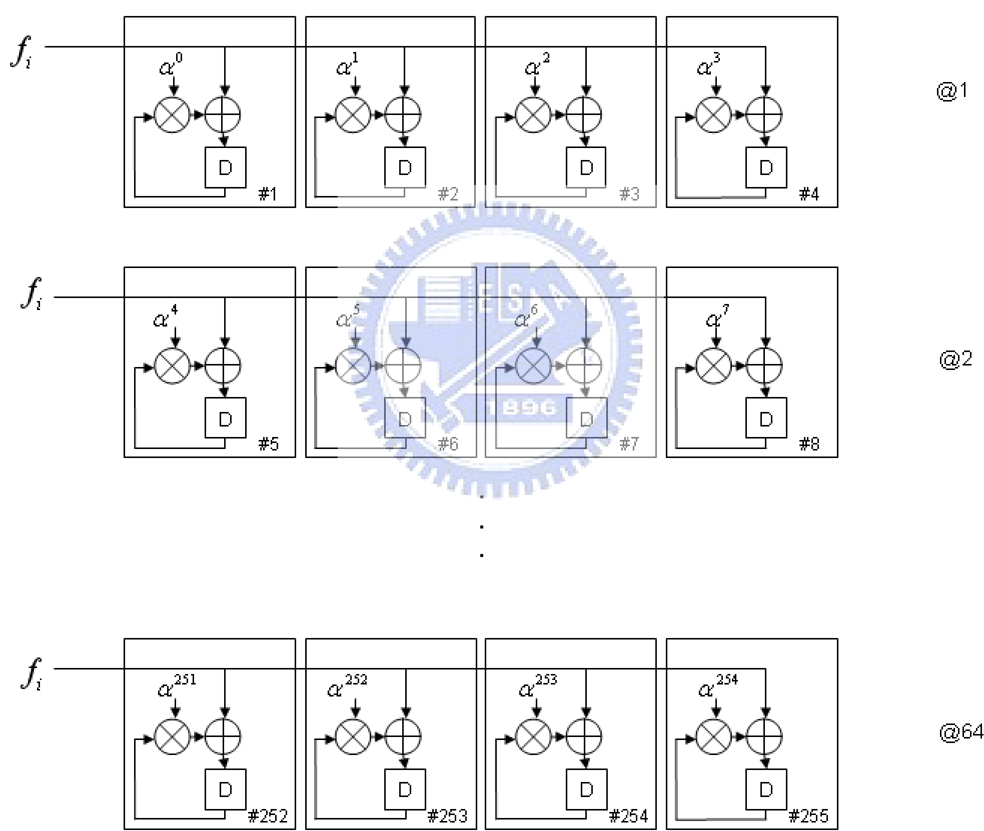


圖 31 更正錯誤能力 t 之變形尋找錯誤位置陣列之時序規劃

同樣的，由圖 31 可以知道，每一列的乘加器在做運算的時候，都是互相沒有關係的，也就是第一列的乘加器在做運算的時候，其他列的乘加器均不會有動作；當換到第二列的乘加器在做運算的時候，其他列的乘加器也是均不會有動作；以此類推，當其他某一列的乘加器在動作的時候，其餘的乘加器都均不會有

動作。至於暫存器的部份，因為在每一列運算完成存回暫存器的值就是把伽羅瓦場中元素代入錯誤多項式的值，因此在某一列經過九次運算後的值就可以取出，當下一列要做運算時可以重複只用這個暫存器，並不會有資料錯誤的問題。由於每一列的乘加器與暫存器都沒有同個時間同時運作的可能，因此可以把 64 列的基本運算單元摺疊成一列，如圖 32 所示。

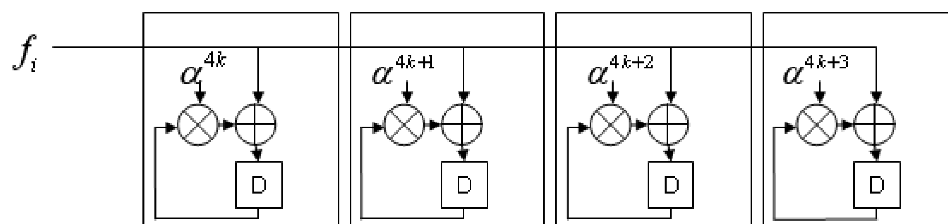


圖 32 更正錯誤能力 t 之變形尋找錯誤位置之摺疊架構

將摺疊演算法調整後的架構分離為乘加器群及暫存器檔案，從暫存器中讀入的值，乘上該乘加器在該時刻對應的 α 值，再加上此時刻輸入的錯誤多項式係數 f_i 後，在存回暫存器。在暫存器送出值給乘加器的路徑上增加一個多工器，因為每次輸入九個錯誤多項式係數都是一個新的回合，所以要讓一開始送入乘加器的值歸零，這樣動作才會正確。如圖 33 所示。

