

國立交通大學
電機與控制工程研究所
碩士論文

可重複規劃之里德所羅門解碼器設計
Design on Reconfigurable Reed-Solomon Decoder



研究生：陳玉書
指導教授：董蘭榮 博士

中華民國九十三年七月

可重複規劃之里德所羅門解碼器設計

Design on Reconfigurable Reed-Solomon Decoder

研究生：陳玉書

Student：Yu-Shu Chen

指導教授：董蘭榮

Advisor：Lan-Rong Dung

國立交通大學
電機與控制工程學系
碩士論文



Submitted to Department of Computer and Information Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Electrical and Control Engineering

July 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年七月

可重複規劃之里德所羅門解碼器設計

學生：陳玉書

指導教授：董蘭榮 博士

國立交通大學電機與控制工程研究所

摘 要

本篇論文提出了一套里德所羅門解碼器硬體的摺疊架構，其所採用的摺疊演算法根據處理速率（Throughput）條件的評估，在架構中使用少量的運算器，規劃其運作時序，就可分時完成工作，如此一來可以減少硬體中乘加運算器的數目，縮小硬體的面積。摺疊演算法非常適合於處理速率不需要很高的陣列硬體架構，尤其是對於陣列的運算單元個數會因規格的不同而有數目上變化的應用。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除錯能力規格來改變，除錯能力越大，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需採用的運算單元個數，摺疊後的架構對於除錯能力較小的規格，只需要減短運算時序的重複週期即可，這樣的架構展現了硬體可重複使用的特性，而且摺疊後的硬體架構有相當高的硬體使用率，在各種除錯能力規格下都很平均。此外，在處理速率規格符合的前提下，若要提升最大除錯能力，加長陣列長度，只需要增加儲存運算結果的記憶容量大小，並延長重複週期就可以達成，這也使得摺疊後的硬體更具有擴充性。

Design on Reconfigurable Reed-Solomon Decoder

Student : Yu-Shu Chen Advisor : Dr. Lan-Rong Dung

Department of Electrical and Control Engineering
National Chiao Tung University

ABSTRACT

This thesis presents a folding approach for reconfigurable Reed-Solomon decoder. 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. For xDSL applications, the computation requirement of Reed-Solomon decoder is varying with the error-correcting capability t and, thus, a flexible reconfigurable architecture becomes very attractive. Our approach allows the Reed-Solomon decoder to be configured by t without slacking processing elements.

誌 謝

兩年的研究生涯，將以這篇碩士論文作為句點，論文中所留下的一字一句，不僅記錄了這兩年研究的成果，也蘊藏了這幾年同窗好友的友誼。

很榮幸能夠在董蘭榮老師門下學習，其不厭其煩地指導與教誨，讓我在這條研究的道路上，有非常明確的方向。此外，還有陳紹基教授、吳安宇教授、吳文榕教授的撥冗指導，給予許多寶貴的意見，讓這篇論文可以更為完整齊備，這一點一滴都讓我感恩在心。

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。

當然還要奉上最深的感謝給我的父母親，你們給予我的優裕環境，以及全心全力的照顧，讓我在求學的過程中得以順遂的向前邁進，這篇論文的誕生，表現了碩士研究生涯的成果，在其中所呈現的進步與成長，以及蘊涵的欣喜與榮耀，將與你們一同分享。

文末，僅以此篇論文獻給所有關心愛護我的人，再次獻上我衷心的感激，謝謝大家！

玉書 謹誌

國立交通大學 系統晶片實驗室

民國九十三年七月

目 錄

中文摘要	i
英文摘要	ii
誌謝	iii
目錄	iv
表目錄	vii
圖目錄	viii
第一章	緒論	1
1.1	研究動機	1
1.2	論文摘要	4
第二章	研究背景	5
2.1	伽羅瓦場 (Galois Field) 定義介紹	5
2.1.1	本質多項式與伽羅瓦場的建立	6
2.1.2	伽羅瓦場的乘加運算	7
2.2	里德所羅門 (Reed-Solomon) 碼定義介紹	8
2.3	里德所羅門編碼 (Encoding) 演算法	9
2.4	里德所羅門解碼 (Decoding) 演算法	10
2.4.1	收到信號的錯誤症狀 (Syndrome) 計算	13
2.4.2	尋找錯誤位置多項式之方法	14
2.4.2.1	Berlekamp-Massey 疊代演算法	14
2.4.2.2	最高公因式演算法 (Euclidean Algorithm)	17
2.4.3	尋找錯誤位置之方法 (Chien Search)	19
2.4.4	尋找錯誤大小的佛尼 (Forney) 演算法	19

第三章	摺疊演算法之推演與建立	21
3.1	摺疊演算法的硬體架構推導	22
3.2	以摺疊演算法實現硬體之方法與特色	27
第四章	硬體實現架構	29
4.1	整合里德所羅門解碼器之矽智產合成器的運作流程與完整 電路架構	30
4.2	伽羅瓦場乘法器	33
4.3	處理錯誤症狀 (Syndrome) 之硬體摺疊架構	36
4.3.1	乘加運算器及暫存器規劃	37
4.3.2	位址產生器的設計	41
4.3.3	錯誤症狀計算硬體之輸入輸出定義	42
4.4	信號症狀計算器與 BM 方塊之資料傳遞介面	43
4.5	尋找錯誤位置 (Error-Locator) 多項式以及錯誤大小評估 (Error-Evaluator) 多項式之硬體摺疊架構	44
4.5.1	RiBM 疊代演算法	44
4.5.2	乘加運算器及暫存器規劃	47
4.5.3	位址產生器的設計	51
4.5.4	RiBM 硬體之輸入輸出定義	52
4.6	BM 方塊與多項式估算器之資料傳遞介面	53
4.7	錯誤修正器 (Error Corrector) 硬體架構	54
4.7.1	多項式估算之硬體摺疊架構	55
4.7.1.1	乘加運算器及暫存器規劃	56
4.7.1.2	位址產生器的設計	58
4.7.1.3	多項式估算硬體之輸入輸出定義	59
4.7.2	伽羅瓦場元素輸出表	60
4.7.3	佛尼演算法	63

4.7.4	錯誤修正器之輸入輸出定義	64
4.8	里德所羅門解碼器之狀態機設計	65
4.9	實現結果	68
第五章	結論	75
5.1	主要貢獻	75
5.2	未來展望	76
參考文獻	78
附錄	80



表 目 錄

【表 2.1】	利用本質多項式 $h(x)=1+x^2+x^3+x^4+x^8$ 建立之 $GF(2^8)$	7
【表 4.1】	里德所羅門解碼器組成方塊之邏輯單元數	72

圖 目 錄

【圖 2.1】	非二進位環狀碼(Cyclic code)的編碼(Encoding)電路 ...	10
【圖 2.2】	里德所羅門解碼流程圖	12
【圖 2.3】	錯誤症狀運算電路	13
【圖 2.4】	LFSR 架構迴圈形式	15
【圖 3.1】	基本運作單元概略圖	23
【圖 3.2】	運作單元陣列	23
【圖 3.3】	運作單元陣列之時序規劃	24
【圖 3.4】	運作單元陣列之摺疊架構	25
【圖 3.5】	摺疊架構之乘加運算器與暫存器規劃	26
【圖 3.6】	摺疊架構之暫存器定址	27
【圖 4.1】	里德所羅門解碼器硬體概略圖	31
【圖 4.2】	里德所羅門合成器之處理流程	32
【圖 4.3】	(a) 平衡的互斥樹狀架構 (b) 非平衡的互斥樹狀架構.....	3
【圖 4.4】	建立在 $GF(2^4)$ 之非規則全平行乘法器硬體架構	35
【圖 4.5】	錯誤症狀計算之陣列硬體架構	37
【圖 4.6】	除錯能力 $t=8$ 之錯誤症狀計算陣列	38
【圖 4.7】	錯誤症狀計算陣列之時序規劃	39
【圖 4.8】	錯誤症狀計算陣列之摺疊架構	39

【圖 4.9】	錯誤症狀計算硬體之乘加運算器與暫存器規劃	40
【圖 4.10】	錯誤症狀計算硬體之位址產生器	42
【圖 4.11】	錯誤症狀計算硬體之輸入輸出定義	42
【圖 4.12】	S2B 方塊定義	43
【圖 4.13】	RiBM 演算法之基本單元	46
【圖 4.14】	RiBM 疊代演算法硬體架構圖	46
【圖 4.15】	除錯能力 $t=8$ 之 RiBM 硬體陣列	47
【圖 4.16】	RiBM 硬體陣列之時序規劃	48
【圖 4.17】	RiBM 硬體陣列之摺疊架構	49
【圖 4.18】	RiBM 硬體之乘加運算器與暫存器規劃	50
【圖 4.19】	RiBM 控制單元方塊圖	50
【圖 4.20】	RiBM 硬體之位址產生器	51
【圖 4.21】	RiBM 硬體之輸入輸出定義	52
【圖 4.22】	B2P 方塊定義	53
【圖 4.23】	錯誤修正器之硬體架構	54
【圖 4.24】	八次多項式估算器之硬體架構	55
【圖 4.25】	八次多項式估算器陣列之時序規劃	56
【圖 4.26】	八次多項式估算器之摺疊架構	57
【圖 4.27】	多項式估算器硬體之乘加運算器與暫存器規劃	58
【圖 4.28】	多項式估算器硬體之位址產生器	59
【圖 4.29】	多項式估算器硬體之輸入輸出定義	60
【圖 4.30】	伽羅瓦場元素輸出表之硬體架構	62
【圖 4.31】	伽羅瓦場元素輸出表之位址產生器	62
【圖 4.32】	錯誤修正器之硬體架構	64
【圖 4.33】	錯誤修正器之輸入輸出定義	65
【圖 4.34】	里德所羅門解碼器之狀態機	67
【圖 4.35】	錯誤症狀計算器之 RTL 模擬	69

【圖 4.36】 RiBM 方塊之 RTL 模擬	70
【圖 4.37】 錯誤修正器之 RTL 模擬	70
【圖 4.38】 里德所羅門解碼器組成方塊之邏輯單元數比例分佈	71
【圖 4.39】 模擬測試驗證環境	73
【圖 4.40】 錯誤症狀計算之運算器使用率與除錯能力值 t 關係圖 ..	74



第一章

緒論

1.1 研究動機



在這個工商業發達的社會，科技生活的來臨，拉近了人與人之間的距離，而網路通訊也成為人們互相聯絡最普遍、最主要的方式，不僅是在音訊方面的聽覺交流，甚至在視訊方面的傳輸也漸漸廣泛的被大眾所採用，在這個繁忙的商業社會中，的確是一個促成了天涯若比鄰的好方法。

傳輸音訊或視訊除了要求頻寬之外，也必須注重傳輸的正確性，若正確性無法達到一定的水準，會使得這些訊息產生相當程度的失真，造成使用者的不便。所以提升訊息錯誤更正的能力，解決信號傳輸時所受到的干擾與雜訊，便成為如今網路傳輸的重要課題。

在現今普遍使用的非對稱數位用戶線路（ADSL），或是下一代的超高速數位用戶線路（VDSL）應用中，錯誤更正編解碼（ECC）所

採用的是里德所羅門碼 (Reed-Solomon Code)，主要原因是看上此種編解碼方式在所有的錯誤更正編解碼中，算是最嚴謹且更正能力最強的一種，尤其是對於叢集錯誤 (Burst Error) 或是隨機錯誤 (Random Error) 的修正能力上，更是讓其他編解碼方式望塵莫及，也因為這個重要的因素，使得里德所羅門碼被應用在許多的傳輸系統上，例如：無線通訊系統、纜線數據機、電腦記憶體…等。

在非對稱數位用戶線路 (ADSL) 與超高速數位用戶線路 (VDSL) 中，錯誤更正的里德所羅門解碼方塊是建立在有限伽羅瓦場 $GF(2^8)$ 上，為了因應各種所需之規格，必須做到可程式化 (Programmable)，致使可以任意調整變換錯誤更正能力，其錯誤更正能力最大必須到達 8 個字碼 (Codeword)。

2002 年 M. K. Song [12] 提出一種錯誤更正碼字元長度可變的里德所羅門解碼器硬體，專門應用在超高速數位用戶線路上。一般來說，在超高速數位用戶線路中，資料的傳遞皆由最高字元開始至最低字元，但 M. K. Song 所提出硬體中，用來尋找錯誤位置與大小的部份，由於要配合超高速數位用戶線路的要求，必須做到容易程式化，故其硬體架構會出現輸入是由最高字元到最低字元，但輸出卻是反相，由最低字元到最高字元，為了這樣子的改變，必須多加了一個字元位置前後顛倒的硬體，其結果才可以再被下一級所採用，但是要完成這個動作，必須等到所有的字元都已經準備就緒，才能一次把所有位元的位置調換過來，故字元位置前後顛倒的動作，似乎會把整個解碼的時間拖垮。另外，這個硬體在錯誤更正能力不需要很高的情形下，每個乘加器運算依然比照錯誤更正能力最大的方式動作，在功率上也造成了不必要的消耗。

為了改善這些問題，在本篇論文中提出一個新的硬體架構，不僅可以達到程式化的目的，並可以大幅降低功率上不必要的損失。除此之外，根據處理速率（Throughput）條件的評估，更可以減少硬體中乘加運算器的數目，縮小硬體的面積，這對於非對稱數位用戶線路（ADSL）與超高速數位用戶線路（VDSL）等需要可程式化的里德所羅門解碼器的應用，有一定程度的助益。

本篇論文所設計的里德所羅門解碼器特色如下：

- 解碼器的整個解碼流程分為三級：
 - 錯誤症狀計算（Syndrome Calculation）
 - 求出錯誤位置及大小評估多項式（Berlekmap-Massey）
 - 錯誤消除（Forney Algorithm、Correction）其中主要貢獻在於將論文中所發展的摺疊演算法套用於解碼流程的這三級中，摺疊後架構所產生的可重複使用性，對於不同規格之除錯能力 t ，可以動態的調整規劃，其概念適合應用在超高速數位用戶線路這類型的動態規格調整之裝置。

- 使用者可從軟體 C 語言中輸入所需之最大除錯能力參數 t ，以及錯誤症狀計算、求出錯誤位置及大小評估多項式、多項式估算這三種硬體架構之使用運算單元個數，即可得到 VHDL 硬體描述語言，其為可合成（Synthesizable）之 RTL 程式。

- 設計之解碼器所能接受之規格：
 - 伽羅瓦場階數 m 為 8
 - 錯誤更正能力 t 可動態設定

1.2 論文摘要

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，並點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

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

先對整個硬體架構及合成器的運作與操作流程作介紹，之後再對解碼器中的個別方塊加以詳細解說，並推導其摺疊之硬體架構。最後則是使用合成器依據所設定的運算單元個數所產生出來的硬體語言，送入相關合成軟體中，獲得相關面積等資訊，並統計解碼器組成方塊間相對關係之面積比例。

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。

[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. $x \bmod h(x)$)，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 α 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 $\{1, \alpha, \alpha^2, \dots, \alpha^{2^n-2}\}$ 。表 2.1 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。

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}

表 2.1 利用本質多項式 $h(x) = 1 + x^2 + x^3 + x^4 + x^8$ 建立之 $GF(2^8)$

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 $t \leq 2^m - 1$ ，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：

$$n = 2^m - 1 \quad (2.1)$$

$$n - k = 2t \quad (2.2)$$

$$d_{\min} - 1 = 2t = n - k \quad (2.3)$$

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 $g_i \in GF(2^m)$ (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} 、 \cdots 、 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：

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

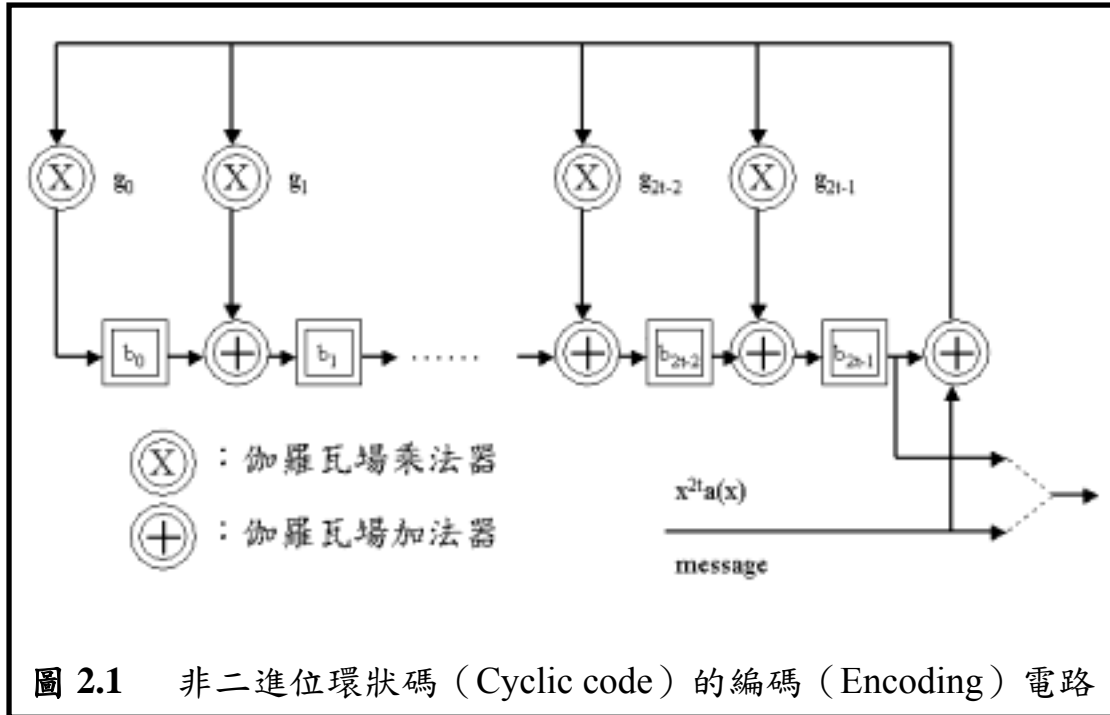
然而這其中的 k 就是訊息長度，並且根據前小節里德所羅門碼的定義： $k = n - 2t$ ，即原先的訊息長度 k 等於編碼後長度 n 扣掉兩倍的除錯能力 t 。

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 $b(x) = b_0 + b_1x + \cdots + b_{2t-1}x^{2t-1}$ (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 α 的次方 α^{m_0} 、 α^{m_0+1} 、 \cdots 、 α^{m_0+2t-1} 也均為 $c(x)$ 的根。編碼電路的電路圖如圖 2.1 所示[2]。



2.4 里德所羅門 (Reed-Solomon) 解碼演算法

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：

$$c(x) = c_0 + c_1x + \dots + c_{n-1}x^{n-1}, c_i \in GF(2^m) \quad (2.7)$$

$$r(x) = r_0 + r_1x + \dots + r_{n-1}x^{n-1}, r_i \in GF(2^m) \quad (2.8)$$

故因為傳輸而產生的雜訊就可以表示為：

$$e(x) = r(x) - c(x) = e_0 + e_1x + e_2x^2 + \dots + e_{n-1}x^{n-1} \quad (2.9)$$

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：

$$\text{Error Values } Y_l = e_{j_l} \quad \text{for } l = 1, 2, \dots, v \quad (2.10)$$

$$\text{Error Locators } X_l = \alpha^{j_l} \quad \text{for } l = 1, 2, \dots, v \quad (2.11)$$

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：

$$\begin{aligned} \Lambda(x) &= (1 - X_1x)(1 - X_2x) \cdots (1 - X_vx) \\ &= \prod_{i=1}^v (1 - X_i x) \\ &= \Lambda_0 + \Lambda_1 x + \Lambda_2 x^2 + \cdots + \Lambda_v x^v \end{aligned} \quad (2.12)$$

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：

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_{t-1} & S_t \\ S_2 & S_3 & \cdots & S_t & S_{t+1} \\ \vdots & & & \vdots & \vdots \\ \vdots & & & \vdots & \vdots \\ S_t & S_{t+1} & \cdots & S_{2t-2} & S_{2t-1} \end{bmatrix} \begin{bmatrix} \Lambda_t \\ \Lambda_{t-1} \\ \vdots \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ \vdots \\ -S_{2t} \end{bmatrix} \quad (2.13)$$

$$\begin{bmatrix} X_1 & X_2 & \cdots & X_{v-1} & X_v \\ X_1^2 & X_2^2 & \cdots & X_{v-1}^2 & X_v^2 \\ \vdots & & & \vdots & \vdots \\ \vdots & & & \vdots & \vdots \\ X_1^v & X_2^v & \cdots & X_{v-1}^v & X_v^v \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ \vdots \\ Y_v \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ \vdots \\ S_v \end{bmatrix} \quad (2.14)$$

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_1)之後，便可以利用方程式 (2.14) 找出錯誤的大小(Y_1)，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 2.2 所示。