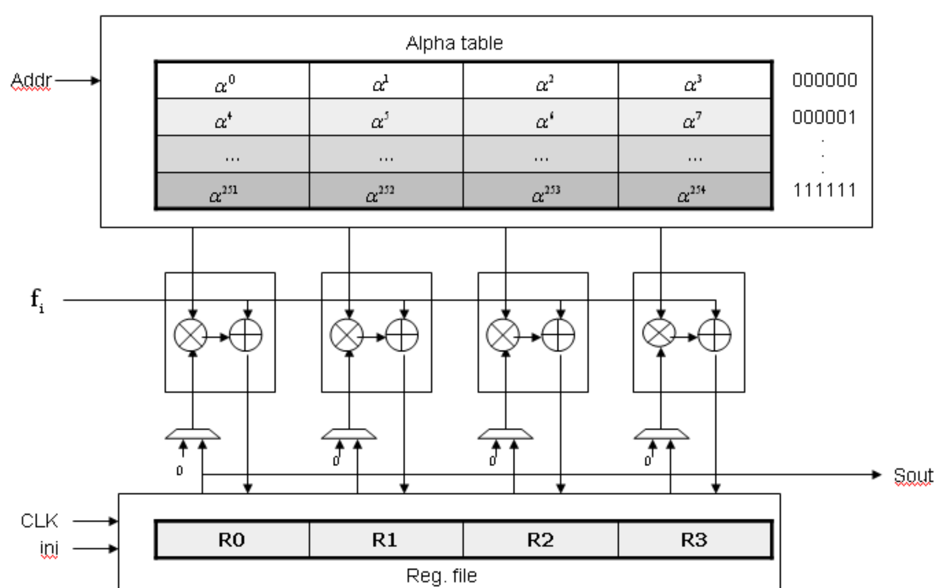


圖 33 變形尋找錯誤位置硬體之乘加運算器與暫存器規劃

根據摺疊之後的架構，每個乘加器在不同時刻的輸入與輸出就必須仔細的安排，圖 33 列出這四個乘加器在不同的時刻暫存器檔案存取的規劃。在動作開始進行之前，ini 訊號會先讓暫存器檔案重置。接著α資料表及暫存器檔案在每個時刻會讀入四筆資料分別送入乘加器中，因此在兩個資料表中可以將同一列的四筆資料作綑綁的動作，暫存器檔案由於只有一列，乘加器運算完的值就只會寫入這一列，所以不需要額外給予定址命名。而α資料表因為有 64 列，所以需要由 Addr 來給予定址命名。舉例來說，當第一個時刻時，Addr 送入 000000 值，因此α資料表會送出$\alpha \sim \alpha^3$到四個乘加器做運算；當第二個時刻時，Addr 送入 000001 值，α資料表就會送出$\alpha^4 \sim \alpha^7$到四個乘加器做運算；而在其他時刻時，以此類推，只要輸入不同的 Addr 值就可以選擇α資料表要送出哪四個值給乘加器做運算。當每一個時刻計算完成時，Sout 所輸出的值即為$f(\alpha^{Addr}) \sim f(\alpha^{Addr+3})$，一個時刻可以算出四個$f(\alpha^i)$。

在推導完乘加器與暫存器檔案的規劃後，再配合一個位址產生器來調整每個時刻所需要選擇的α資料表，以及何時暫存器送給乘加器的值要歸零，就能使得尋找錯誤位置硬體正確無誤的動作。

圖 34 就是具有此控制機制特質的位址產生器方塊圖，這個方塊的運作方式與處理錯誤症狀架構的位址產生器是類似的，內部含有三個計數器，一個依照時脈(CLK)將多項式係數計數器(f_cnt)向上遞增計數，每當輸入完九個多項式係數後，多項式係數計數器就會歸零；一個是位址訊號(Addr)，每當輸入九個多項式係數則向上遞增計數，其餘時間位址訊號都不改變；另外一個是用來計算全部伽羅瓦場的元素計算完的時間，當計算經過 64 回合後，代表最後一組伽羅瓦場的元素都計算完成，進而提升完成訊號(Done)訊號為高位準。重置訊號(ini)則是讓計數的位址歸零。

此外，由於變形的尋找錯誤位置架構是一次算四個伽羅瓦場中的元素，所以在輸入九個多項式係數一次後，就可以得到四個想要求的值，然後每經過一次的輸入九個多項式係數就可以得到四個伽羅瓦場元素代入的值。

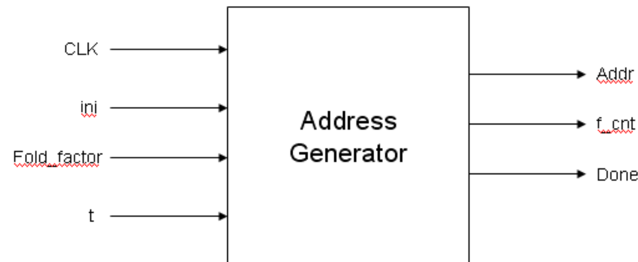


圖 34 變形尋找錯誤位置之位址產生器

4-5 尋找錯誤症狀硬體架構與尋找錯誤位置硬體架構的合併

從(式 4-6)及(式 4-8)這兩個數學式子中，不難發現，這兩個式子有著相同的特性，就是都是做乘累加，也因為這樣的特性，讓我們思考在這兩部份的硬體架構是否也有共通點。由前面章節所介紹的尋找錯誤症狀與尋找錯誤位置的硬體架構可以發現，雖然在架構上並不是完全一樣而有辦法直接套用，但是只需要修改其中一套硬體架構，讓兩套硬體架構完全一樣，使得這兩部份的演算法可以用同一套硬體架構就可以完成。因此我們選擇了修改尋找錯誤位置的硬體架構，在4-4.3 與 4-4.4 兩小節詳細的說明了如何修改這部份的硬體架構。而在這個章節中，會詳細的說明如何將這兩套硬體架構合併成一套，並且如何控制同一套硬體卻可以做兩個不同的工作。我們在這裡把這套硬體稱為尋找錯誤症狀-位置(Syndrome-Chien search)的硬體架構。

4-5.1 尋找錯誤症狀-位置(Syndrome-Chien search)的硬體架構

已經知道尋找錯誤症狀與尋找錯誤位置的硬體架構具有相同的特性可以將這兩部份硬體合併成一套，而且在 4-4.3 與 4-4.4 兩小節已經將尋找錯誤位置的硬體架構做了修改，因此可以由之前推導的摺疊硬體架構圖下手，做合併的工

作。從圖 21 與圖 32 的摺疊架構可以發現，尋找錯誤症狀所用到的暫存器比較多，而且也有使用到銜接暫存器，因此可以就使用尋找錯誤症狀的摺疊架構當主體，只需要在輸入部份的多工器多加入錯誤多項式的係數即可。如圖 35 所示。

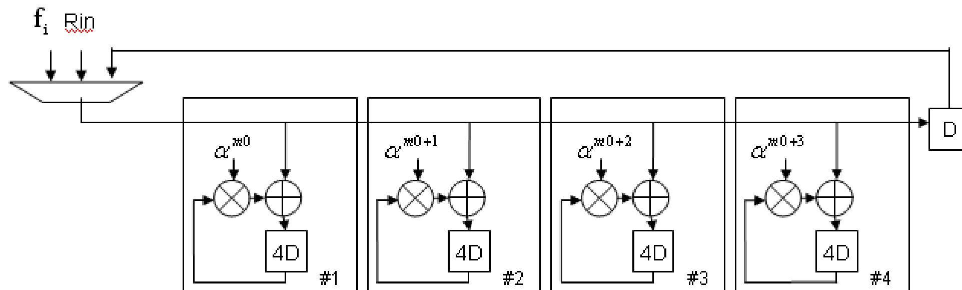


圖 35 尋找錯誤症狀-位置陣列之摺疊架構

4-5.2 尋找錯誤症狀-位置(Syndrome-Chien search)硬體架構之乘加運算器與暫存器規劃

了解尋找錯誤症狀-位置的硬體摺疊架構後，我們還必須要知道訊號是如何送入乘加器做運算，並且更加明確的定義這些訊號。由圖 22 與圖 33 可以看出有 α 資料表和暫存器檔案以及如何控制這些資料的訊號。在 α 資料表的部份，尋找錯誤位置所要使用的表比較大，並且可以包含尋找錯誤症狀的表，因此在規劃 α 資料表的時候只需要使用尋找錯誤位置的部份即可。在暫存器檔案的部份，尋找錯誤症狀所要使用的暫存器比較多，而在計算錯誤位置的時候，可以使用第一列的四個暫存器，並不會有衝突，因此在規劃暫存器檔案的時候只需要使用尋找錯誤症狀的部份即可。在銜接暫存器的規劃上，因為尋找錯誤症狀會使用到，因此還是要把這個暫存器加上去，而當在計算錯誤位置的時候，不會選擇他當輸入，所以就可以當作沒有這一部份。整個規劃如圖 36 所示。

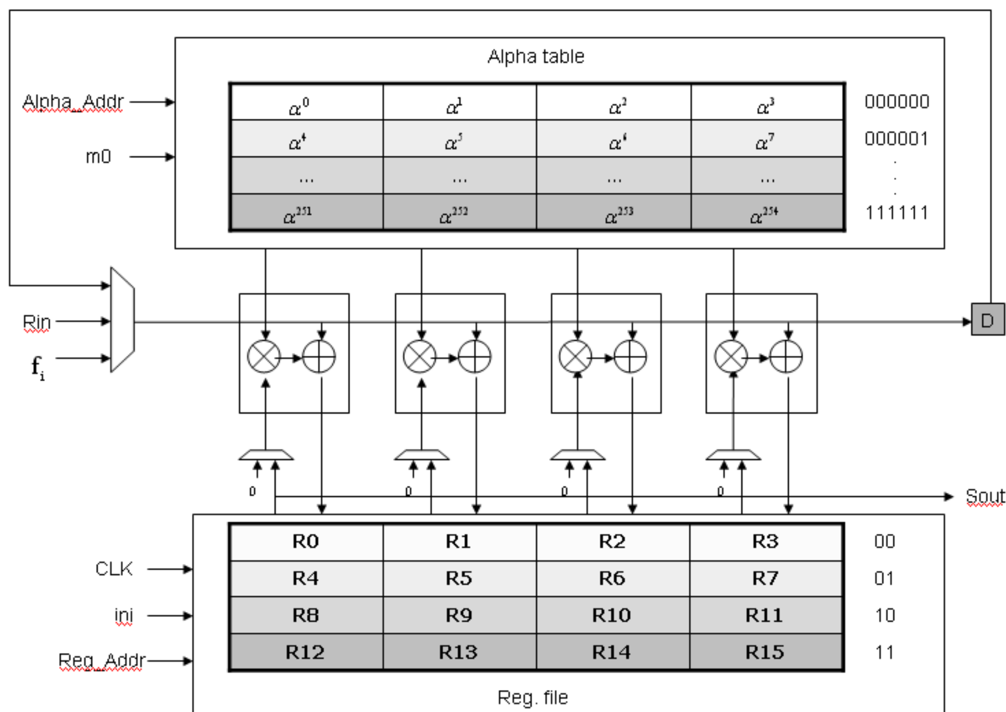


圖 36 尋找錯誤症狀-位置硬體之乘加運算器與暫存器規劃

由於一套硬體要做兩件不同的工作，因此控制訊號會比較複雜，但是只要規劃好還是一樣可以正確工作。假設我們選擇的工作是尋找錯誤症狀，在第一個時間點時輸入會選擇 R_{n-i}，α 資料表也會送出 $\alpha^{m_0} \sim \alpha^{m_0+3}$ 到四個乘加器，而暫存器檔案也會送出上一回合儲存在 R0~R3 中的乘累加結果到乘加器；在第二、三、四個時間點時輸入會選擇銜接暫存器的值，α 資料表和暫存器檔案也會隨著送入不同的 Addr 值作切換而將值送入乘加器做運算，經過多個回合的運算就可以得到錯誤症狀的值。接下來我們選擇的工作換成尋找錯誤位置，當第一個回合第一個時間點時，會選擇錯誤多項式係數 f_8 當作輸入，第二個時間點時，則是由 f_7 當作輸入，其他時間點以此類推，共有九個時間點，才算一個回合，而在此回合中，α 資料表就只會送出 $\alpha^0 \sim \alpha^3$ 到四個乘加器，當下個回合時才會送出 $\alpha^4 \sim \alpha^7$ 到四個乘加器；在暫存器檔案送值到乘加器的路徑上多加一個多工器，當在每一回合的第一個時間點時會選擇 0，其餘時間點都會選擇暫存器檔案的值送入乘加器。每個回合結束都可以得到四個伽羅瓦場元素代入錯誤多項式的值，經過 64 個回