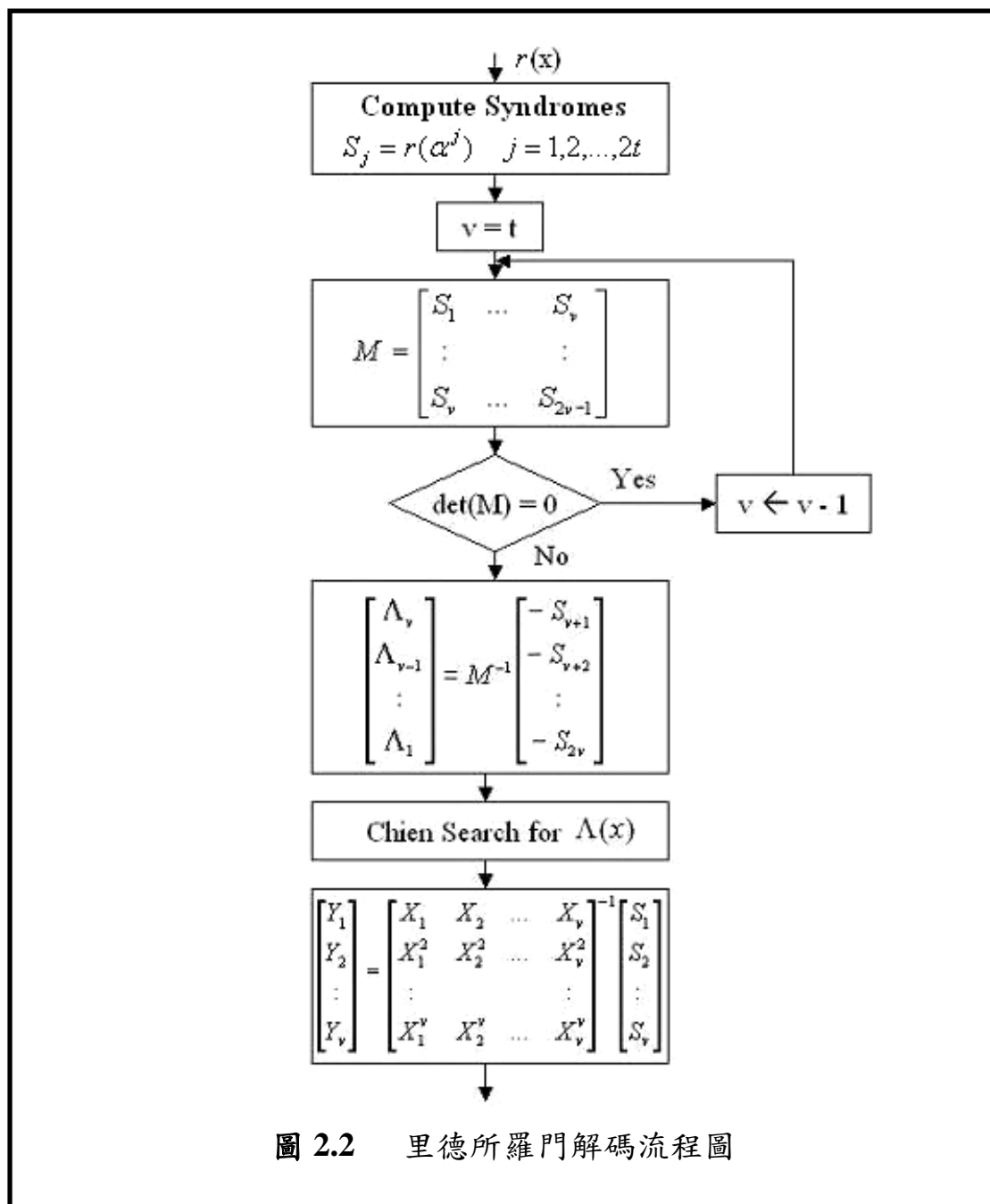


圖 2.2 里德所羅門解碼流程圖

2.4.1 收到信號的錯誤症狀 (Syndrome) 計算

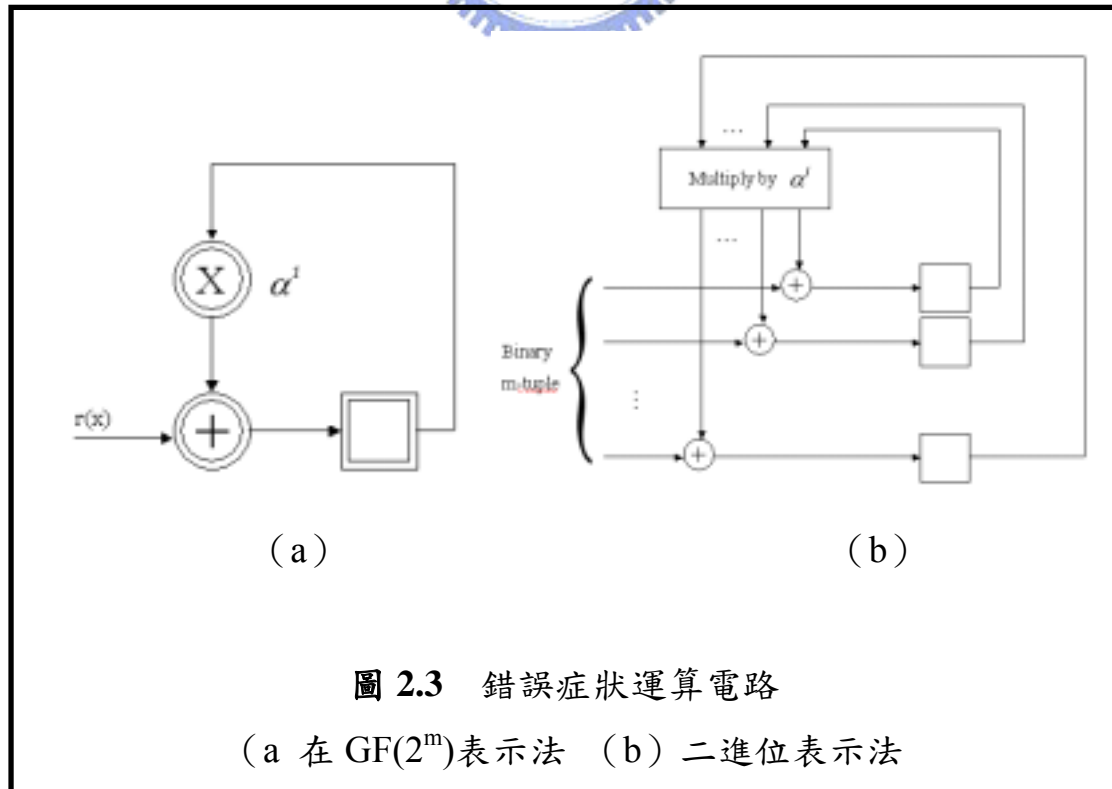
在解碼的過程中，錯誤症狀的計算是所要進行的第一步驟。由前面的章節得知，接收到的信號可表示為 $r(x) = r_0 + r_1x + \dots + r_{n-1}x^{n-1}$ ，從 2.2 小節得知字碼產生多項式 (Codeword Generator Polynomial) 的根為 $\alpha^{m_0} \sim \alpha^{m_0+2t-1}$ ，其中 t 代表除錯的能力，又因為 $c(x) = m(x)g(x)$ ，所以便可以得到以下的關係：

$$r(\alpha^{m_0+i}) = c(\alpha^{m_0+i}) + e(\alpha^{m_0+i}) = e(\alpha^{m_0+i}) = \sum_{j=0}^{n-1} e_j \alpha^{(m_0+i)j}, \quad i = 0, 2, \dots, 2t-1 \quad (2.15)$$

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：

$$S(x) = \sum_{j=0}^{2t-1} S_j \cdot x^j \quad \text{where} \quad S_j = \sum_{i=1}^n r_{n-i} \cdot (\alpha^{m_0+j})^{n-i} \quad (2.16)$$

整個過程就如同乘累加運算一樣，運算電路如圖 2.3 所示。



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

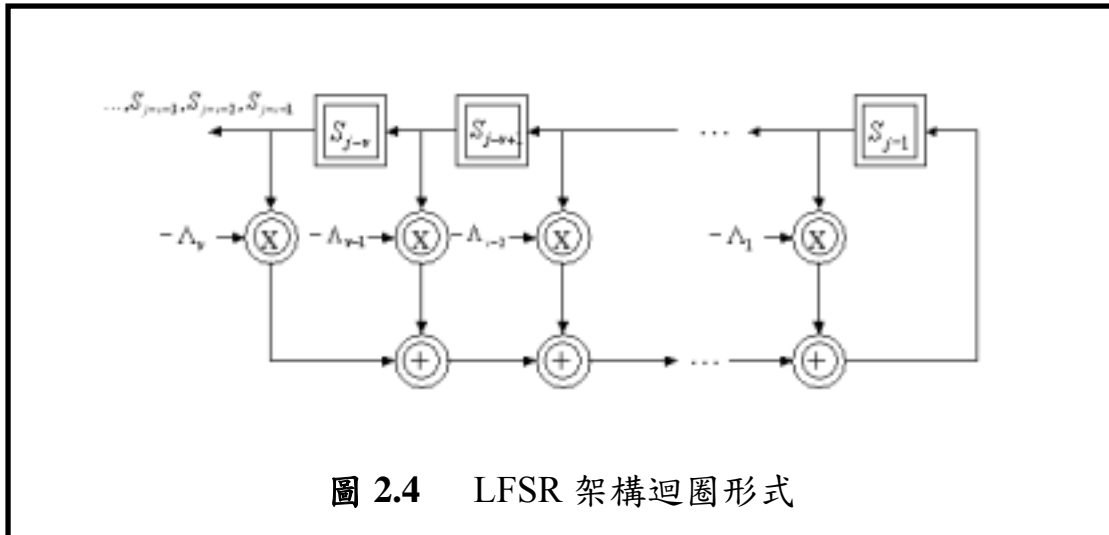
西元 1967 年 E. Berlekamp 提出一種極有效率之演算法，同時適合使用於解碼 BCH code 及 RS code，再加上後來 Massey 進一步的改良，其複雜度僅隨著 t^2 成長[1]，所以是一個求取錯誤位置多項式比較有效率的演算法，因此適宜用來處理較大除錯能力的里德所羅門解碼架構。

根據 2.4 節方程式 (2.13)，可以把其表示成此種迴圈形式：

$$S_j = -\sum_{i=1}^v \Lambda_i S_{j-i} = -(\Lambda_v S_{j-v} + \Lambda_{v-1} S_{j-v+1} + \cdots + \Lambda_1 S_{j-1}),$$

for $j = v+1, v+2, \dots, 2t$ (2.17)

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 2.4 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。



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 $\Lambda^{(k)}(x) = \Lambda_k x^k + \Lambda_{k-1} x^{k-1} + \dots + \Lambda_1 x + 1$ 成為一個最短長度為 k 的连接多項式 (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连接多項式；相反的，假如兩者中間沒有差異，就繼續拿此连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，直到此线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]：

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

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。

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 $S(x) = S_0 + S_1x + \dots + S_{2t-1}x^{2t-1}$ 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 $\Lambda(x) = \Lambda_0 + \Lambda_1x + \dots + \Lambda_t x^t$ ，則錯誤大小評估多項式可得知定義如下：

$$\omega(x) = \omega_0 + \omega_1x + \dots + \omega_{t-1}x^{t-1} \quad (2.18)$$

$$\boxed{\Lambda(x)S(x) \equiv \omega(x) \pmod{x^{2t}}} \quad (2.19)$$

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：

$$\begin{aligned}r_1(x) &= a(x) - q_1(x) \cdot b(x) \\r_2(x) &= b(x) - q_2(x) \cdot r_1(x) \\r_3(x) &= r_1(x) - q_3(x) \cdot r_2(x) \\&\vdots \\0 &= r_{n-1}(x) - q_{n+1}(x) \cdot r_n(x)\end{aligned}\tag{2.20}$$

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：

$$\begin{aligned}r_{-1}(x) &= a(x), \quad r_0 = b(x) \\q_{k+1}(x) &= \left\lfloor \frac{r_{k-1}(x)}{r_k(x)} \right\rfloor \\r_k(x) &= u_k(x)a(x) + v_k(x)b(x) \\u_{-1}(x) &= 1, \quad u_0(x) = 0 \\v_{-1}(x) &= 0, \quad v_0(x) = 1\end{aligned}\tag{2.21}$$

$$\begin{aligned}
r_{k+1}(x) &= u_{k+1}(x)a(x) + v_{k+1}(x)b(x) \\
&= r_{k-1}(x) - q_{k+1}(x)r_k(x) \\
&= (u_{k-1}(x)a(x) + v_{k-1}(x)b(x)) \\
&\quad - q_{k+1}(x)(u_k(x)a(x) + v_k(x)b(x)) \\
&= (u_{k-1}(x) - q_{k+1}(x)u_k(x))a(x) \\
&\quad + (v_{k-1}(x) - q_{k+1}(x)v_k(x))b(x)
\end{aligned} \tag{2.22}$$

因此，藉由這些定義及推導，可以統整出一些關係式：

$$r_{k+1}(x) = r_{k-1}(x) - q_{k+1}(x)r_k(x) \tag{2.23}$$

$$u_{k+1}(x) = u_{k-1}(x) - q_{k+1}(x)u_k(x) \tag{2.24}$$

$$v_{k+1}(x) = v_{k-1}(x) - q_{k+1}(x)v_k(x) \tag{2.25}$$

$$q_{k+1}(x) = \left\lfloor \frac{r_{k-1}(x)}{r_k(x)} \right\rfloor \tag{2.26}$$

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 $\omega(x) \equiv r_k(x) = \text{GCD}(a(x), b(x))$ ，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：

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

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

3. 遞迴方式從 $i = 1$ 到 $\deg(r_i(x)) < t$ ：

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

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

$$Q_{i-1}(x) = \left[\frac{r_{i-2}(x)}{r_{i-1}(x)} \right]$$

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]：

$$Y_i = \frac{X_i^{-(m_0-1)} \omega(X_i^{-1})}{X_i \prod_{j=1, j \neq i}^v (1 - X_j X_i^{-1})} = - \frac{X_i^{-(m_0-1)} \omega(X_i^{-1})}{\Lambda'(X_i^{-1})} = - \frac{x^{m_0} \omega(x)}{x \Lambda'(x)} \Big|_{x=X_i^{-1}} \quad (2.27)$$

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。

$$\begin{aligned}x\Lambda'(x) &= x(\Lambda_1 + 2\Lambda_2x + 3\Lambda_3x^2 + \dots) \\ &= x(\Lambda_1 + \Lambda_3x^2 + \dots) \\ &= \Lambda_1x + \Lambda_3x^3 + \dots\end{aligned}\tag{2.28}$$

所以 $x\Lambda'(x)$ 相當於 $\Lambda(x)$ 奇數次方項的總和，這個特性會影響到後面硬體架構的設計。



第三章

摺疊演算法之推演與建立

現今有許多的硬體設計為了追求簡單化以及規律性，大多採用運算單元陣列的形式，其中最著名的就是 Systolic 陣列。Systolic 陣列是由 Kung [7] 在 1980 年提出來的一種硬體架構，這種架構是由一些簡單的基本運作單元 (PE) 所連接組成，所以擁有規則且簡單的特性，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作效能以及維持很高的處理速率 (Throughput)。

1995 年 Keiichi Iwamura [8] 提出一篇以 Systolic 陣列架構為基礎的里德所羅門解碼硬體，其主要的特色就是只使用一種相同基本運作單元來實現解碼架構，且利用了管線化的技巧，除了讓整個硬體非常的簡單有規律，也可以高速的運作。

從另一個角度來看，倘若在整個系統中，里德所羅門解碼方塊的下一級並不需要很高的處理速率 (Throughput)，那麼這個高處理速

率的優點，勢必可以拿來交換一些其他的好處。摺疊硬體就是一個以降低處理速率而獲得硬體面積減少的一個方法，將一連串基本運作單元的陣列經過摺疊方法，使用少量的運算單元，分時完成原本的工作，雖然完成工作的速度變慢，但運算單元卻減少，對於不必要很高處理速率的系統，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覆使用性，增加了硬體的彈性。

3.1 摺疊演算法的硬體架構推導

對於硬體摺疊的好處，是讓人顯而易見的，但是對於摺疊方式的推導，卻沒有一個非常明確的流程讓人遵循，所以在這一個章節中，首要的目的就是介紹硬體摺疊的步驟，並將其稱為摺疊演算法，只要依照步驟實行，整個過程是相當有規劃的進行，且摺疊後的架構在時脈上也不至於出錯。

一般傳統運算單元陣列，對於訊號傳遞的路徑，可分為兩類：向前路徑（Forward Path）以及回授路徑（Feedback Path）。向前路徑上的訊號，在兩兩相鄰的運算單元之間互有關係，回授路徑上的訊號，則只有在本體運算單元中有影響。對於陣列中的每個基本運作單元，主要區分成乘加器（MAC）與暫存器（Register）兩部分，圖 3.1 即為一個普遍的基本運作單元，為了演算法上的推導方便，在這裡將乘加器和暫存器分開表示，往後在運算單元的表示上，也皆以乘加器和暫存器分開的方式呈現。

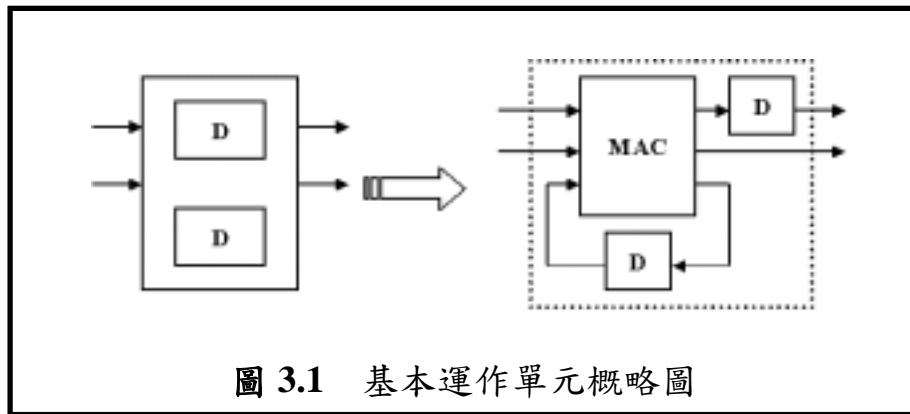


圖 3.1 基本運作單元概略圖

現在就以一個串接 12 個基本運作單元的陣列為例子，說明摺疊演算法的推導。一開始在摺疊之前，首先考慮所需要使用的運算單元個數。假設採用 4 個運算單元，於是可以將 12 個分為 3 列，每一列的運算單元必須完整的排列對齊，這樣的目的是打算將原本一個回合內完成的資料計算，分為 3 個的時間點來運作，每一列各分配到一個時間點進行。圖 3.2 所示就是運算單元的排列方式。

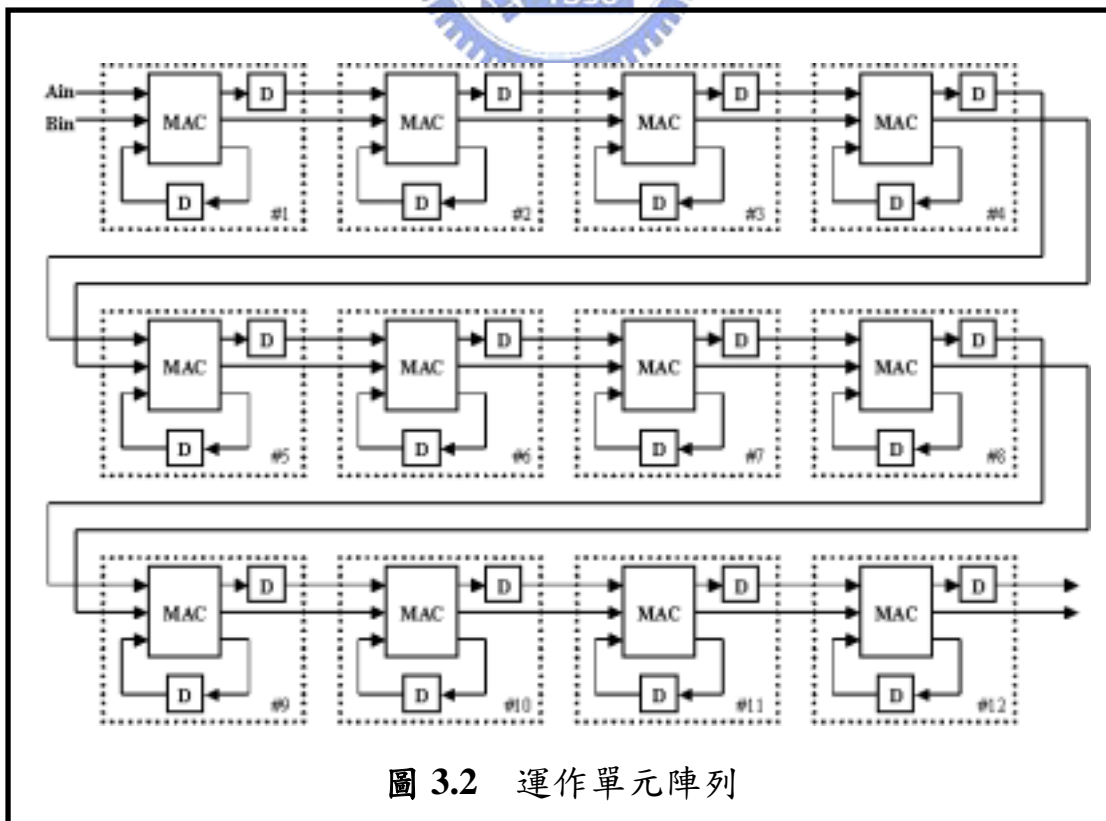


圖 3.2 運作單元陣列

接著，整體的架構開始作一些推導轉換。將每個暫存器變成 3 倍的大小，接著在列與列的每個向前路徑上加入資料銜接的暫存器，這麼一來，在每回合的第一個時間點時，Ain 及 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，同時，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的銜接暫存器中，而第二列及第三列的運算單元則沒有動作；到了第二個時間點，第二列的第一個乘加器就讀入銜接暫存器的值，第二列的運算單元開始運作，並且將結果存入該列各暫存器的第二個位置，其餘兩列的運算單元均不動作，當然第四個向前路徑上暫存器的值也傳至下一個連接暫存器；到了第三個時間點，第三列的運算單元會開始計算並將結果存到各暫存器的第三個位置，第一、二列則無動作；之後進入下一個新的回合，如此一直反覆循環。圖 3.3 說明了上述的動作。