合後，全部伽羅瓦場元素代入錯誤多項式的值皆可以得到。

4-5.3 位址產生器的設計

在看完上述概略介紹選擇不同的工作時會做哪些運算，並且希望這些運算能夠達到預期的結果。因此位址產生器(Address Generator)的設計在這裡就相當重要，因為必須控制一套硬體做兩件不同工作，而經由位址產生器產生的訊號來控制硬體，使得硬體能夠正確的工作。

因此我們在設計上使用了四個計數器：Alpha_Addr、Reg_Addr、f_cnt、count。如圖 37 的位址產生器方塊圖。我們先說明 s 訊號的作用，s 訊號是用來選擇現在的工作室尋找錯誤症狀或者是尋找錯誤位置，而以下的說明皆假設更正錯誤能力為 8。Alpha_addr 是用來控制 α 資料表該送什麼值給乘加器做運算，假設現在的工作是尋找錯誤症狀，則會依據時脈(Clock)做向上加 1 的動作，而達到摺疊係數(Fold Factor)這個計數上限時則歸零，如此一直循環下去；而當現在的工作是尋找錯誤位置時，會根據 f_cnt 這個計數器是否要向上加 1，當 f_cnt=8 時，則向上加 1，其餘時間皆不改變。Reg_Addr 是用來控制暫存器檔案該送哪個暫存器裡面的值給乘加器做運算，假設工作是尋找錯誤症狀，則會依據時脈(Clock)做向上加 1 的動作，而達到摺疊係數(Fold Factor)這個計數上限時則歸零，如此一直循環下去；而當工作是尋找錯誤位置時，則會停留在 00 不會改變。F_cnt 是用來控制當工作是尋找錯誤位置時，現在的時刻該輸入哪個錯誤位置多項式的係數，以及是否算完一個回合了，所以當工作在尋找錯誤症狀時是不會改變的。Count 則是用來標記該項工作運算的回合數，進而掌握完成的時間，所以不管是哪個工作時，都會依據時脈(Clock)做向上加 1 的動作。在方塊圖中的輸入 ini 訊號是重置的訊號。輸出的 Done 訊號則是告知是否此時資料已經計算完，而可以截取的。

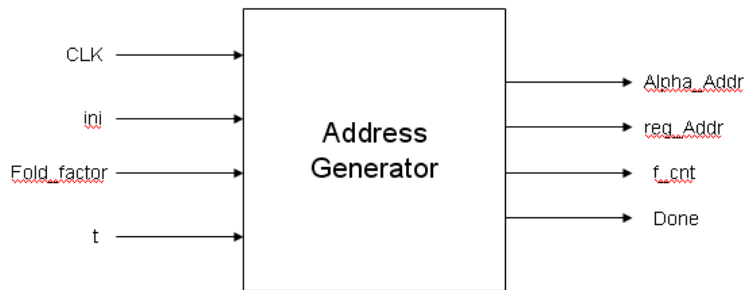


圖 37 位址產生器

4-6 尋找錯誤位置多項式及錯誤大小評估多項式之軟體實現

在尋找錯誤位置多項式及錯誤大小評估多項式這部分，2-4.2 小節中提到有 Berlekamp-Massey 疊代演算法及最大公因式演算法兩種。BM 疊代演算法會在經過 2t(t 為更正錯誤能力)次的遞迴運算後得到錯誤位置多項式，然後再和錯誤症狀做運算找到錯誤大小評估多項式。而最大公因式演算法則是讓錯誤症狀行程的多項式與 x^{2t} 一直重複做輾轉相除，直到餘數的最高次方項小於 t 才停止，因為運算時需要做伽羅瓦場的反轉，因此增加了許多運算量，而這個演算法的好處就是可以同時將錯誤位置多項式及錯誤大小評估多項式求出。也由於運算量的關係，不管是在軟體或是硬體的實作上，都是 BM 疊代演算法被運用的較多。

這兩種演算法還是被繼續研究，希望能提出更好的演算法或硬體架構能夠減少運算量或硬體空間。而有學者在最大公因式演算法又提出了新的演算法，稱做修改的最大公因式演算法(Modified Euclid algorithm, ME)[10][11][15][16]，這個演算法將不需要做伽羅瓦場的反轉，也因此省了許多運算。以下為 ME 之演算法。

$$A(x) = x^{2t}$$
$$S(x) = \sum_{k=1}^{2t} S_k x^{2t-k} \quad \text{(式 4-9)}$$

$S(x)$ 為錯誤症狀多項式。

$$
\begin{aligned}
&R_0(x)=A(x) \;,\; Q_0(x)=S(x)\\
&\lambda_0(x)=0 \;,\; \mu_0(x)=1\\
&\gamma_0(x)=1 \;,\; \eta_0(x)=0
\end{aligned}
\qquad \text{(式 4-10)}
$$

$$
\begin{aligned}
R_i(x)=&[\sigma_{i-1}b_{i-1}R_{i-1}(x)+\overline{\sigma_{i-1}}a_{i-1}Q_{i-1}(x)]\\
&-x^{|l_{i-1}|}[\sigma_{i-1}a_{i-1}Q_{i-1}(x)+\overline{\sigma_{i-1}}b_{i-1}R_{i-1}(x)]
\end{aligned}
\qquad \text{(式 4-11)}
$$