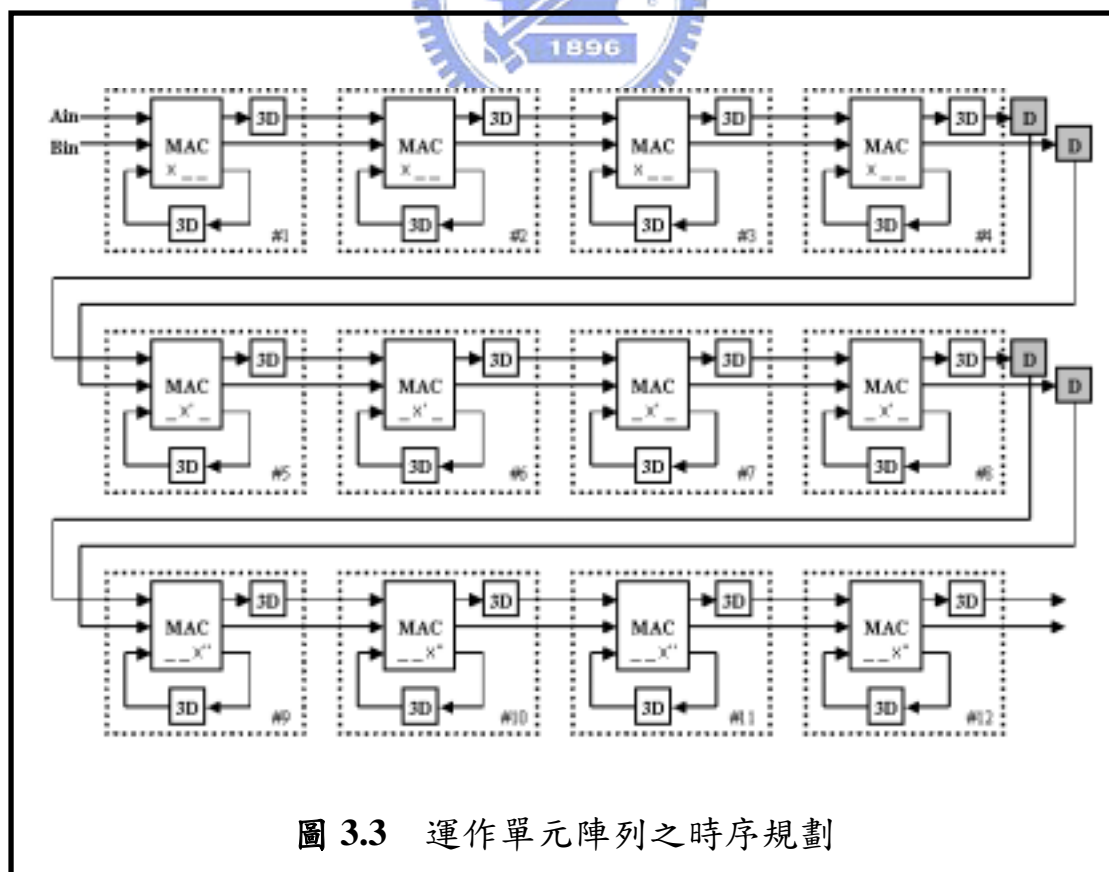
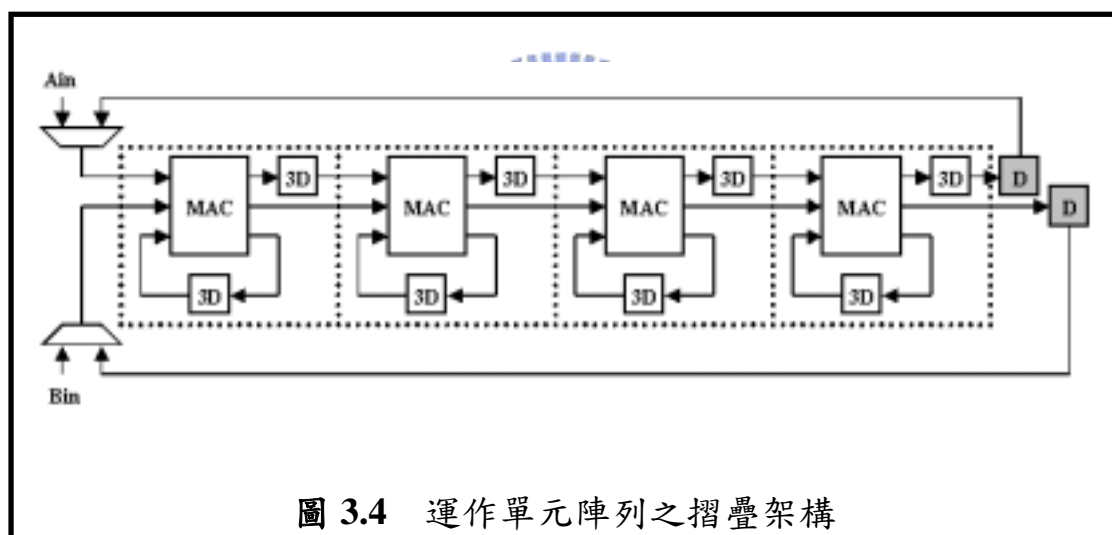


圖 3.3 運作單元陣列之時序規劃

由此可知，這三列中的第一個乘加器，在每回合的三個動作時間點中是不重疊的，也就是每列的第一個乘加器不會在同一個時間點同時動作，而每列的第二、三、四個乘加器亦然。由於在每列中相對位置的乘加器的工作時間點是兩兩互斥的，也就是工作的時機互相不衝突，所以可以將三列的運算單元壓縮成一列，成為圖 3.4 所展示的。於是，只要在一個回合中，送入一個 Ain 及 Bin 資料，並且在這個回合裡，根據不同時間點，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，就可得到正確的運算。這個方法把整體的運算時間拉長了，但卻減少三分之二的乘加器數目，這就是目的所在。



原先 12 個運算單元的陣列，經過摺疊演算法調整後，可以明確的分出乘加器群 (MAC Group) 及暫存器檔案 (Register File)，要使整個運算流程正確的動作，只需要在每個時刻調整暫存器檔案所需輸出及儲存的資料，以及正確的切換在最前端的多工器 (MUX)，使其在 Addr = 00 時，由外部 Ain 及 Bin 輸入，其餘時刻則由銜接暫存器輸入，就可以完成工作，至於乘加器群只要不停的運算就可以了。圖 3.5 是經過摺疊演算法所產生的架構圖。

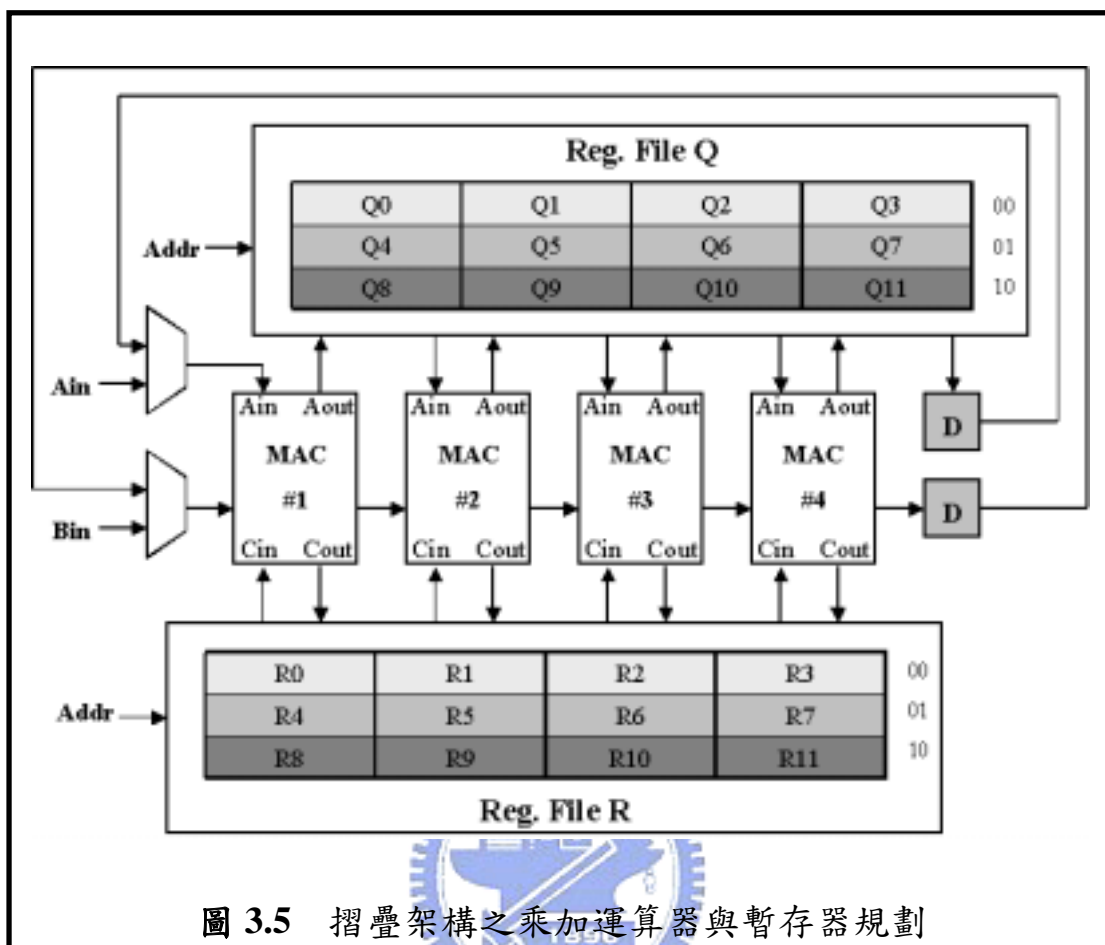


圖 3.5 摺疊架構之乘加運算器與暫存器規劃

值得一提的是，在暫存器檔案中每一行（Column）共有三個暫存器，就是先前推導過程中三倍暫存器（3D）大小所代表的意義。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必須在同一時刻送到乘加器群運算，因此將每一列的四個暫存器作捆綁，給予同一個暫存器定址名稱，往後只需要正確控制暫存器位址，架構就會正常運作，圖 3.6 即為暫存器捆綁及定址的示意圖。

摺疊演算法最主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。在下一節中主要說明套用摺疊演算法時運算單元個數的選擇，以及摺疊硬體架構的好處。

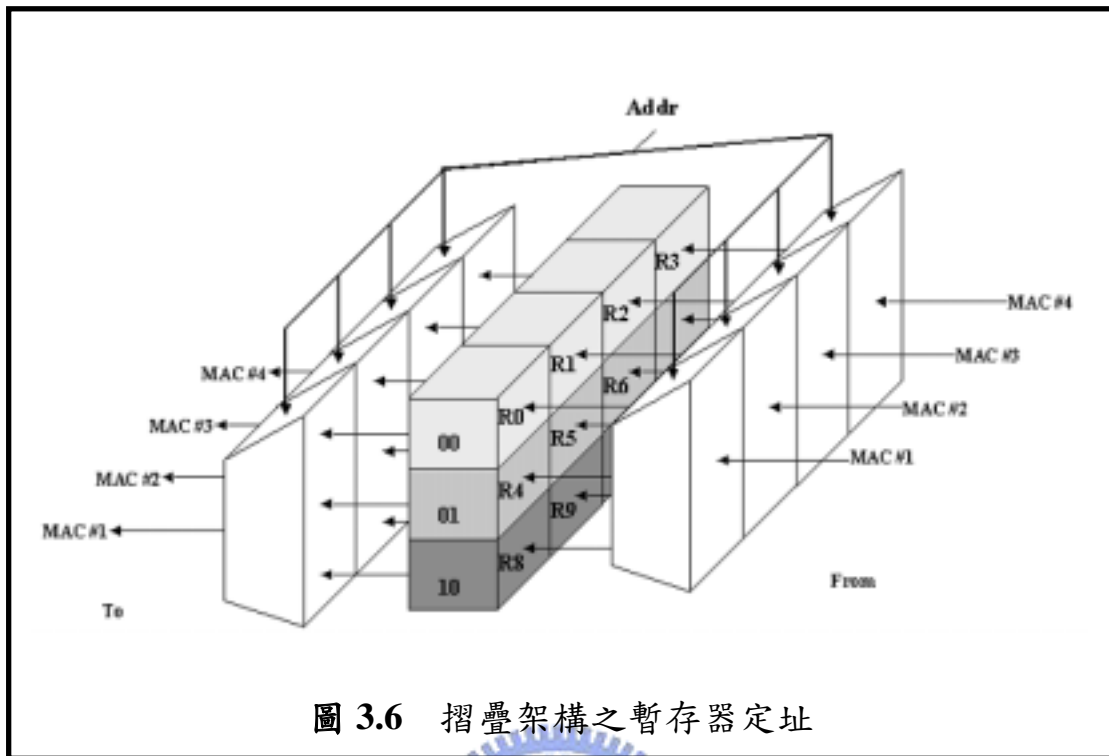


圖 3.6 摺疊架構之暫存器定址

3.2 以摺疊演算法實現硬體之方法與特色

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。假設現在系統中有一個方塊，其硬體架構由 N 個運算單元所組成，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：

$$IP = X_{put}^{-1} / t_{mac} \quad (3.1)$$

$$m = \lceil N / IP \rceil \quad (3.2)$$

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 $\lceil IP \rceil$ 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除錯能力規格來改變，除錯能力越大，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運算單元個數，摺疊後的架構對於除錯能力較小的規格，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，若要提升最大除錯能力，加長陣列長度，只需要擴充暫存器檔案大小，延長重複週期，這也使得摺疊後的硬體更具有彈性。



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，以及合成器的運作流程。

首先，在第一節中會先概括的提到整個里德所羅門的完整電路架構，以及最後所實現的矽智產合成器運作的流程。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。接下來幾個小節會分別介紹各級運算的摺疊硬體架構，從信號症狀、尋找錯誤位置多項式、錯誤大小多項式到最後所提出的錯誤修正器，都會一一作仔細的介紹。最後就是將合成器所產生的硬體語言交給軟體合成分析，並討論與統計其合成結果。

4.1 整合里德所羅門解碼器之矽智產合成器的運作流程與完整電路架構

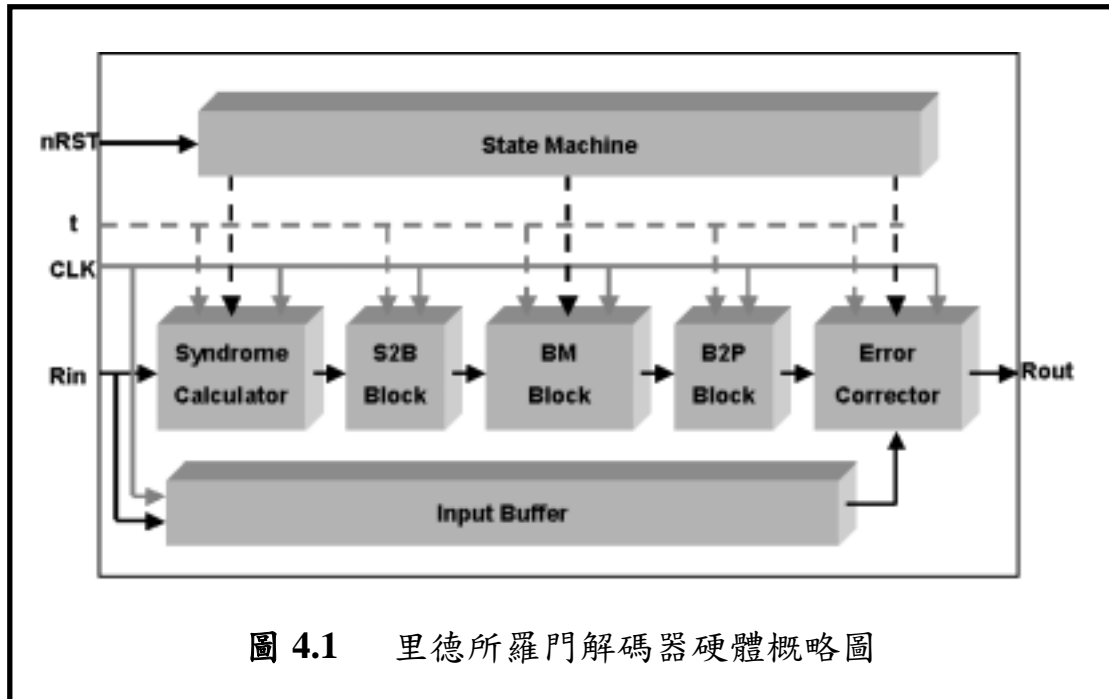
由第二章的里德所羅門解碼基本原理中，可以知道整個解碼過程是經由：

- (一) 信號症狀計算
- (二) 尋找錯誤位置多項式及錯誤大小評估多項式
- (三) 錯誤修正

故設計時將整個系統定義成七個區塊：

- (一) 信號症狀計算方塊 (Syndrome Calculator)
- (二) 信號症狀計算器與 BM 方塊之資料傳遞介面 (S2B Block)
- (三) BM (Berlekamp-Massey Algorithm) 方塊 (BM Block)
- (四) BM 方塊與多項式估算器之資料傳遞介面 (B2P Block)
- (五) 錯誤修正器 (Error Corrector)
- (六) 輸入信號緩衝器 (Input Buffer)
- (七) 狀態機 (State Machine)

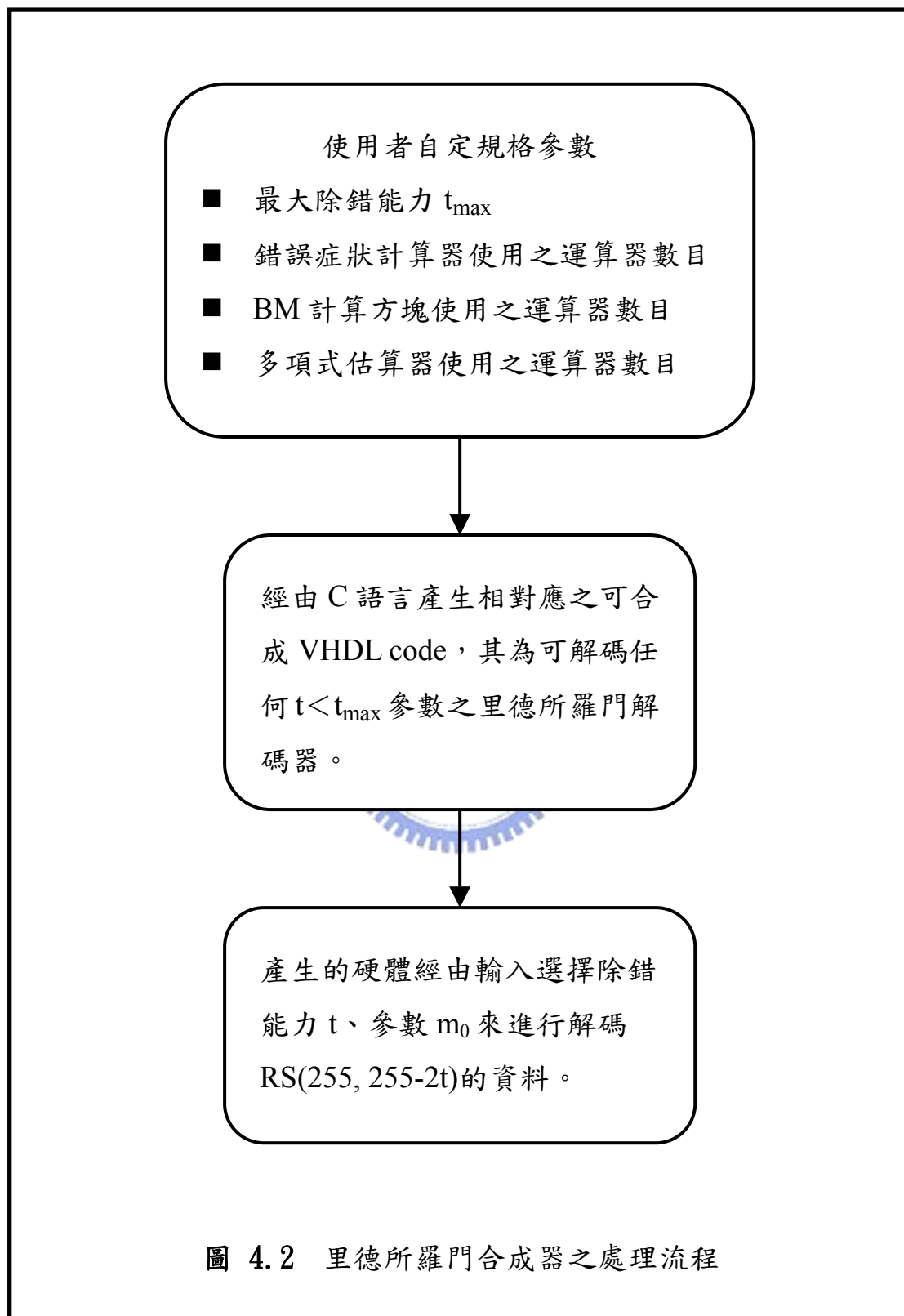
收到的信號必須先經由信號症狀計算方塊找出其所對應的症狀，接著再將症狀輸入 BM 方塊，找出相對應的錯誤位置多項式以及錯誤大小評估多項式。找出上述兩個多項式，就可以輸入錯誤修正器，這中間包括了多項式估計器和計算錯誤大小的佛尼 (Forney) 演算法實現，以及將錯誤從收到的信號更正回來的功能。當然，少不了要儲存接收信號的輸入信號緩衝器，以及協調整體運作的狀態機。另外還有 S2B 與 B2P 這兩個資料傳遞介面方塊。整體電路概略圖如圖 4.1 所示。