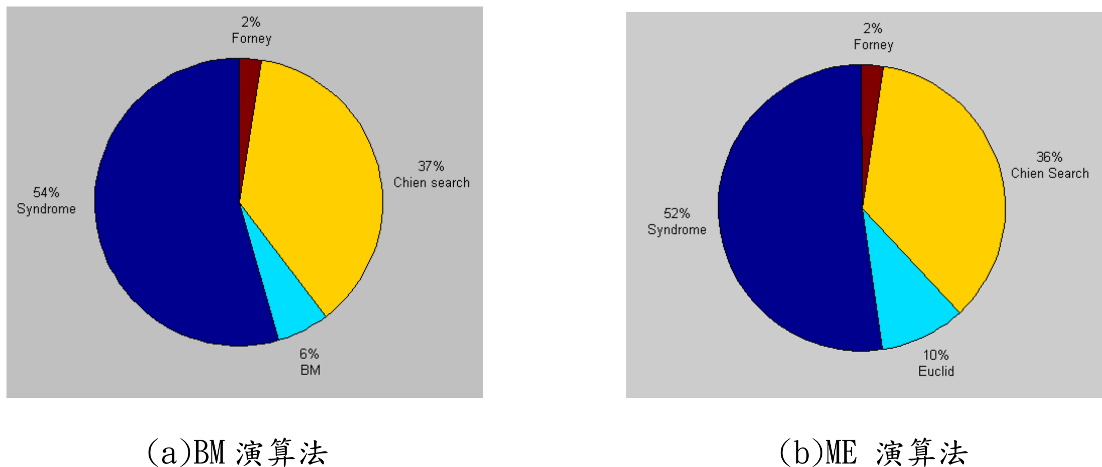
$$
\begin{aligned}
\lambda_i(x)=&[\sigma_{i-1}b_{i-1}\lambda_{i-1}(x)+\overline{\sigma_{i-1}}a_{i-1}\mu_{i-1}(x)]\\
&-x^{|l_{i-1}|}[\sigma_{i-1}a_{i-1}\mu_{i-1}(x)+\overline{\sigma_{i-1}}b_{i-1}\lambda_{i-1}(x)]
\end{aligned}
\qquad \text{(式 4-12)}
$$

$$Q_i(x)=\sigma_{i-1}Q_{i-1}(x)+\overline{\sigma_{i-1}}R_{i-1}(x) \qquad \text{(式 4-13)}$$

$$\mu_i(x)=\sigma_{i-1}\mu_{i-1}(x)+\overline{\sigma_{i-1}}\lambda_{i-1}(x) \qquad \text{(式 4-14)}$$

$$
\begin{aligned}
&l_{i-1}=\deg(R_{i-1}(x))-\deg(Q_{i-1}(x))\\
&\sigma_{i-1}=1 \quad \text{if } l_{i-1}\ge 0\\
&\sigma_{i-1}=0 \quad \text{if } l_{i-1}<0
\end{aligned}
\qquad \text{(式 4-15)}
$$

(式 4-10)為初始值，接著重複運算(式 4-11)~(式 4-15)，直到 $R_i(x)$ 的最高次項小於 t 時則停止，而此時 $\lambda_i(x)$ 即為我們要求的錯誤位置多項式，$R_i(x)$ 為錯誤大小評估多項式。由於此演算法少了要做伽羅瓦場的反轉，因此少了許多運算量，也少了許多硬體。但是 ME 演算法實現出來所需要使用的硬體還是比 BM 演算法來的大，而在軟體的運算時間上，我們做過測試，是用更正錯誤能力 t=8，計算相同的樣本，ME 演算法所花的時間還是比 BM 演算法來的多。如圖 38 所示。



(a)BM 演算法　　(b)ME 演算法

圖 38 BM 與 ME 演算法運算時間比較

由上述的說明，可以知道不管是軟體還是硬體，BM 演算法還是有著比較好的優勢，因此我們使用軟體來做尋找錯誤位置多項式及錯誤評估大小多項式的部分，使用 BM 演算法。所以我們只要照著 2-4.2.1 小節所介紹的演算法實現即可。

4-7 尋找錯誤大小之軟體實現

這一部分也和前一小節一樣，使用軟體的程式來實現，所以只要有了演算法，就可以很容易的實現出來。這個步驟已經是里德所羅門解碼的最後一個步驟，所以經由前面計算出來的結果，就有能力可以進行尋找錯誤大小的計算。所以由 2-4.4 小節所介紹的佛尼(Forney)演算法推導出來的數學式子(式 2-27)，就可以容易的利用軟體程式將錯誤大小得到，因為這部分沒有其他演算法可以做比較，因此在這部分就用這個演算法來完成我們要的工作。

4-8 里德所羅門解碼之軟、硬體系統整合

我們的里德所羅門解碼器是以軟體為基礎，外掛硬體加速器。所以解碼器有些部分是利用軟體來完成，有些部分是以外掛硬體加速器完成。尋找錯誤位置多項式及尋找錯誤大小是軟體部分來完成，而尋找錯誤症狀及尋找錯誤位置是由外掛硬體加速器完成。因此軟、硬體之間的溝通相當重要，而這裡是用 3-5 節所提到的 On-chip Peripheral Bus 來做溝通。我們利用控制暫存器(Control Register)與狀態暫存器(Status Register)的方式來讓軟、硬體做溝通，也就是軟體處理器會填寫控制暫存器來讓外掛硬體加速器工作，所以外掛硬體加速器會以控制暫存器送出來的訊號來決定要做怎樣的工作，當然這個控制暫存器內的值是控制哪個訊號必須先設計好。控制暫存器是處理器命令外掛硬體加速器的一個介面，而狀態暫存器則是外掛硬體加速器要和處理器溝通的介面。狀態暫存器可以告訴處理器現在的外掛硬體加速器是處於什麼狀態，是否已經將資料處理完而可以送給處理器。狀態暫存器也是一樣，必須先設計好暫存器裡的值代表著什麼意義。所以我們來對控制暫存器及狀態暫存器內的值作定義。

我們使用的外掛硬體加速器就是 4-5.2 小節所設計的硬體架構，所以我們從這一小節的硬體架構來看需要哪些訊號可以控制要做什麼工作。先定義控制暫存器的部分，首先是輸入的資料 data_in，由於兩種工作的輸入皆是八位元，所以我們只需要八位元大小的暫存器即可;接著是選擇要工作尋找錯誤症狀還是工作尋找錯誤位置的訊號 s，因為只有兩種工作，所以只需要一位元即可，當 s=0 時做尋找錯誤症狀，當 s=1 時做尋找錯誤位置；再來是 m0 這個訊號，是在尋找錯誤症狀時需要的訊號，也只需要一位元即可；最後是 ini 重置訊號，當要做不同工作時讓 ini=1，其餘時間都為零，這也是需要一位元。上述的設計如圖 39 所示。

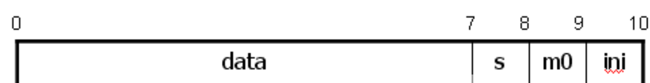


圖 39 控制暫存器

接著來設計狀態暫存器，我們每次由硬體加速器算完的結果資料都是 32 位元，因此需要 32 位元的暫存器，然而我們只需要一位元的 Done 訊號告知處理器這筆資料是否是有效的資料。所以狀態暫存器的設計如圖 40。

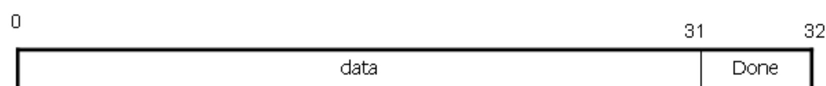


圖 40 狀態暫存器

有了控制暫存器及狀態暫存器，軟體處理器和外掛硬體加速器就可以方便的溝通，我們也就可以容易的加以控制，來達到里德所羅門解碼的功能。圖 41 為控制暫存器及狀態暫存器軟、硬體間的連接。

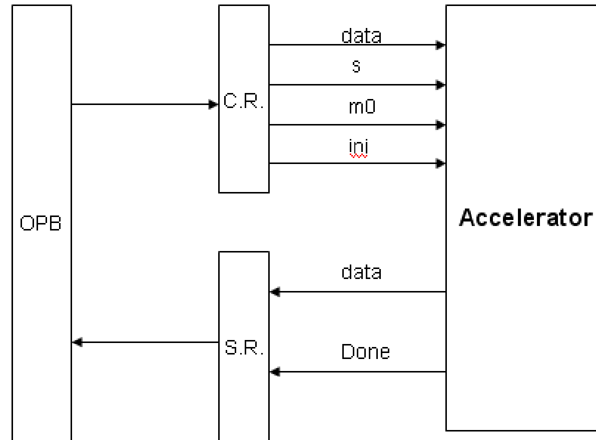


圖 41 控制&狀態暫存器的連接

4-9 實現結果

這裡硬體部分所使用的 VHDL 硬體語言，是可合成(Synthesizable)的 RTL 程式，並非只是行為的描述(Behavior Model)。而我們設計的兩塊硬體在完成一組字碼的字算，各自所需要的延遲時間為(假設我們使用的摺疊係數都是 4)：

- 尋找錯誤症狀：$255 \times 4 \times clk = 1020 \times clk$
- 尋找錯誤位置：$64 \times 9 \times clk = 576 \times clk$

我們將撰寫好的 RTL 程式放在模擬軟體 Modelsim 中做模擬，是以 t=8，m0=0 來進行模擬。測試數據列於附錄。

1. 尋找錯誤症狀

$S_1 = r(\alpha^{1}) = \alpha^{58} = 0x69$　　$S_9 = r(\alpha^{10}) = \alpha^{32} = 0x9D$

$S_2 = r(\alpha^{2}) = \alpha^{125} = 0x33$　　$S_{10} = r(\alpha^{11}) = \alpha^{12} = 0xCD$

$S_3 = r(\alpha^{3}) = \alpha^{100} = 0x11$　　$S_{11} = r(\alpha^{12}) = \alpha^{211} = 0xB2$

$S_4 = r(\alpha^{4}) = \alpha^{115} = 0x7C$　　$S_{12} = r(\alpha^{10}) = \alpha^{254} = 0x8E$

$S_5 = r(\alpha^{5}) = \alpha^{135} = 0xA9$　　$S_{13} = r(\alpha^{11}) = \alpha^{77} = 0x3C$

$S_6 = r(\alpha^{6}) = \alpha^{23} = 0xC9$　　$S_{14} = r(\alpha^{12}) = \alpha^{97} = 0xAF$

$S_7 = r(\alpha^{7}) = \alpha^{34} = 0x4E$　　$S_{15} = r(\alpha^{12}) = \alpha^{22} = 0xEA$

$S_8 = r(\alpha^{8}) = \alpha^{36} = 0x25$　　$S_{16} = r(\alpha^{12}) = \alpha^{252} = 0xAD$