由於硬體使用摺疊演算法的緣故，使得整體架構變的更有彈性，可以動態調整不同規格的除錯能力 t 值，對於需要動態變換規格的裝置，有一定程度的助益。

在每一個運算方塊中，硬體的擴充性都很好，可以因應不同的規格作出有規律性的變化，所以便找出其規則性，設計出里德所羅門解碼之合成器。使用者可以輸入參數值，經由 C 語言的判斷及運算去產生符合其要求之可合成的 VHDL 硬體語言。

合成器有幾個參數是使用者必須自行設定的，在此作一些說明，需要設定的資料包括應用所需要最大的除錯能力值，以及錯誤症狀計算、BM 演算法、多項式估算三大主要計算方塊所需採用的運算單元個數。因此使用者只要設定以上所需的數值，合成器便會產生相對應之硬體摺疊架構，而且對於任何除錯能力值小於設定值的規格，皆可在此硬體架構上實現。此合成器的處理流程在圖 4.2 中統整說明。



4.2 伽羅瓦場乘法器

K. K. Parhi [3] 在 2001 年提出了一篇專門為了里德所羅門碼所設計之低功率且低延遲時間 (Latency) 之有限場乘法器。

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

$$A = \sum_{k=0}^{m-1} a_k \alpha^k, \quad B = \sum_{k=0}^{m-1} b_k \alpha^k \quad (4.1)$$

若 C 為 A 與 B 之乘積，則：

$$C = A \cdot B = \sum_{k=m}^{2m-2} d_k \alpha^k + \sum_{k=0}^{m-1} d_k \alpha^k \quad \text{where } d_k = \sum_{i=0}^k a_i b_{k-i} \quad (4.2)$$

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：

$$\alpha^k = \sum_{i=0}^{m-1} g_i^{(k)} \alpha^i \quad \text{where } 0 \leq i \leq m-1 \text{ and } m \leq k \leq 2m-2 \quad (4.3)$$

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：

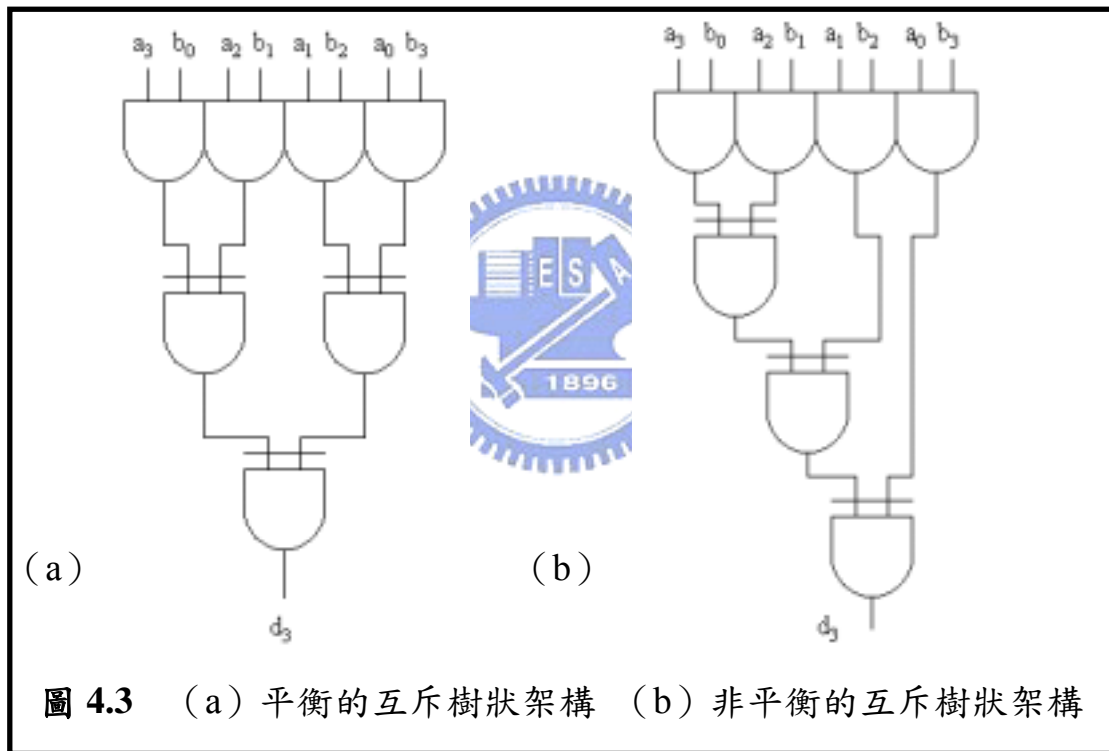
$$\begin{aligned} C &= \sum_{k=m}^{2m-2} d_k \alpha^k + \sum_{k=0}^{m-1} d_k \alpha^k = \sum_{k=m}^{2m-2} d_k \left(\sum_{i=0}^{m-1} g_i^{(k)} \alpha^i \right) + \sum_{k=0}^{m-1} d_k \alpha^k \\ &= \sum_{k=0}^{m-1} \alpha^k \left(d_k + \sum_{j=m}^{2m-2} d_j g_k^{(j)} \right) = \sum_{k=0}^{m-1} \alpha^k c_k \end{aligned} \quad (4.4)$$

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

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 (4.2) 得知)
- (二) 模數的化簡 (Modular Reduction) (方程式 (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： $(a_3 \text{ AND } b_0) \text{ XOR } (a_2 \text{ AND } b_1) \text{ XOR } (a_1 \text{ AND } b_2) \text{ XOR } (a_0 \text{ AND } b_3)$



從圖 4.3 中可以看出假如使用 (a) 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) 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 4.4 是當非規則全平行乘法器建立在 $GF(2^4)$ 時之硬體架構，我們可以注意到上半部就是前面所提到的第一步驟：兩數相乘；下半部就是第二步驟：模數化簡。其中 $g_i^{(k)}$ ($0 \leq i \leq 3, 4 \leq k \leq 6$) 是事先算好的。

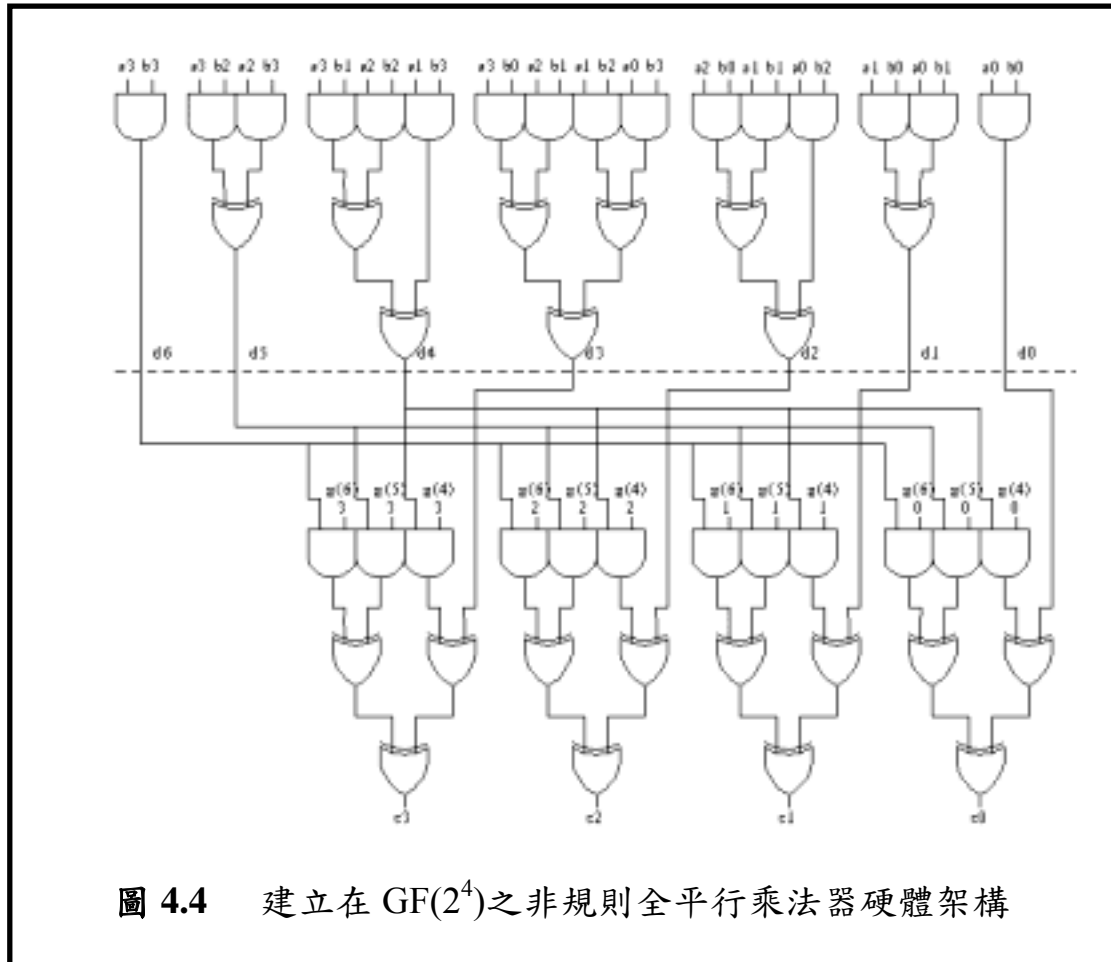


圖 4.4 建立在 $GF(2^4)$ 之非規則全平行乘法器硬體架構

4.3 處理錯誤症狀 (Syndrome) 之硬體摺疊架構

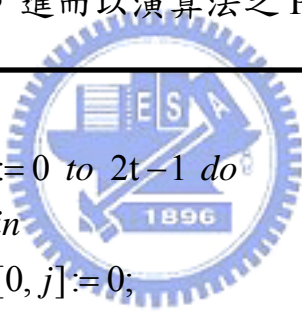
由 2.4.1 小節可知，錯誤症狀的計算可以被視為是一連串伽羅瓦場的乘加運算，根據定義：

$$S(x) = \sum_{j=0}^{2t-1} S_j \cdot x^j \quad \text{where} \quad S_j = \sum_{i=1}^n R_{n-i} \cdot (\alpha^{m_0+j})^{n-i} \quad (4.5)$$

然而 S_j 可以化成下面的形式：

$$S_j = (\dots((R_{n-1} \cdot \alpha^{m_0+j} + R_{n-2}) \cdot \alpha^{m_0+j} + R_{n-3}) \dots) \alpha^{m_0+j} + R_0 \quad (4.6)$$

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] · αm0+j + Rn-i
      end;
      Sj := z[n, j]
    end
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 4.5 所示，每一個運算單元負責處理一個錯誤症狀的計算。

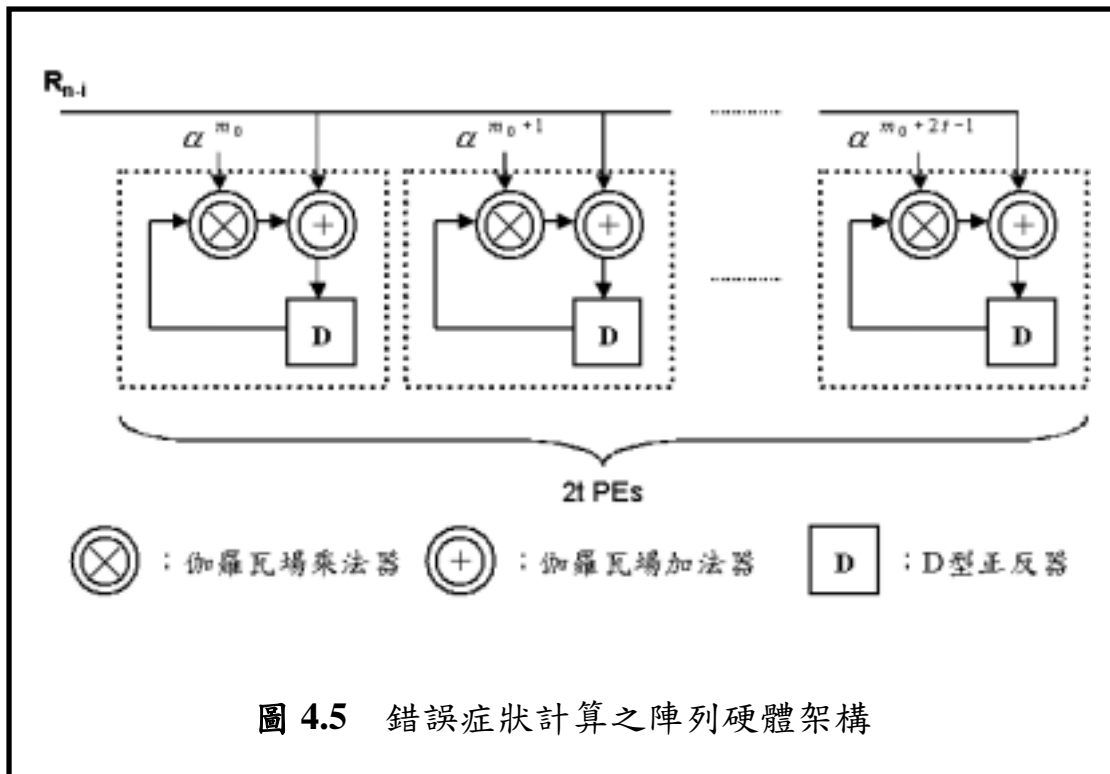


圖 4.5 錯誤症狀計算之陣列硬體架構

4.3.1 乘加運算器及暫存器規劃

這一節就以上述的陣列架構為起點，開始建構出處理錯誤症狀的硬體，接著利用摺疊演算法的推導，將摺疊硬體的好處套用在里德所羅門解碼器上。

在開始摺疊錯誤症狀的陣列架構之前，首先必須考慮採用的運算單元個數。根據解碼演算法定義，假設除錯能力為 t ，則必須計算出 $2t$ 個錯誤症狀，因此其處理錯誤症狀硬體為 $2t$ 個運算單元的陣列。現在就以除錯能力最大為 8 作為例子，而期望使用 4 個運算單元來架構處理錯誤症狀的硬體。由於除錯能力最大為 8，所以原先的陣列長度為 16；採用 4 個運算單元，則可以將 16 個運算單元分為 4 列，如圖 4.6 所示。

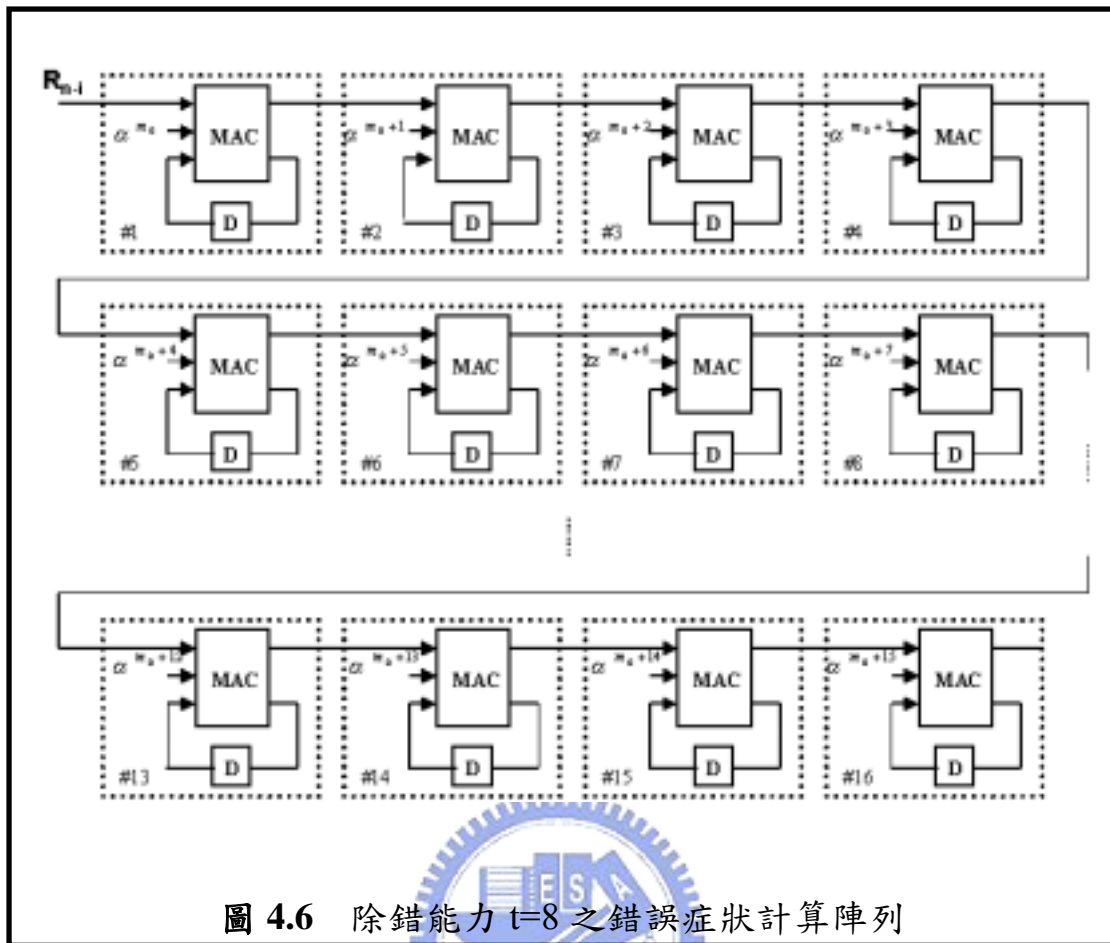


圖 4.6 除錯能力 $t=8$ 之錯誤症狀計算陣列

現在打算將一個回合的資料分為 4 個的時間點來運算，每一列各在一個時間點完成，首先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，因此在第一個時間點時，送入資料 R_{n-i} ，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中，而此筆 R_{n-i} 資料也會傳至第一列與第二列的銜接暫存器，至於第二、三、四列的乘加器則均不動作；到了第二個時間點，讀入銜接暫存器中的 R_{n-i} 值，第二列開始運作，並將結果存入該列各暫存器的第二個位置，其餘三列均不動作，當然 R_{n-i} 也傳至下一個連接暫存器；到了第三個時間點，第三列會開始計算並將結果存到各暫存器的第三個位置，其餘各列無動作；第四個時間點的動作以此類推，圖 4.7 說明了上述的動作。

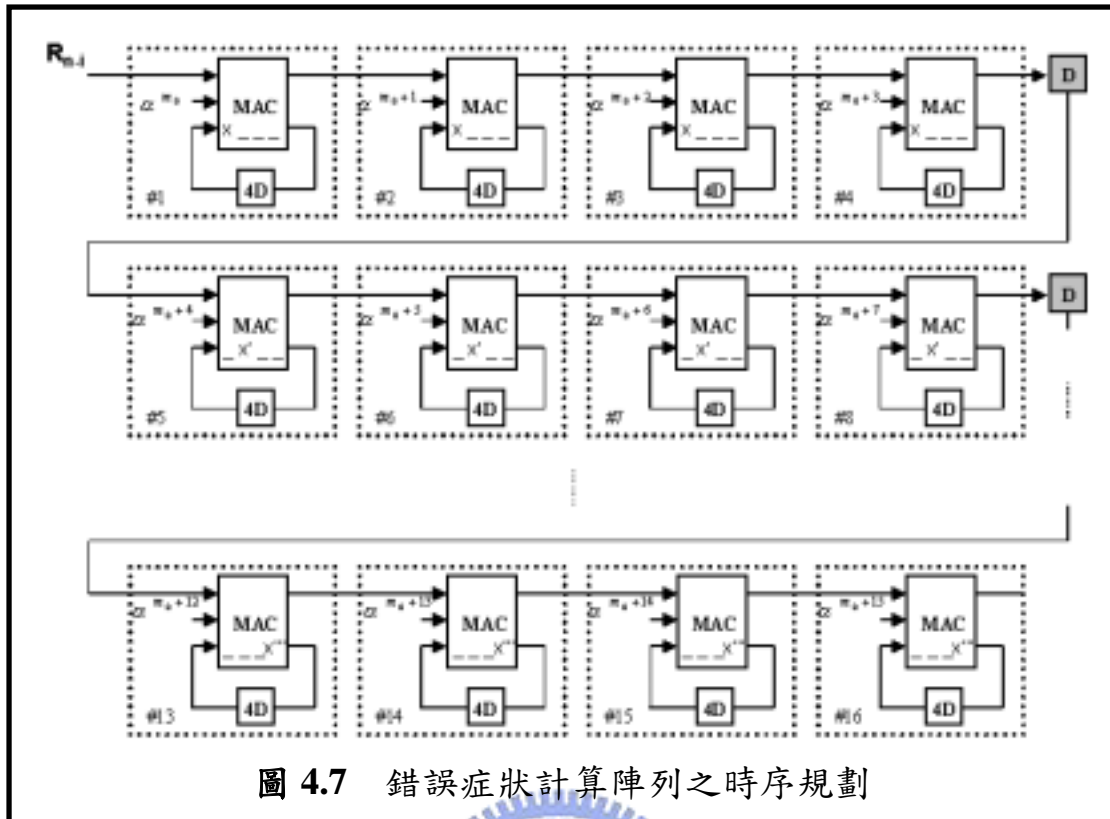


圖 4.7 錯誤症狀計算陣列之時序規劃

由此可知，每列中相對應位置之乘加運算器，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運算器不會在同一個時間點同時動作，所以可以將這四列壓縮成一列。於是只要在一個回合中送入一個 $R_{n,i}$ 資料，且在這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位置就可得到正確的運算。因此接下來的工作就是根據不同時間點，調整每個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。圖 4.8 展示的是摺疊後的硬體架構。

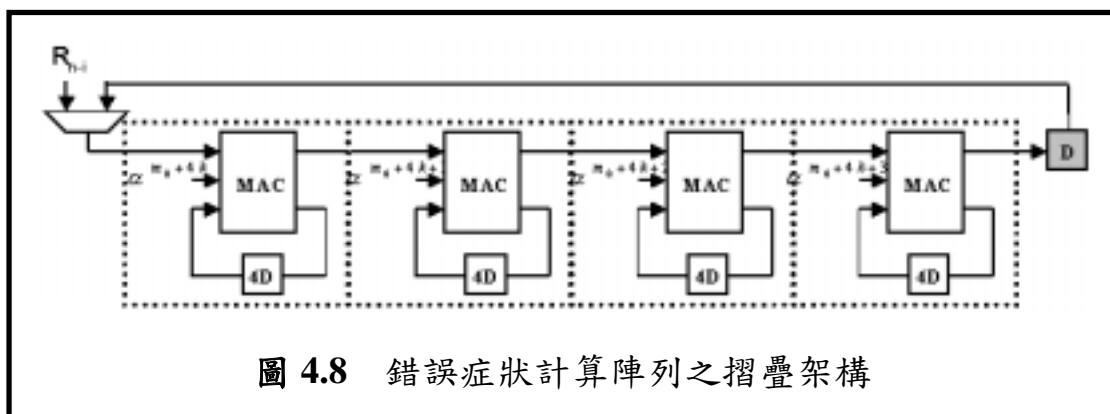


圖 4.8 錯誤症狀計算陣列之摺疊架構

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，其中每個乘加運算器擁有三個輸入端及兩個輸出端，從暫存器中讀入上一回合所乘累加結果，乘上該運算器在該時間點對應的 α 值，再加上此時刻的外部輸入 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 α 值以及 R_{n-i} ，輸出則存回累加暫存器，並將 R_{n-i} 送入銜接暫存器，如圖 4.9 所示。

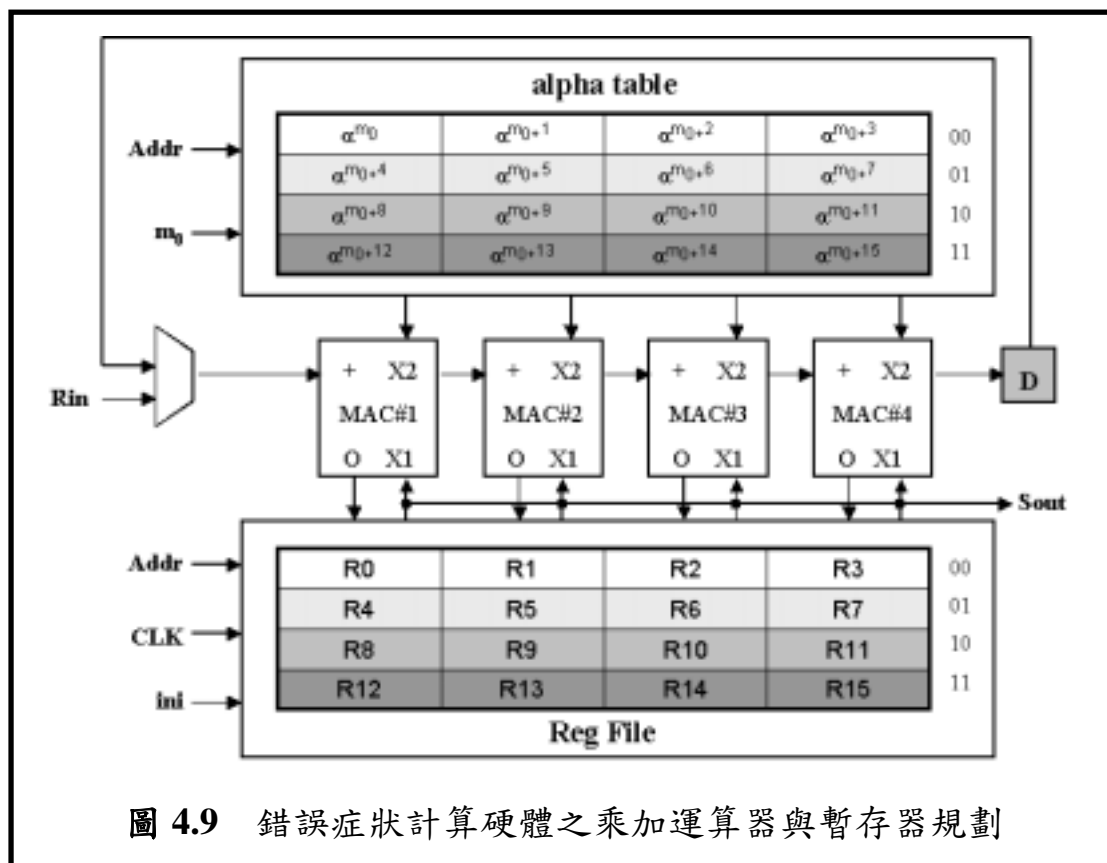


圖 4.9 錯誤症狀計算硬體之乘加運算器與暫存器規劃

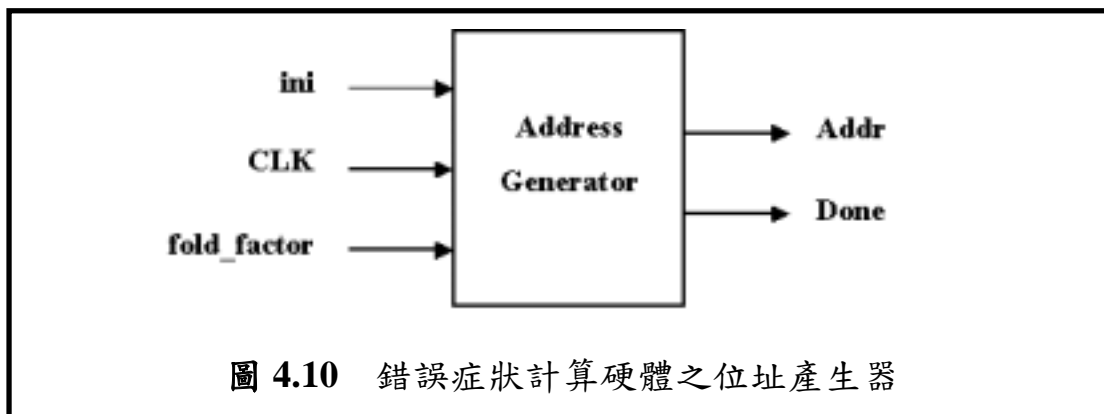
由圖中看來， α 資料表及暫存器檔案在每個時間點各會讀入四筆資料分別送入四個運算器中，因此在兩個資料表中可以將同一列的四筆資料作網綁的動作，由 Addr 來給予定址命名。舉例來說，當第一個時間點時，Addr 送入 00 值，因此 α 資料表會送出 $\alpha^{m_0} \sim \alpha^{m_0+3}$ 到四個運算器；同理，暫存器檔案也會送出上一回合儲存在 R0~R3 中的乘累加結果到運算器去計算；而在第二、三、四個時間點時，則以此類推，只需要輸入不同的 Addr 值就可達成。當然一旦計算完成時，Sout

所輸出的值即為錯誤症狀，其依照 Addr 的轉換，一個時脈連續輸出四筆錯誤症狀。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 輸入，選擇載入 $\alpha^0 \sim \alpha^{15}$ 或 $\alpha^1 \sim \alpha^{16}$ 兩種模式。

4.3.2 位址產生器的設計

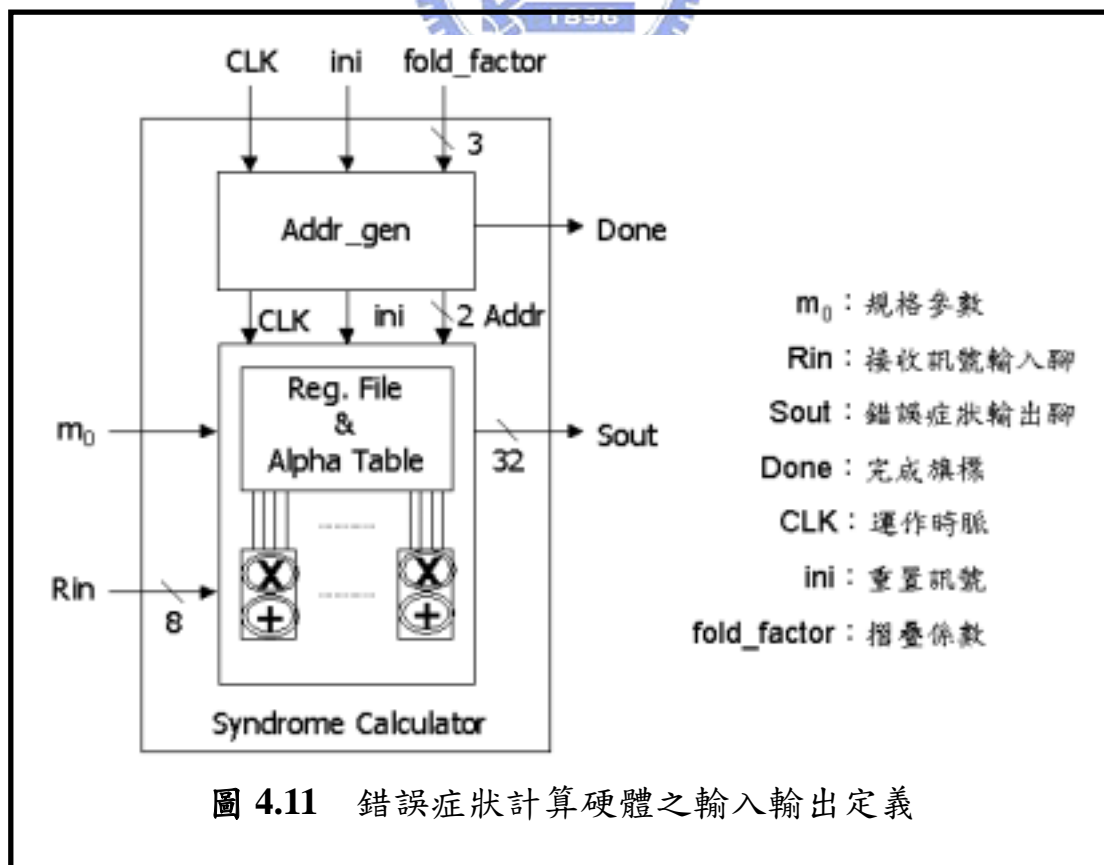
位址產生器 (Address Generator) 是用來規劃每個時間點所應送入資料表位址的一個控制機制，依據時脈 (Clock) 對位址訊號作遞增的動作，其作用就如同一個有計數上限的計數器。每當一個時脈發生，位址訊號就會向上加 1，而達到摺疊係數 (Fold Factor) 這個計數上限時則歸零。以方才推導的架構為例子來說，當除錯能力 t 為 6 時，摺疊係數等於 3，因為原先的 12 個運算單元只想用 4 個運算器來完成，所以必須摺成 3 摺，因此位址計數也會從 00 計數，經過 01 到 10，而後又歸回 00；如果除錯能力為 8 時，則摺疊係數等於 4，也就是計數 00、01、10、11 後歸於 00。

圖 4.10 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用，另一個則是標記硬體運作的回合數，進而掌握錯誤症狀計算的完成時間。在方塊中的輸入 ini 訊號是一個重置作用的訊號，使位址計數歸零，此外這個訊號也會傳遞至暫存器檔案，將累加暫存器作初始化歸零。CLK 則為計數標準，每當在時脈訊號的上緣觸發時，計數就會向上加 1。Fold_factor 訊號是用來規定計數的上限。輸出的 Done 訊號則是當完成錯誤症狀計算時，Done 訊號升為高準位，告知此時的資料是該截取的，其餘時間 Done 訊號均為低準位。



4.3.3 錯誤症狀計算硬體之輸入輸出定義

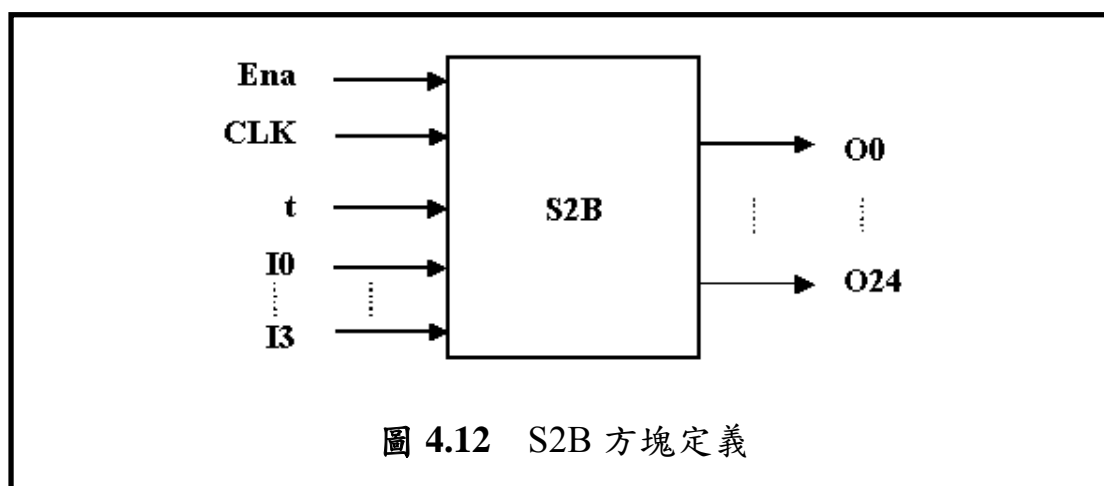
在這小節的最後，對於計算錯誤症狀這個方塊的輸入輸出腳位作個定義，這樣在整個系統整合時才不易搞混。輸入的部分有接收的到信號 Rin、時脈 CLK、重置 ini、摺疊係數 fold_factor，以及參數 m_0 ；輸出部分包括四筆錯誤症狀輸出，還有完成旗標 Done，如圖 4.11 所示。



4.4 信號症狀計算器與 BM 方塊之資料傳遞介面

為了讓整個里德所羅門解碼器可以動態調整規格參數，在計算方塊間的資料銜接，必須適當的規劃，讓解碼器在不同的規格參數下也能夠順利將資料傳到下一級。

信號症狀計算器與 BM 方塊之資料傳遞方塊 (S2B Block) 主要是由暫存器與多工器所組成，圖 4.12 為此方塊的定義。依照致能輸入訊號的指示，在其為高準位時，每個時脈將輸入值 I 存入內部的暫存器中，輸入值的個數與錯誤症狀計算器所擁有的運算單元個數相同，在此以前一節所採用的四個運算單元為例，所以有四個輸入埠，另外，致能訊號即為錯誤症狀完成訊號，當計算完成開始輸出資料的同時，致能訊號也被啟動。由此可見，S2B 方塊就是將最後一回合錯誤症狀計算完成的資料，全部存到其內部的暫存器中。至於多工器的部分則與輸出有關，下一級的 BM 方塊在初始化時必須載入 $3t+1$ 個初始值，其順序為一個 1、 t 個 0 以及 $2t$ 個錯誤症狀，因此外部輸入 t 值就是用來選擇不同的參數規格 t ，將已存在於內部暫存器中的錯誤症狀值，送到輸出埠的正確位置。而輸出埠的個數以最大除錯能力 t 是 8 為例，共有 25 個輸出埠。



4.5 尋找錯誤位置多項式以及錯誤大小評估多項式之硬體摺疊架構

在尋找錯誤位置多項式與錯誤大小評估多項式這部分，2.4.2 小節中提到有 Berlekamp 疊代演算法以及最大公因式演算法兩種。之前學者針對這兩種演算法，分別提出不少關於其硬體架構實現之論文 [8][9][10]。

1995 年，K. Iwamura [8] 提出用 Systolic 陣列實現最大公因式演算法，雖然可以很規則地運用到相同的基本運作單元，但是在時間的消耗及控制的難易度上面卻是增加了不少。其中需要用到兩套時脈波來控制不同的 D 型正反器，整個運作的延遲時間 (Latency) 也很長。2001 年由 D. V. Sarwate [9] 提出一種使用 Berlekamp 疊代演算法的硬體架構，並加以修改變形，進而可以同時求取錯誤位置多項式與錯誤大小評估多項式，其硬體實現方法也是運用到陣列形式，且在 $2t$ 的時脈週期中可以完成。

4.5.1 Reformulated Inversionless Berlekamp-Massey 疊代演算法

D. V. Sarwate [9] 所提出的變形演算法，可以在 $2t$ 個時脈週期內 (t 為里德所羅門碼的除錯能力)，一併將錯誤位置多項式以及錯誤大小評估多項式同時找出來，比起最大公因式演算法，其大大的減短運作的延遲時間。在此，演算法的變形過程不再一一敘述，只將此演算法列出如下。

Reformulated Inversionless Berlekamp-Massey (RiBM) Algorithm

Initialization :

$$\tilde{\delta}_{3t}(0) = 1; \tilde{\delta}_i(0) = 0 \text{ for } i = 2t, 2t+1, \dots, 3t-1. k(0) = 0. \gamma(0) = 1$$

Input : $s_i, i = 0, 1, \dots, 2t-1$

$$\tilde{\delta}_i(0) = \tilde{\theta}_i(0) = s_i, (i = 0, 1, \dots, 2t-1)$$

for $r = 0$ step 1 until $2t-1$ do

begin

$$\text{Step 1. } \tilde{\delta}_i(r+1) = \gamma(r) \cdot \tilde{\delta}_{i+1}(r) - \tilde{\delta}_0(r) \cdot \tilde{\theta}_i(r), (i = 0, 1, \dots, 3t)$$

$$\text{Step 2. if } \tilde{\delta}_0(r) \neq 0 \text{ and } k(r) \geq 0$$

then

begin

$$\tilde{\theta}_i(r+1) = \tilde{\delta}_{i+1}(r), (i = 0, 1, \dots, 3t)$$

$$\gamma(r+1) = \tilde{\delta}_0(r)$$

$$k(r+1) = -k(r) - 1$$

end

else

begin

$$\tilde{\theta}_i(r+1) = \tilde{\theta}_i(r), (i = 0, 1, \dots, 3t)$$

$$\gamma(r+1) = \gamma(r)$$

$$k(r+1) = k(r) + 1$$

end

end

$$\text{Output : } \lambda_i(2t) = \tilde{\delta}_{i+i}(2t), (i = 0, 1, \dots, t); \omega_i^{(h)}(2t) = \tilde{\delta}_i(2t), (i = 0, 1, \dots, t-1).$$

D. V. Sarwate 也提到變形的 Berlekamp 疊代演算法可使用陣列的形式來實現，架構上可區分為基本運算單元以及控制單元兩部分，陣列長度為 $3t+1$ 。圖 4.13 為 RiBM 演算法之基本運作單元與控制單元。圖 4.14 為其整體硬體架構圖，在基本運算單元及控制單元中的數值表示各暫存器的初始化設定值，在 $2t$ 個時脈後，儲存於運算單元上方各暫存器中的值即為兩多項式的係數。