從圖 42 中可以看出一個回合為四個時脈週期，這是當 t=8 時摺疊係數是 4 的緣故。在最後一個回合中輸出的十六個錯誤症狀的值，跟原先經過程式計算的相同。

圖 42 尋找錯誤症狀之 RTL 模擬

2. 尋找錯誤位置

圖 43 為尋找錯誤位置的模擬，這部分因為每 9 個時脈就會送四個值出來，因此只能看到一小部分，為零的時候代表找到錯誤的位置了。

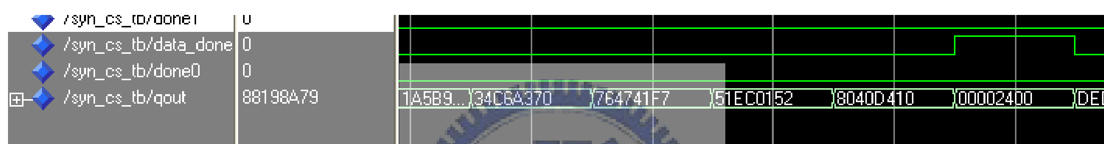
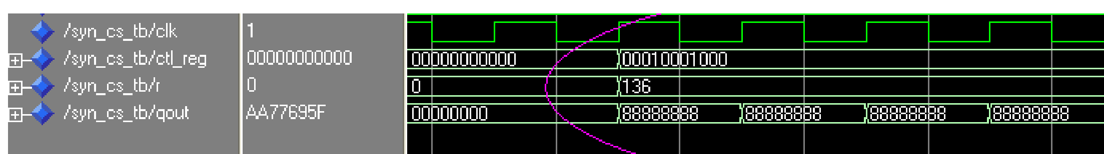


圖 43 尋找錯誤位置模擬

3. 處理器利用控制暫存器來控制硬體模擬



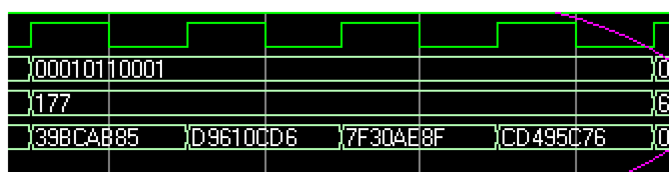


圖 44 利用控制暫存器控制硬體

由圖 44 所示，Ctl_reg 是控制暫存器，前三個位元為 0，也就是 ini、m0、s 都為 0，所以目前要做的工作是尋找錯誤症狀。可以從 ctl_reg 的最後 8 位元看出輸入為 136，也就是 r 這根腳所顯示，因為一開始暫存器裡面的值為 0，因此當有輸入進來則對輸入值做加法的動作，所以我們可以看到在暫存器裡面存的

值是 0X88，當等到下個輸入值進來的時候，會把剛剛存到暫存器裡面的值拿出來和那時刻時間點的 alpha 值做相乘，接著在和輸入值做加法，最後在將值存回暫存器。

4. 硬體利用狀態暫存器和處理器溝通

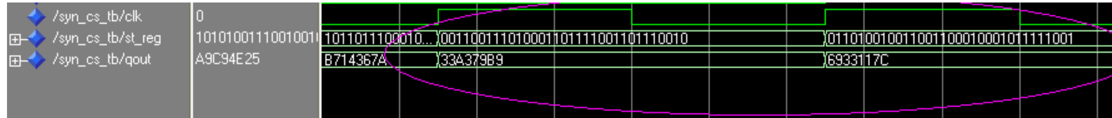


圖 45 利用狀態暫存器和處理器溝通

由圖 45 可以看出，st_reg 就是我們的狀態暫存器，前 32 位元是經由乘累加計算出來的值，但是不一定是我們要的值，必須從狀態暫存器的第一個位元判斷，Done 訊號是否變成 1，我們圈起來的部分剛好是分界點，前一個時間點第一個位元是 0，所以後面的輸出訊號就不是我們要求的所以不拿取，等到下一個時間點時，Dome 訊號變成 1 了，所以在拉起來的這段期間，st_reg 後 32 位元，都是我們要的值，因為和第一點使用同一個測試訊號，所以可以利用在第一點已經先算好的值做比對。這裡我們一次會取出四個有效值。

確定所設計的硬體在功能上沒有問題後，將我們設計的硬體放在 Xilinx 所提供之設計工具 Xilinx ISE(使用 Spartan3 這塊 FPGA 板)中進行邏輯合成、靜態時序分析以及佈局與繞線的動作，檢視其面積大小，這裡所使用的時脈週期為 50MHz。由設計工具回報的資料顯示，這部分的硬體使用了 5832 個邏輯閘。而表 4.1 分別顯示單獨設計錯誤症狀以及錯誤位置的硬體需要使用多少邏輯閘來完成，和我們把兩部份功能設計在一起所使用的邏輯閘做比較。從表中我們可以發現原本應該使用兩塊硬體的面積比我們將這兩部份設計成一塊硬體的面積要來的大，也就是我們這樣的設計可以節省 35%的邏輯閘個數。

表 3 三種硬體使用的邏輯閘個數

	Gate count
Syndrome	4909
Chien Search	3943
Co-syndrome-chien search	5832

確定所設計的硬體在功能上沒有問題後，就必須來跟軟體連結。這節一開始我們就估算了硬體需要花多少的時間才能將值算完，而當我們把軟、硬體整合後，我們也估算了用軟體部分計算的時間。如表 4。

表 4 里德所羅門解碼的各功能方塊所花時間表

	Ours	PAC	IBM
Syndrome	1,020	11,422	897,601
BM	59,830	10,696	59,830
Chien Search	576	9,971	505,743
Forney	40,774	6,573	40,774

說明一下此表，第一個就是此論文的主要核心，以軟體為基礎外掛硬體加速器的架構，第二個是工研院自行研發的 DSP，第三個是單純用軟體來實現里德所羅門解碼，可以看出尋找錯誤症狀和尋找錯誤位置佔了大部份的時間，而當我們將這兩部份使用硬體加速器實現，才可以大大的減少運算時間。