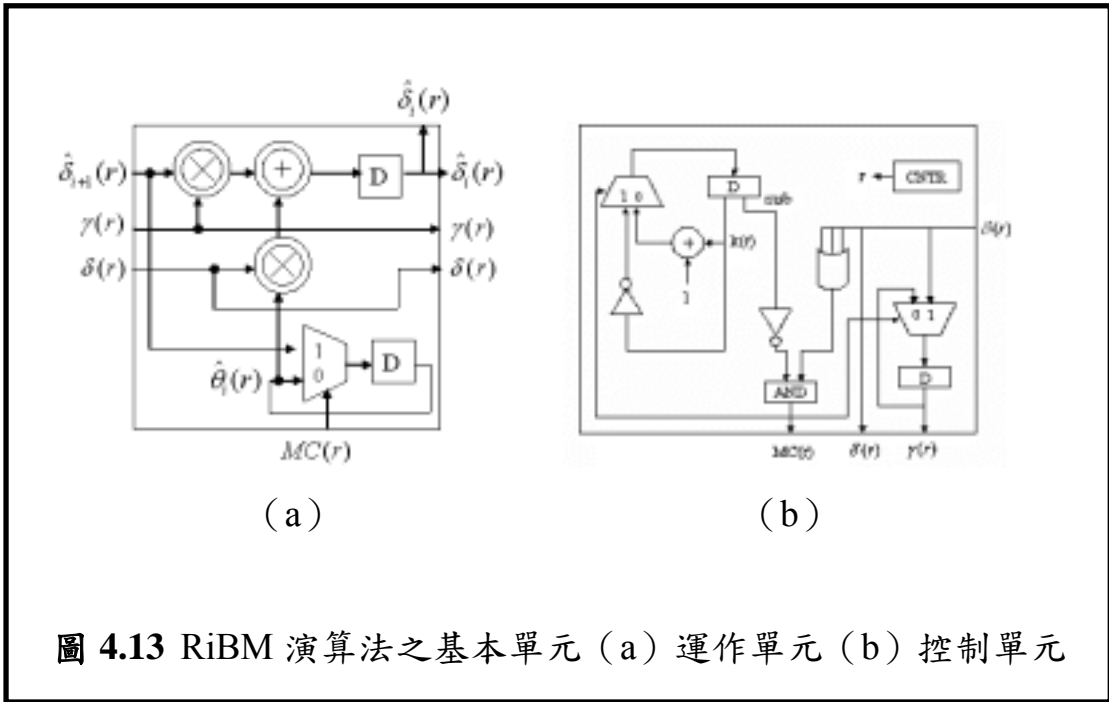


圖 4.13 RiBM 演算法之基本單元 (a) 運作單元 (b) 控制單元

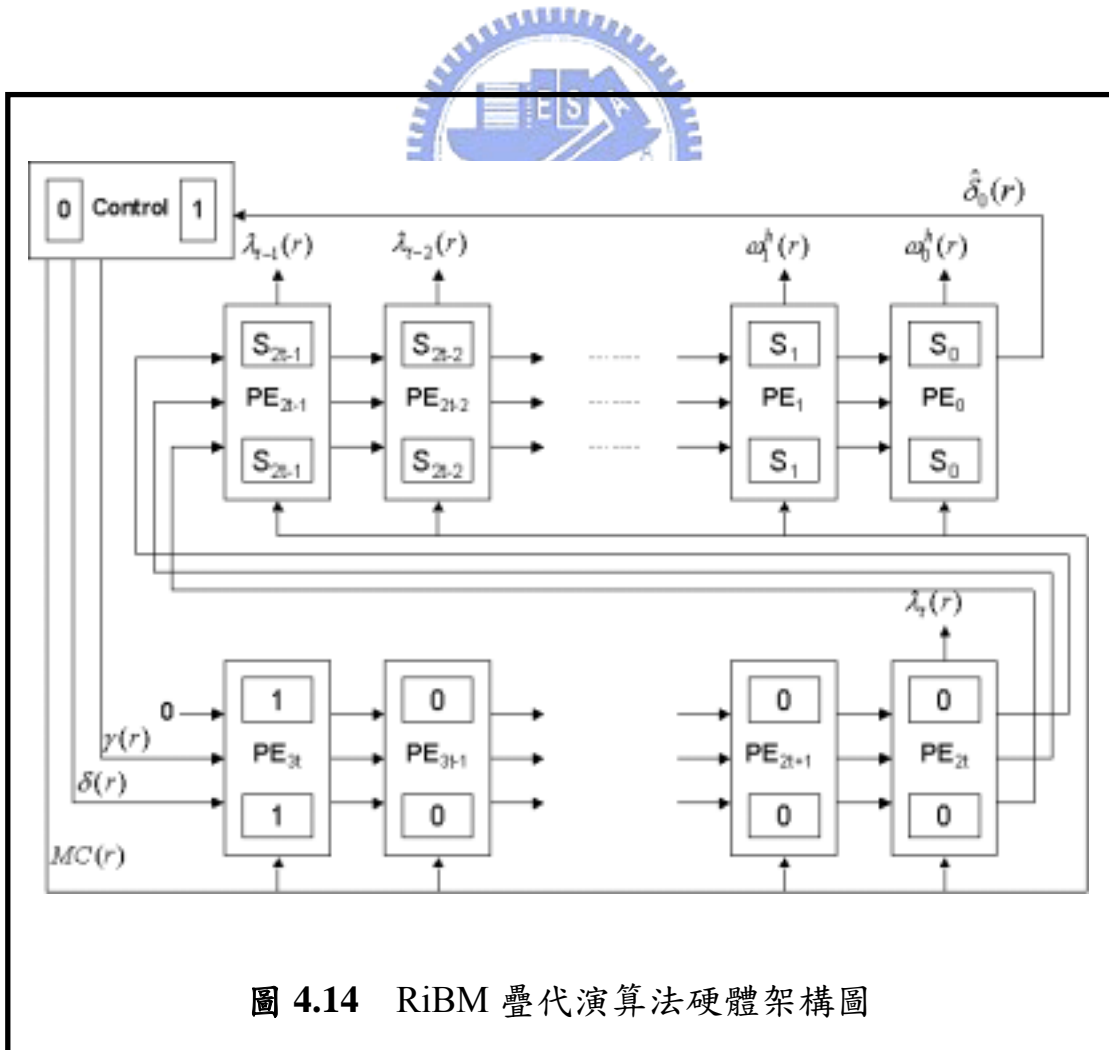


圖 4.14 RiBM 疊代演算法硬體架構圖

4.5.2 乘加運算器及暫存器規劃

現在將 RiBM 架構中的運算器與暫存器分離表示，則會形成圖 4.15 的型式。由圖中可知，每一個運算器有四個輸出，其中的三個輸出是屬於向前路徑的，其餘的一個則是迴授路徑。接下來將採用摺疊的技巧來節省一些硬體面積。由於採用 RiBM 的陣列架構，因此其硬體陣列長度為 $3t+1$ ，現在以除錯能力最大為 8 作為例子，希望使用 4 個運算單元來架構此硬體方塊，則可以將 25 個運算單元分為 7 列。

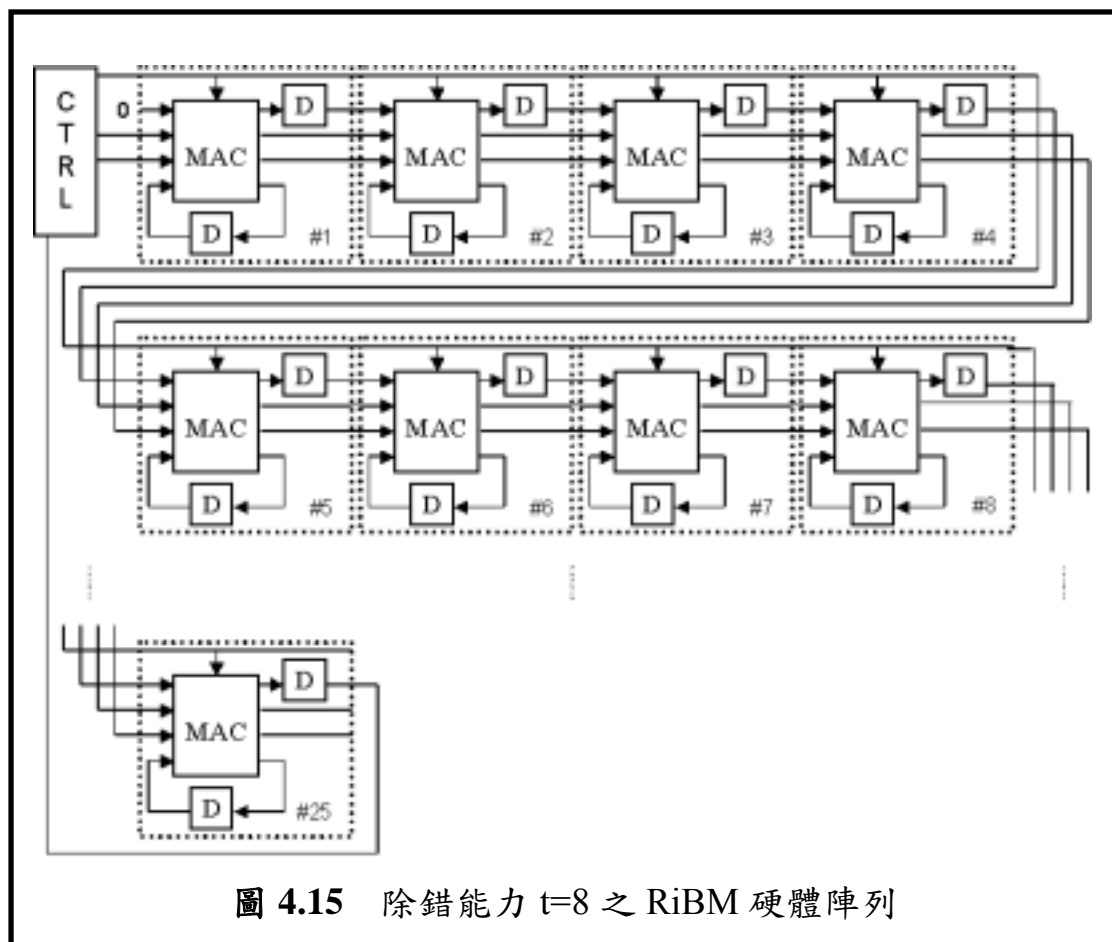


圖 4.15 除錯能力 $t=8$ 之 RiBM 硬體陣列

現在要將一個回合分為 7 個時間點，因此所有的暫存器都必須乘上 7 倍，而第一列在第一個時間點動作，第二及第三列各在第二及第三時間點動作，其餘以此類推，另外，原本在向前路徑上的訊號，列

與列交接處必須插入連接暫存器，用來傳遞各列在不同時間點工作所需的資料，調整過後的結果，從圖 4.16 可以清楚的瞭解。於是這 7 列運算器的工作時間點將會兩兩彼此錯開，各自在不同的時間點作運算。還有一點值得注意的是，在控制器部分需要從原始架構中最後一個基本運算單元的暫存器迴授資料，來作為下一回合輸出控制訊號的修改，所以只有在完成一整個回合之後，也就是各列運算器都各完成一次運作之後，控制器才會作一次更新，而從下一回合開始輸出新的控制訊號。

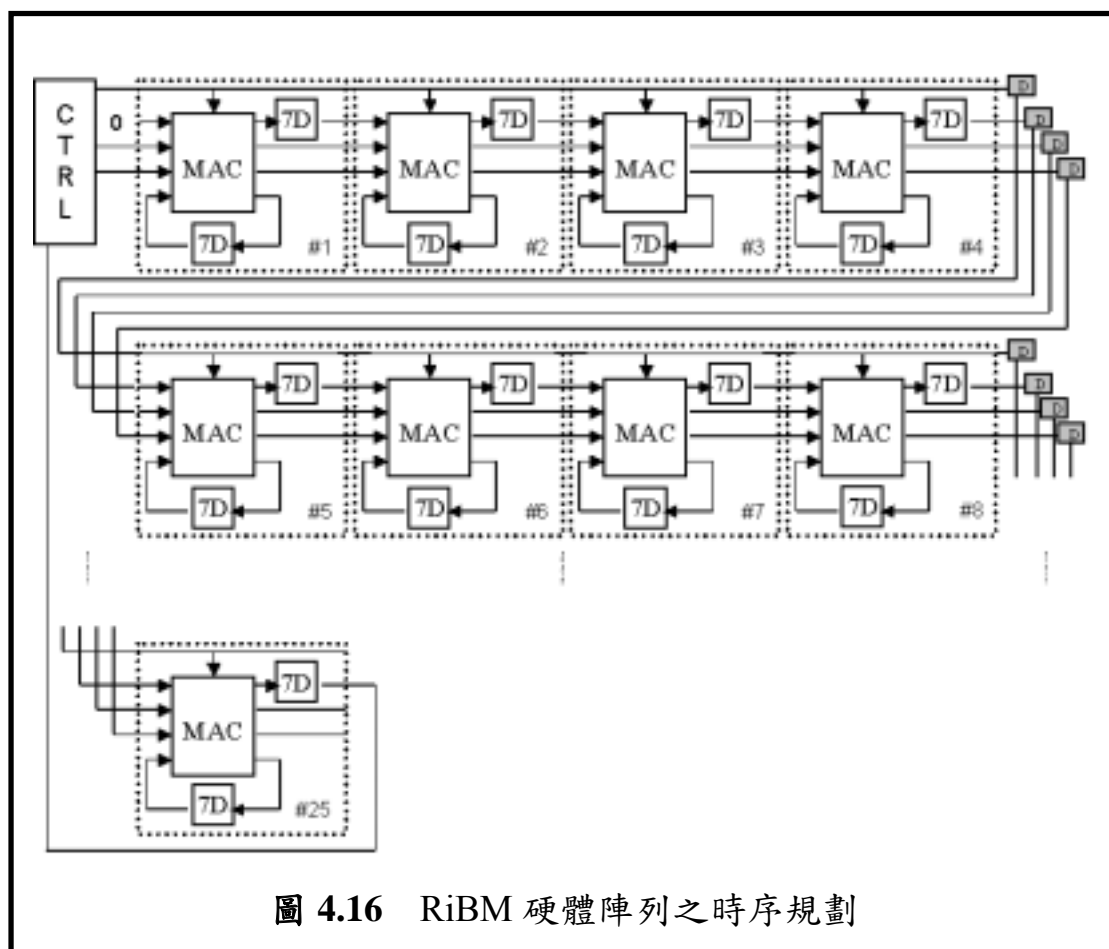


圖 4.16 RiBM 硬體陣列之時序規劃

由於這七列運算器的工作時間點是兩兩互斥的，因此可以將其壓縮，作摺疊的動作，只使用四個運算器，分別在七個時間點達成原本一個回合所應完成的事。壓縮後的架構如圖 4.17。

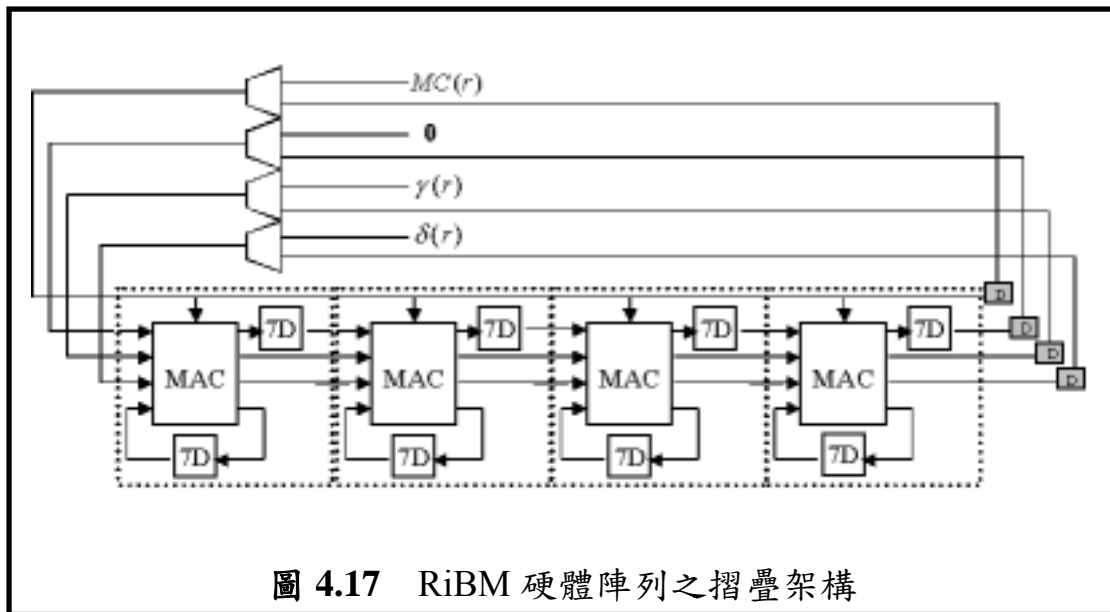


圖 4.17 RiBM 硬體陣列之摺疊架構

在完成摺疊的步驟之後，現在則要規劃每個回合的各時間點各運算器所需輸入及輸出的資料。RiBM 的運算器有三個訊號是來自控制器，所以主要必須規劃的是如何存取剩下的上下兩套暫存器檔案。圖 4.18 是摺疊後 RiBM 架構的乘加器與暫存器檔案的規劃。在兩個暫存器檔案中下方的暫存器是屬於迴授暫存器，因此在相對應的時間點，由 Addr 送入正確的位置來讓運算器讀入資料，並在運算之後再存入相同的暫存器即可。上方的暫存器是在向前路徑上，所以根據時間點由暫存器讀出的四筆資料，前三筆分別送入第二、三、四個運算器，而最後一筆則存入連接暫存器中，而第一個運算器的輸入則由多工器來選擇，當每一回合的第一個時間點，也可說是當 Addr=000 時，由 0 當作輸入，其餘皆由連接暫存器當輸入；至於四個運算器的輸出，則分別存入相對應的四個暫存器中。附帶一提，上下兩套暫存器在整體架構開始動作之前，必須有一個初始值設定的步驟，ini 訊號就是一個重置的指示；此外，上方的暫存器還有一個 t 輸入訊號，這是為了根據不同的除錯能力 t，選擇所應該迴授到控制器的 δ 值，因為 t 值不同，基本運算單元陣列的長度就會不同，因此演算法中的最後一個運算單元也會不同。

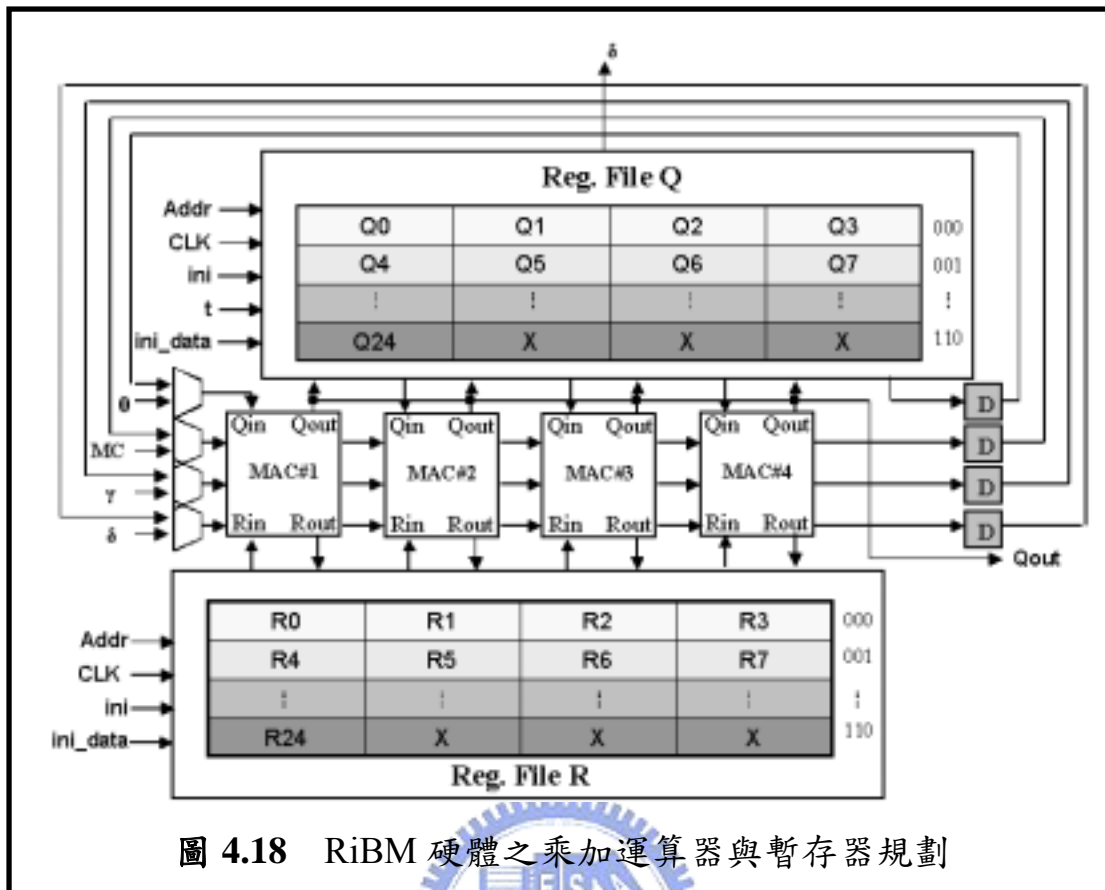


圖 4.18 RiBM 硬體之乘加運算器與暫存器規劃

在控制器的部分，其內部也有兩個暫存器，用來儲存輸出更新的資料，而資料一回合只更新一次，所以必須有 Ena_ctrl 訊號來傳達更新的時機，在整個計算流程開始之前，內部的暫存器也必須作初始化，這個動作則由 ini 告知。這樣一來，在同一回合中控制器會送出三個訊號給運算器，而迴授的 δ 訊號，則根據參數 t 的不同，由暫存器檔案輸出提供。圖 4.19 是 RiBM 控制單元的方塊圖。

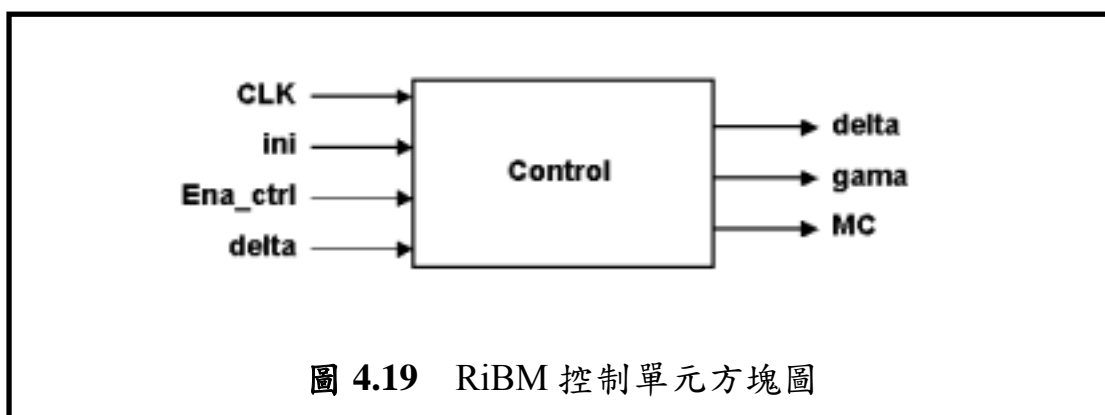


圖 4.19 RiBM 控制單元方塊圖

4.5.3 位址產生器的設計

在 RiBM 架構中，為了確保暫存器檔案可以在正確的時刻作正確的存取，並讓控制單元在正確時刻更新資料，接下來需要設計一個位址產生器。

圖 4.20 即為 RiBM 架構的位址產生器的方塊圖，這個產生器內部包含兩個計數器，其中一個計數器根據時脈 (CLK) 作為計數標準，讓暫存器檔案由位址 000 開始存取，直到摺疊係數 (fold_factor) 這個上限為止，每當 Addr 由 000 重新計數的同時，也就是一個新回合的開始。另一個計數器則是用來紀錄運算所執行的回合數，根據輸入的錯誤修正能力參數 t，當到達 $2t$ 個回合時，代表計算完成，此時所輸出的高準位 Done 訊號，即是用來告知工作完成的一個旗標。

此外，Ena_ctrl 是操作控制單元更新資料的訊號，在每一個新回合開始之前，也就是每回合的最後一個時間點，必須驅動控制器允許其更新，其餘時刻則關閉更新的機制。Ini 訊號則是一個重置的指標，可以強迫位址訊號歸零。

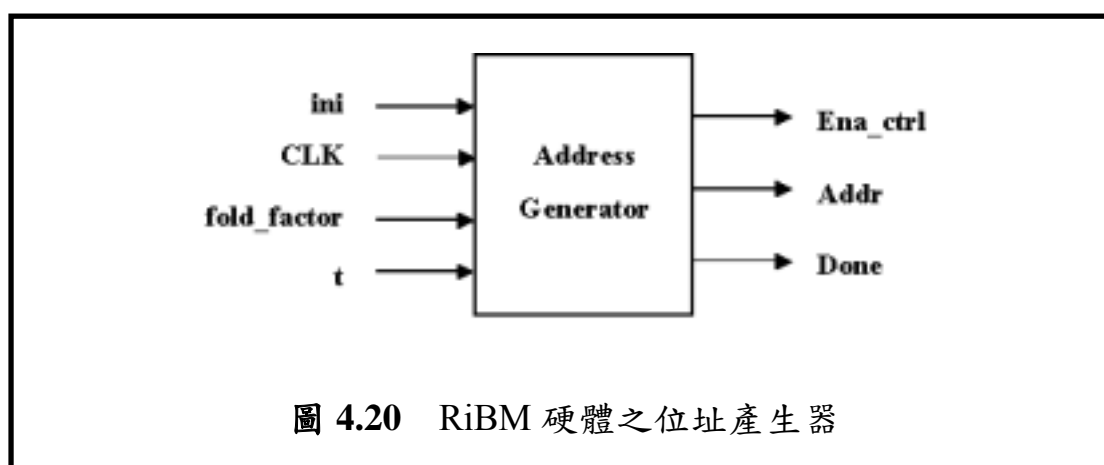
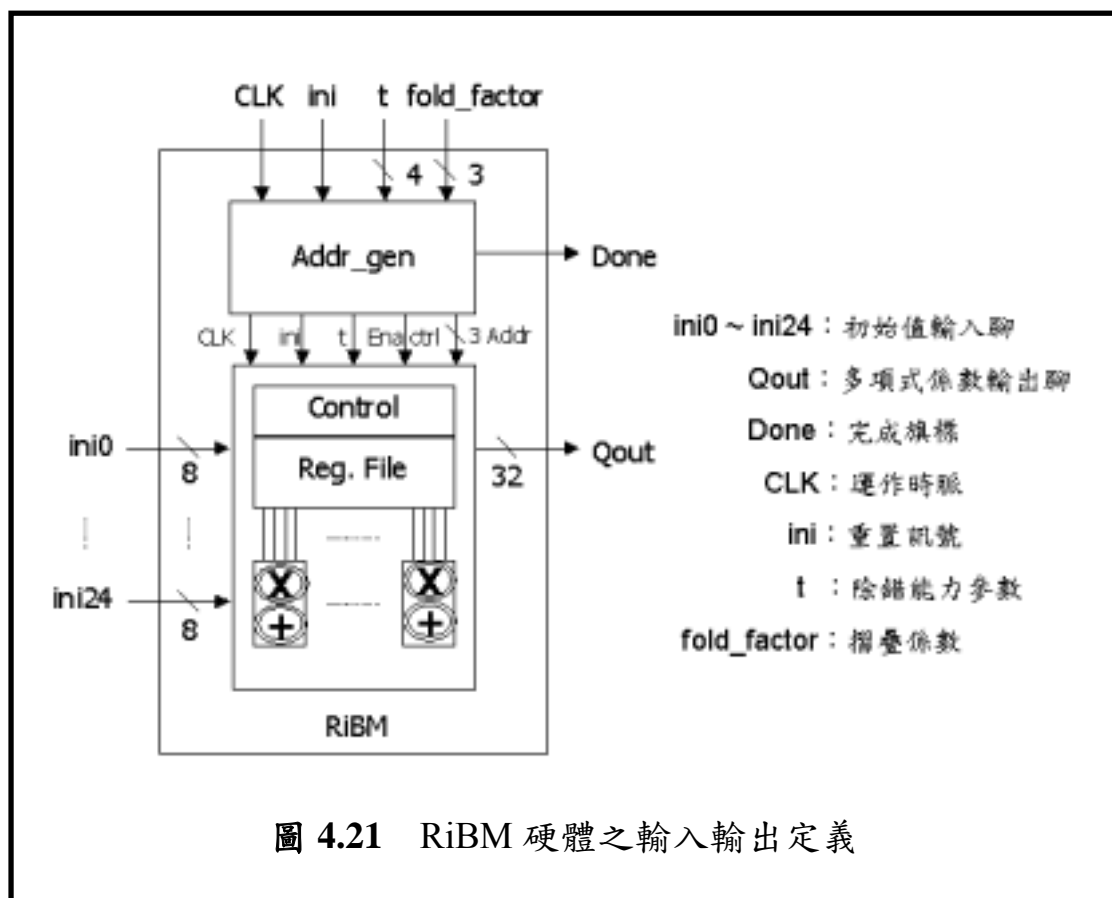


圖 4.20 RiBM 硬體之位址產生器

4.5.4 RiBM 硬體之輸入輸出定義

最後依照慣例，對整個 RiBM 摺疊硬體架構的輸入輸出腳位作一個明確的介紹。圖 4.21 就是腳位定義的圖示，在這裡要特別說明的是輸入的初始值腳位共有 25 組，每組有 8 個位元，因為整個架構是以除錯能力參數 t 最大是 8 為前提來作推導，因此共有 $3t+1$ 個運算單元需要初始化，所以必須有 25 個初始值輸入腳位。



4.6 BM 方塊與多項式估算器之資料傳遞介面

BM 方塊與多項式估算器之資料傳遞方塊 (B2P Block) 和 S2B 方塊一樣，都是為了讓解碼器在不同的規格參數下能夠順利完成資料傳遞，其為 BM 架構與錯誤修正器中多項式估算方塊間的傳遞介面，內部也是由暫存器與多工器所組成，圖 4.22 為其定義。

依照 BM 方塊所輸出的計算完成訊號，將其送到致能輸入埠，在高準位時，每個時脈會將輸入值 I 存入內部被致能的暫存器中，若 BM 架構採用的運算單元為四個，則輸入值的個數也必須與其相同。故 B2P 方塊就是將最後一回合 BM 架構計算完成的資料，全部存到其內部的暫存器中，而完成訊號高準位所經歷的時脈數，也是一個回合的時間長度。

至於外部輸入 t 值就是根據不同的參數規格 t ，調整多工器輸出其相對應的暫存器值。輸出對象包括 $\Lambda(x)$ 、 $x \Lambda'(x)$ 與 $\omega^{(h)}(x)$ 三個多項式估算器，依照不同的 t 值，各有 $t+1$ 、 $t+1$ 與 t 個係數，若最大除錯能力 t 為 8，則共需 26 個輸出埠。此外，需要特別一提的是 $x\Lambda'(x)$ 相當於 $\Lambda(x)$ 奇數次方項的總和，所以在 $x\Lambda'(x)$ 係數的輸出上只要將 $\Lambda(x)$ 的偶次方係數設為零即可。

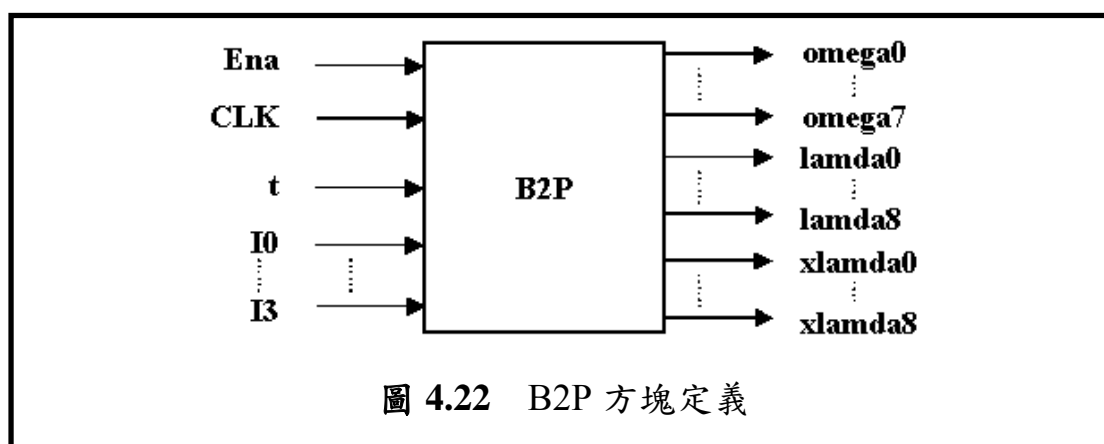


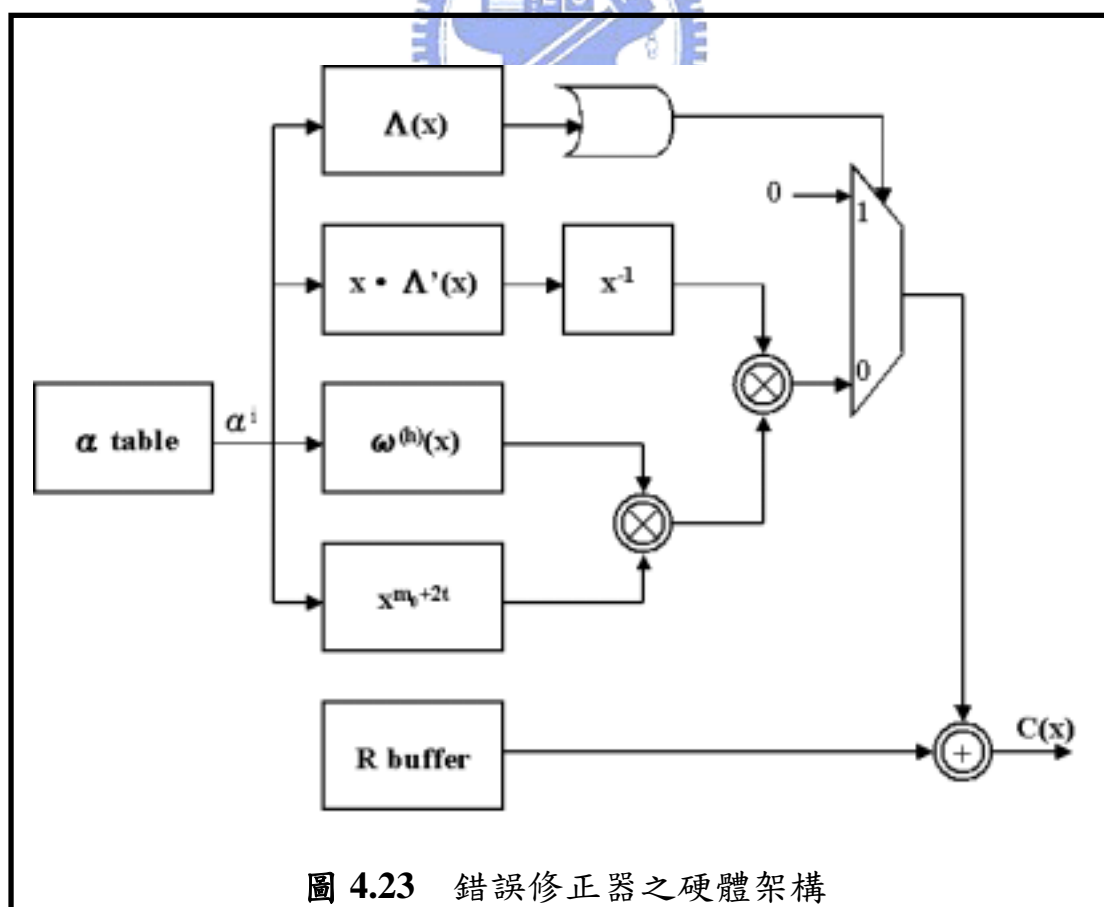
圖 4.22 B2P 方塊定義

4.7 錯誤修正器 (Error Corrector) 硬體架構

錯誤修正器主要是利用 BM 架構所計算出的錯誤位置多項式及錯誤大小評估多項式，配合佛尼演算法，對收到的信號進行修正。由於先前求取錯誤位置多項式及錯誤大小評估多項式是採用變形的演算法，故進行錯誤修正的佛尼演算法也必須跟著作一些調整改變，修正後的演算法表示如下：

$$Y_i = -\frac{X_i^{-(m_0+2t-1)} \omega^{(h)}(X_i^{-1})}{\Lambda'(X_i^{-1})} = -\frac{x^{m_0+2t} \omega^{(h)}(x)}{x\Lambda'(x)} \Bigg|_{x=X_i^{-1}} \quad (4.7)$$

錯誤修正的硬體結構中有幾個組成方塊，主要包括多項式估算器、Chien 尋根法的伽羅瓦場元素輸出表，以及佛尼演算法的計算邏輯，圖 4.23 即為錯誤修正器的整體硬體架構。



4.7.1 多項式估算之硬體摺疊架構

在里德所羅門解碼流程中的最後一個步驟是錯誤修正，在開始進行之前，必須先對錯誤位置多項式作尋根的動作，也就是將伽羅瓦場中所有的元素代入多項式中尋找錯誤位置；另一方面，為錯誤位置的元素也必須代入錯誤大小評估多項式求出錯誤大小的值。因此在進行錯誤修正這一步驟需要一套硬體，能夠計算多項式的值。

圖 4.24 是一個能夠計算最高為八次方多項式之硬體架構，其可計算除錯能力最大為 8 的錯誤位置多項式之值。現在假設此八次多項式可表示為： $f(x) = f_0 + f_1x + f_2x^2 + \dots + f_8x^8 = (((f_8x + f_7)x + \dots)x + f_1)x + f_0$ ，則元素 α^i 從輸入端開始經過九個係數的乘累加，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輸入，倘若此刻在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在下一刻就會得到。

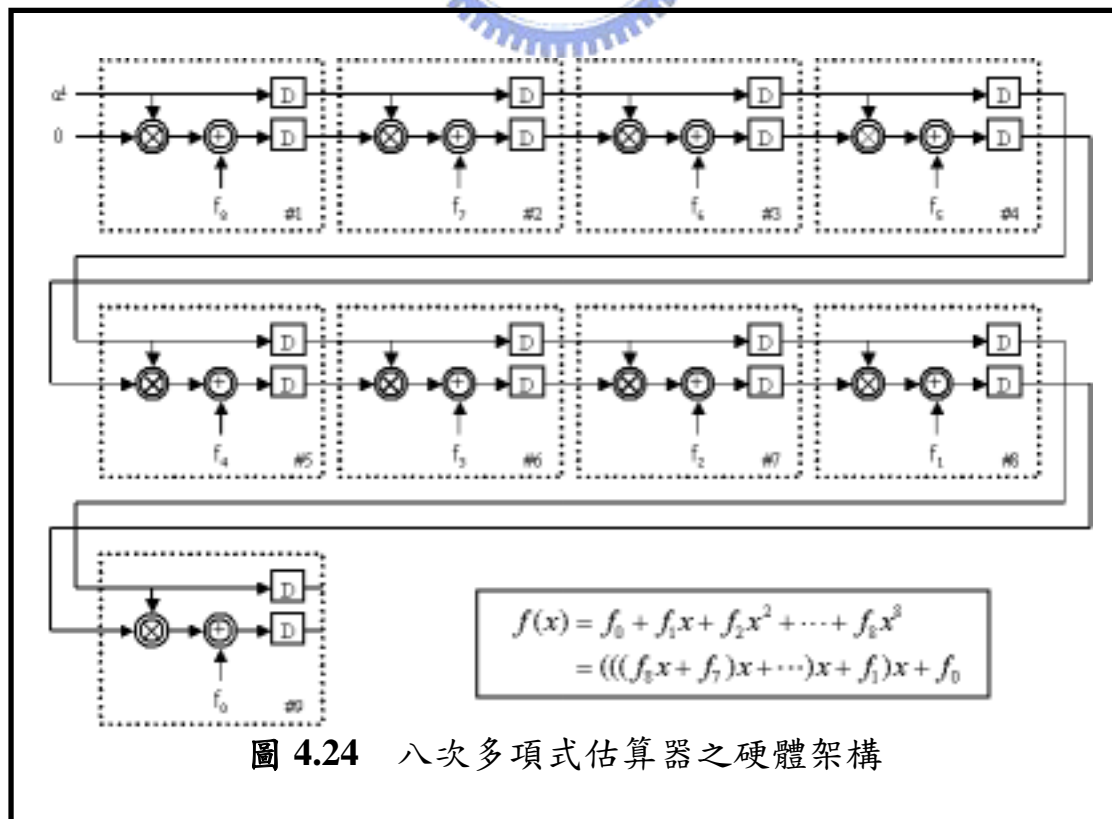


圖 4.24 八次多項式估算器之硬體架構

4.7.1.1 乘加運算器及暫存器規劃

瞭解計算多項式硬體的架構之後，接著如同先前所介紹摺疊的步驟一樣，開始對硬體作一些調整。首先選擇採用四個運算乘加器為一組，也就是希望摺疊後的硬體，只使用四個運算乘加器，於是九個乘加器可以分為三列，並且將原先在一回合內必須完成的動作，劃分為三個時間點來執行，所以每個暫存器必須變成三倍大小，第一列、第二列及第三列分別在第一時間點、第二時間點及第三時間點工作，此外，各列在向前路徑上的銜接處還要額外加上連接暫存器，如同圖 4.25 所表示的。如此一來，這三列各相對應位置的運算器的工作時間點均不互相衝突，故在經過九回合乘累加後，在第三列的第一個運算器計算存入暫存器的值，就是此八次多項式計算最終所應該得到的結果。