第五章　結論與未來發展

由於使用軟體做某些運算會花許多時間去處理，而讓系統的大部份的時間都浪費在這些運算，大大的降低了系統整體的效能。像是本篇論文主要討論的里德所羅門解碼用到的伽羅瓦場乘法，必須先建立兩個表，當兩數要做伽羅瓦場乘法時，則要先去查表轉換成指數形式後，做相加動作，這樣才完成一個伽羅瓦場的乘法；而當是做伽羅瓦場的乘累加運算，要先查表轉換成指數形式，做相加動作，接著還要轉回原本的形式，然後做 XOR，這樣才完成一個乘累加的動作。光是計算一次乘累加就要來來回回轉換，浪費許多運算時間，而在里德所羅門解碼的演算法中，尋找錯誤症狀和尋找錯誤位置就是一直在重複做乘累加，假如是 RS(255,239)這組里德所羅門碼，則尋找錯誤症狀及錯誤位置就各有 255 次乘累加需要運算，所以使用軟體來處理會把時間都花在這兩部份。

我們在本篇論文提出一個以軟體為基礎外掛硬體加速器架構，主要就是要解決上述的那些問題，我們將尋找錯誤症狀及尋找錯誤位置這兩部份用硬體方式實現，因為硬體有專門處理伽羅瓦場乘法，只需要在一個時脈內就可以完成，因此大大的提升了處理這兩部份的效能。又因為這兩部份的演算法很類似，只需要對硬體架構做些改變，就可以將兩部份的硬體合併成一套，減少硬體的使用。而我們又搭配了摺疊演算法，因此使得硬體有可重複使用的特性，也又減少了硬體的使用。

在未來發展方面，在里德所羅門解碼中除了錯誤（Error）之外，另外還有一種類似錯誤的消除（Erasure）[1]，其與錯誤之間的不同在於錯誤是在接收到輸入之字碼（codeword）時，並不知道其所發生之位置與大小；但消除則是在接收到的字碼中，會有錯誤位置的標記，但仍然無法得知其大小。消除（Erasure）此種類似錯誤旗標的信號，產生的原因主要來自於錯誤控制編解碼（ECC）的前一級解調變（Demodulation）時所產生。因為其可能會發現解調變出來的信號介於兩種碼之間，能夠猜出這個位置應該有發生錯誤，但並不知道錯誤大小，於是

便先在這個位置上放一個旗標，表示此位置可能有錯，於是就留到里德所羅門解碼部分來處理。許多介紹里德所羅門解碼的書中[1] [2]，皆可找到下面關於錯誤與消除和除錯能力之間的關係：

$$2y + x \leq d - 1 = 2t \qquad (5.1)$$

$$y : total\ \ error\ \ number$$

$$x : total\ \ erasure\ \ number$$

$$t : capability\ \ of\ \ correction$$

但發現假如有消除此種錯誤旗標加入，勢必會使得原先對錯誤（Error）更正的能力降低，因為其能力必須分到消除（Erasure）信號上。

參考文獻：

[1] Irving S. Reed and Xuemin Chen, Error-Control Coding for Data Networks, Kluwer Academic Publisher published, 1999.

[2] Stephen B. Wicker, Error Control Systems for Digital Communication and Storage, Prentice-Hall, Inc. published, 1995.

[3] R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Polynomial GCD Computation," Tech. Rep., Carnegie-Mellon University, Computer Science Department, May 1982.

[4] Keiichi Iwamura, Yasunori Dohi, and Hideki Imai, "A Design of Reed-Solomon Decoder with Systolic-Array Structure," IEEE Transactions on Computers, Vol. 44, No. 1, pp. 118-122, January 1995.

[5] PowerPC 405 Processor Reference Guide
http://www.xilinx.com/bvdocs/userguides/ppc_ref_guide.pdf

[6] PowerPC 405 Processor Block Reference Guide
http://www.xilinx.com/bvdocs/userguides/ug018.pdf

[7] EDK OS and Libraries Reference Guide
http://www.xilinx.com/ise/embedded/edk6_3docs/oslibs_rm.pdf

[8] Processor IP Reference Manual
http://www.xilinx.com/ise/embedded/proc_ip_guide.pdf

[9] Keshab K. Parhi, VLSI Digital Signal Processing Systems: Design and Implementation, Wiley-Interscience, Inc. published, 1998.

[10] H. M. Shao, T. K. Truong, L. J. Deutsch, J. H. Yuen, and I. S. Reed,"A VLSI design of a pipeline Reed-Solomon decoder," in *Proc. IEEEInt. Conf. ICASSP*, 1985, pp. 1404–1407.

[11] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for polynomial GCD Computations," Dep. Comput. Sci., Carpegie-Mellon Ulgiv., Pittsburgh,PA,

Rep.， 1982.

[12] D.V. Sarwate and N. R. Shanbhag, “High-Speed Architectures for Reed-Solomon Decoders,” IEEE Transactions on Very Large ScaleIntegration Systems, Vol. 9, No. 5, pp. 641-655, October 2001.

[13] H-C Chang, C. B. Shung, and C-Y Lee, “ A Reed-Solomon Product-Code Decoder Chip for DVD Applications,” IEEE Journal ofSolid-State Circuits, Vol. 36, No. 2, pp. 229-238, February 2001.

[14] 陳玉書，「可重複規劃之里德所羅門解碼器設計」，國立交通大學，碩士論文，2004。

[15] Y. X. You, J. X.Wang, F. C. Lai, and T. Z. Ye, “Design and implementation of high-speed Reed-Solomon decoder,” in *Proc. IEEE Int. Conf. Circuits Syst. Commun. (ICCSC)*, 2002, pp. 146–149.

[16] H. H. Lee, M. L. Yu, and L. Song, “VLSI design of Reed-Solomon decoder architectures,” in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS' 2000)*, 2000, pp. 705–708.