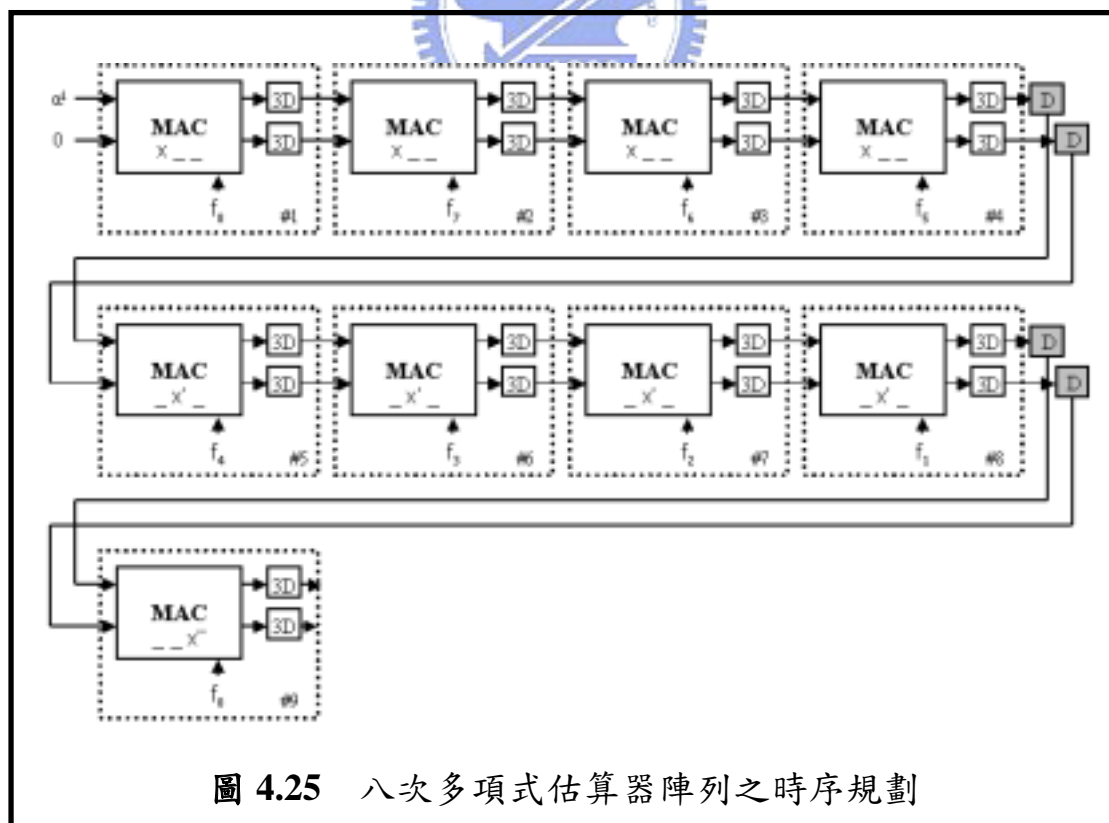
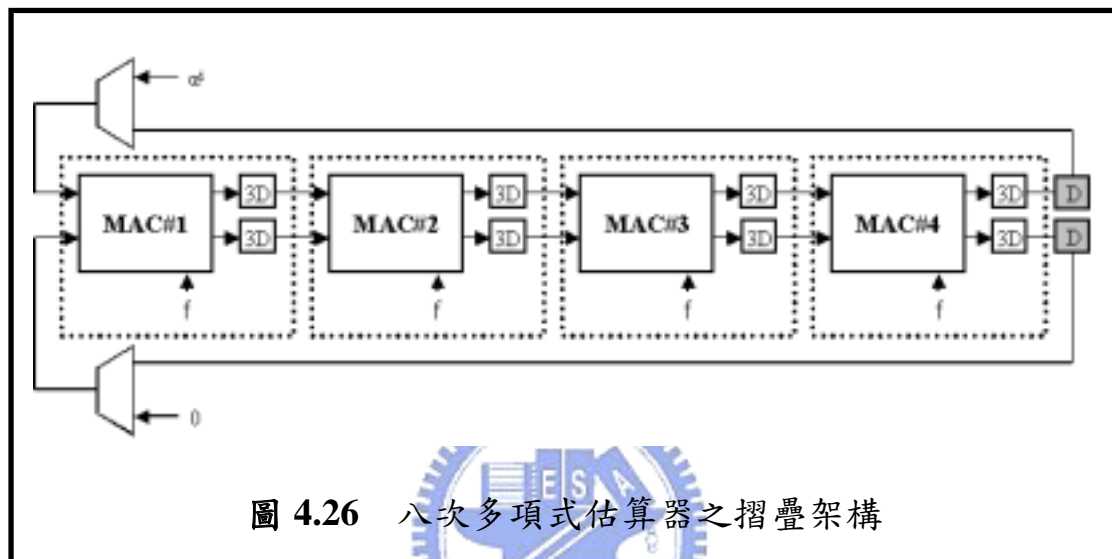


圖 4.25 八次多項式估算器陣列之時序規劃

同樣的，為了節省硬體，將三列乘加器壓縮摺疊，形成圖 4.26 的模樣，圖中的四個乘加器在一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個回合的第一個時間點才會選擇 α^i 或是 0 來當作輸入，其餘時間將會選擇由連接暫存器當作輸入來源。



根據摺疊之後的架構，每個運算器在不同時間點的輸入輸出就必須仔細的安排，圖 4.27 列出這四個乘加運算器在不同的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 A 與 D 會被設為 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，送出資料至運算器計算，當然，除了係數暫存器外，其於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，其餘皆以銜接暫存器為輸入來源。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完成所存入的暫存器的位置，以 t=8 為例，錯誤位置多項式為一個 8 次方多項式，因此 Q 所送出的資料應為 D8 的值；若 t=5 則為 D5 的值。

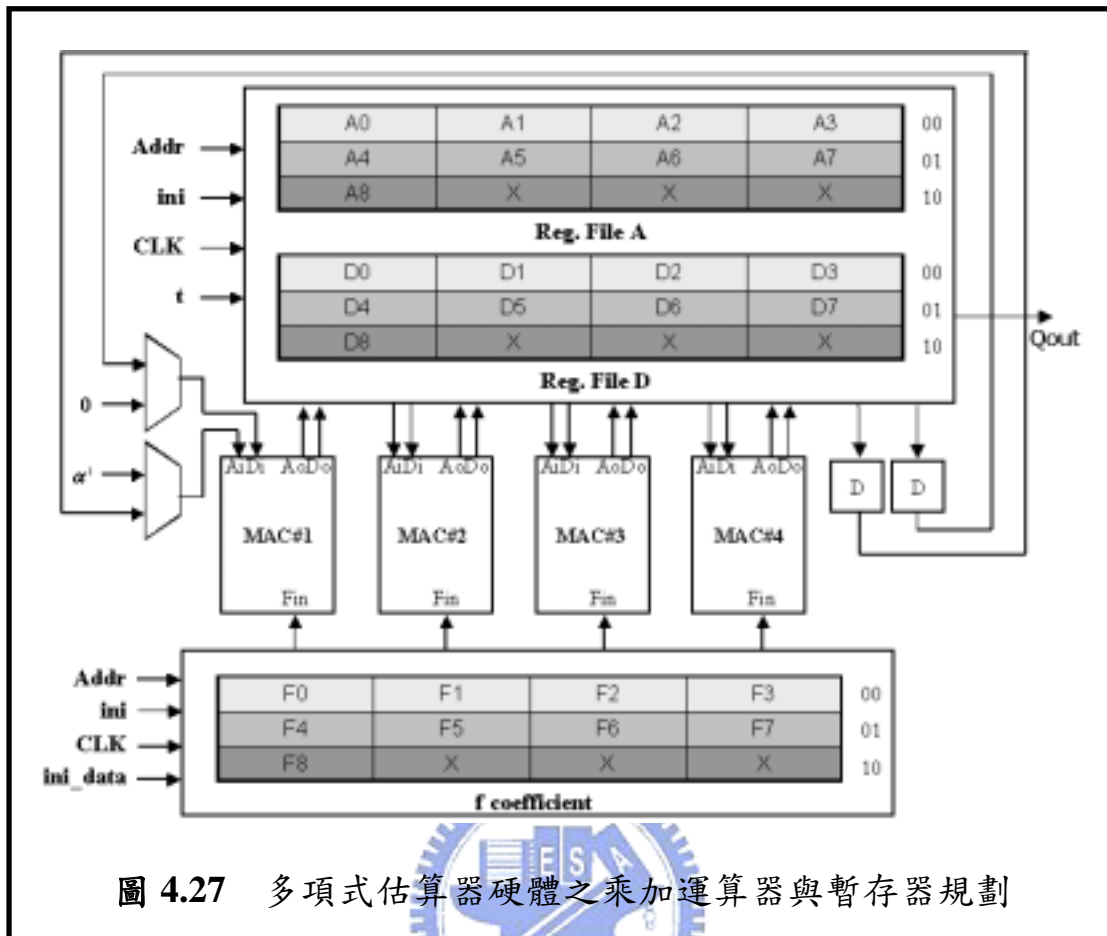


圖 4.27 多項式估算器硬體之乘加運算器與暫存器規劃

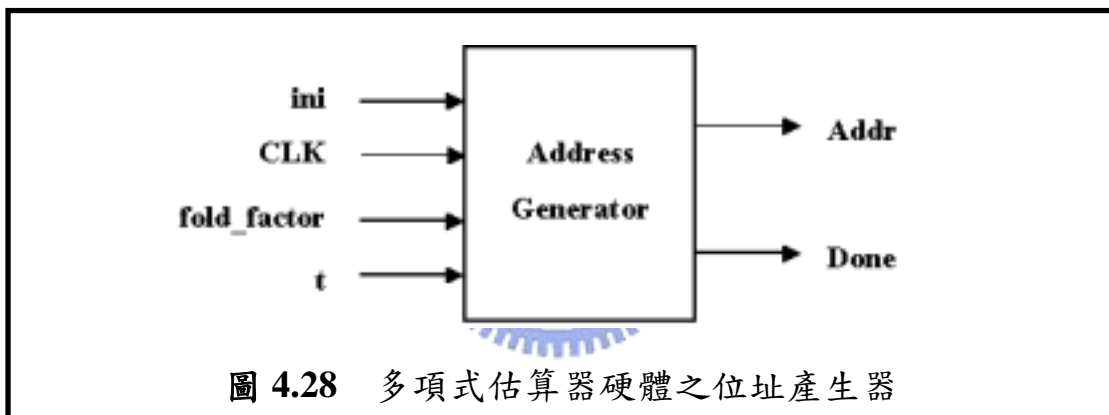
4.7.1.2 位址產生器的設計

在推導完運算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暫存器，就能使得多項式估算硬體正確無誤的運作。

圖 4.28 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以及 RiBM 架構的位址產生器一樣，內部含有兩個計數器，一個依照時脈 (CLK) 將位址訊號 (Addr) 向上遞增計數，當達到摺疊係數 (fold_factor) 上限時則歸零；另一個用來計算結果完成的時間，當計算經過 t+1 回合後，在暫存器檔案

中的第 $t+1$ 個暫存器內所存的值，就是多項式的計算結果，進而提升完成（Done）訊號為高準位。而重置（ini）訊號則是強迫計數的地址訊號歸零。

此外，由於整個多項式估算必須連續將伽羅瓦場內 255 個非零元素代入求值，所以完成訊號在第一個元素代入後的第 $t+1$ 個回合會被提升到高準位，而接下來的連續 255 個回合也會一直保持，之後才回復到低準位，也就是說，暫存器檔案的第 $t+1$ 個暫存器內存的資料，從第 $t+1$ 個回合開始，接著連續的 255 回合中所存入的值就是伽羅瓦場內 255 個元素各自代入的計算結果。



4.7.1.3 多項式估算硬體之輸入輸出定義

圖 4.29 是多項式估算硬體架構的輸入輸出腳位定義，其中係數初始值輸入腳位共有九個，用來設定所要計算之多項式的係數。假設所要計算的多項式為 t 次多項式，則由最高次（ t 次）係數開始，分別從 ini_0 端開始輸入，直到常數項係數為止，依序排列，其於未安排的多項式初始值輸入腳位不予理會，因為不影響計算的結果。

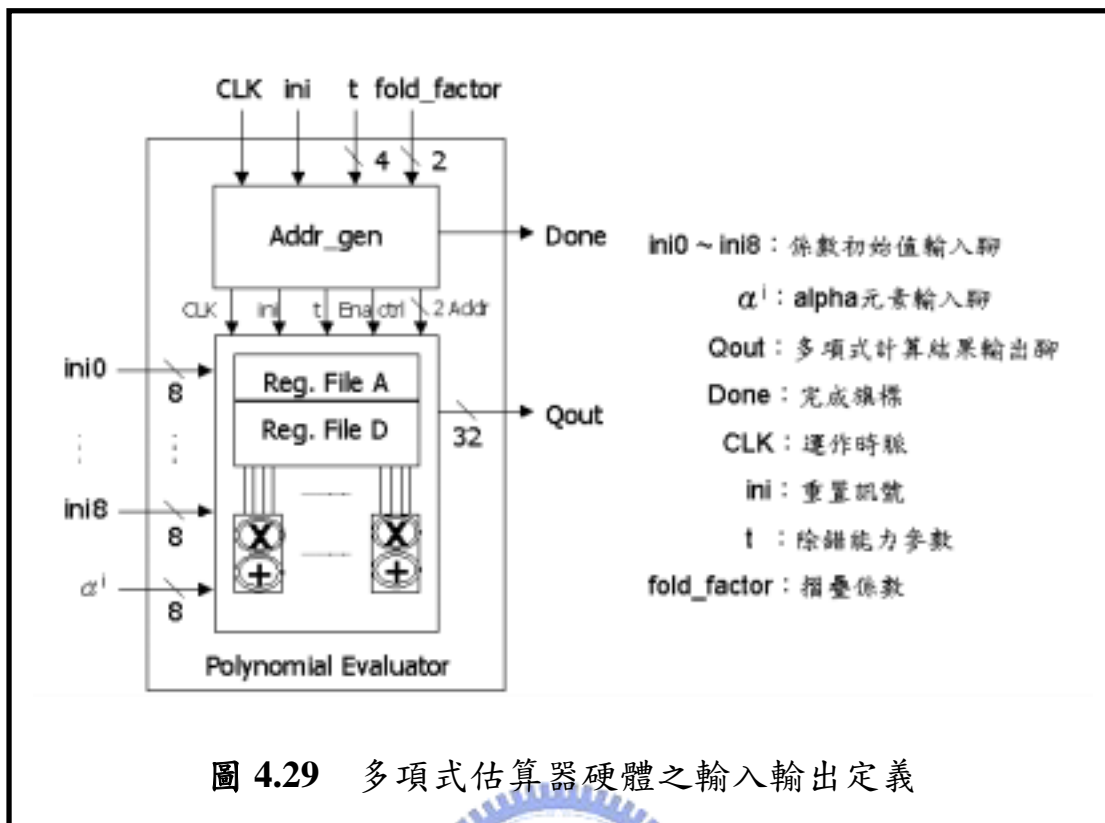


圖 4.29 多項式估算器硬體之輸入輸出定義

4.7.2 伽羅瓦場元素輸出表

里德所羅門解碼中的最後一個錯誤修正步驟，其必須將伽羅瓦場中所有的元素都代入錯誤位置多項式及錯誤大小評估多項式去求值，因此需要一個伽羅瓦場元素的輸出表，在每回合送出一個元素到多項式求值硬體中計算。這個輸出表是以位置定址的方式，送入位址訊號來選擇輸出的伽羅瓦場元素，送入位址 i ，則輸出 α^i 之值。

另一方面，在變形後的佛尼演算法中，必須有一個方塊，特別用來計算 x^{m_0+2t} 的值，由於不同規格的 m_0 及除錯能力 t ，會增加這個方塊在硬體設計上的複雜度，於是在此想到另一個變通的辦法，在先前有提到說伽羅瓦場元素輸出表是以位置定址的方式，所以如果想計算

$(\alpha^i)^{m_0+2t}$ 可以將位址 i 直接乘上 m_0+2t ，所得的值再送到輸出表中尋找相對位置的元素，此元素就是 $(\alpha^i)^{m_0+2t}$ ，這麼一來，將原先在伽羅瓦場中次方的運算，轉換成在實數系上的乘法，不但簡化了運算的流程，對於不同規格的 m_0 及 t 的問題也一併的解決。

將 x^{m_0+2t} 的問題轉換成用查表的方法，使得在參數的變化上可以更有彈性，但是也引申一些問題。圖 4.30 說明了問題所在，與解決的辦法。假設規格 m_0 的典型值為 0 或 1，而 t 值最大為 8，則當 8 位元的位址 i 乘上 5 位元的 m_0+2t 時，會產生 13 位元的位址，這對於只接受 8 位元位址輸入的元素輸出表來說，勢必要作修正，其方法就是對於最後的位址乘積取 255 的模數 (mod 255)，伽羅瓦場元素以 255 個為循環，所以 α^0 等同於 α^{255} ，取模數的結果則會得到相等的元素值。至於取模數的動作，只需要用連續兩次的加法，就可以達到相同的效果，接下來針對這個說法，給予詳盡說明與證明。

證明：

步驟 1. 假設 $a = \text{xxxxx}$ 為 5 位元， $b = \text{yyyyyyyyy}$ 為 8 位元
 則 $\text{xxxxxyyyyyyyy} \bmod 255 = (a \cdot 256 + b) \bmod 255$
 $= (a + b) \bmod 255$

其中 $a + b < 511$

步驟 2. 假設 c 為 1 位元， d 為 8 位元

如果 $255 < a + b < 511$ ，則 $c = 1, d < 255$

$$(a + b) \bmod 255 = (c \cdot 256 + d) \bmod 255$$

$$= c + d \quad 255$$

如果 $a + b < 256$ ，則 $c = 0, d < 255$

$$(a + b) \bmod 255 = (c \cdot 256 + d) \bmod 255$$

$$= c + d = d \quad 255$$

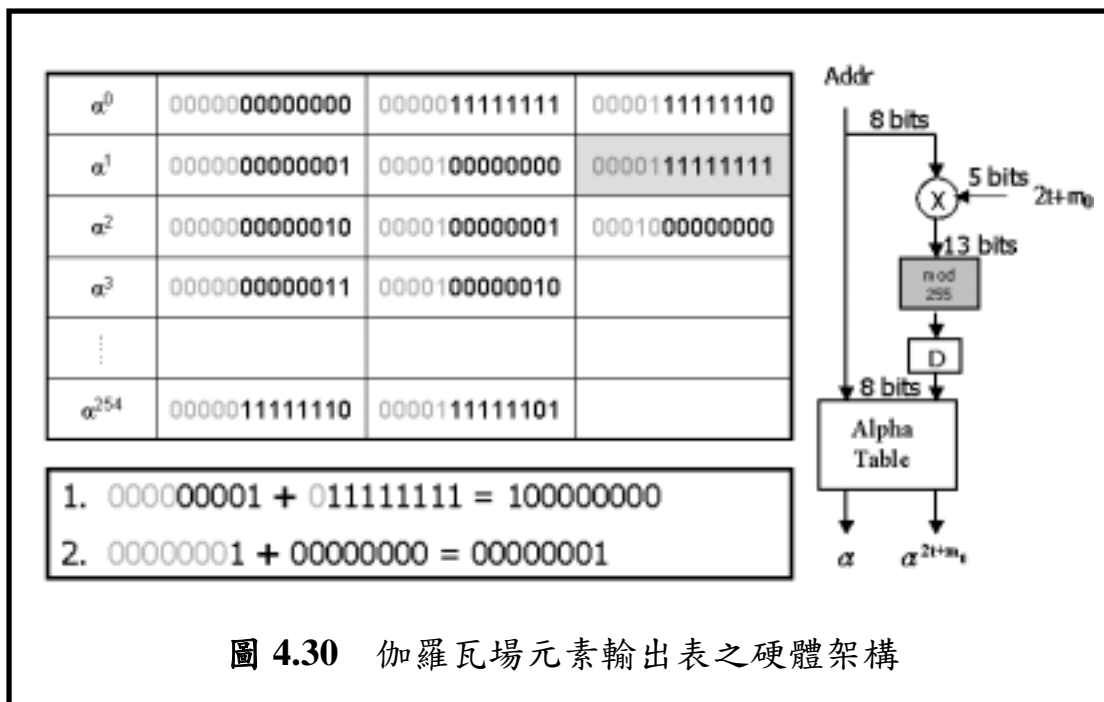


圖 4.30 伽羅瓦場元素輸出表之硬體架構

經過改良後的伽羅瓦場元素輸出表，只需要再配合適當的元素位址產生器，就可以讓元素輸出完美運作。圖 4.31 為此位址產生器的方塊圖。圖中的位址輸出訊號會根據摺疊係數判斷一個回合的時脈週期數，每一回合輸出位址會向上加一。參數 m_0 是為了調整 Chien 尋根法代根的起始元素 α^{m_0} ， m_0 的典型值為 0 或 1，針對這兩個主要的典型值，輸入 m_0 可以用來選擇位址訊號由 0 或是 1 開始遞增，意即由 α^0 或 α^1 開始連續代入 255 個元素值。此外輸入 t 是用來計算整個錯誤修正完成的時間，當錯誤修正後的資料開始輸出時，完成訊號會呈現高準位狀態。關於致能訊號，根據摺疊係數在每回合最後一個時間點致能，這一點將在下一節提出說明。

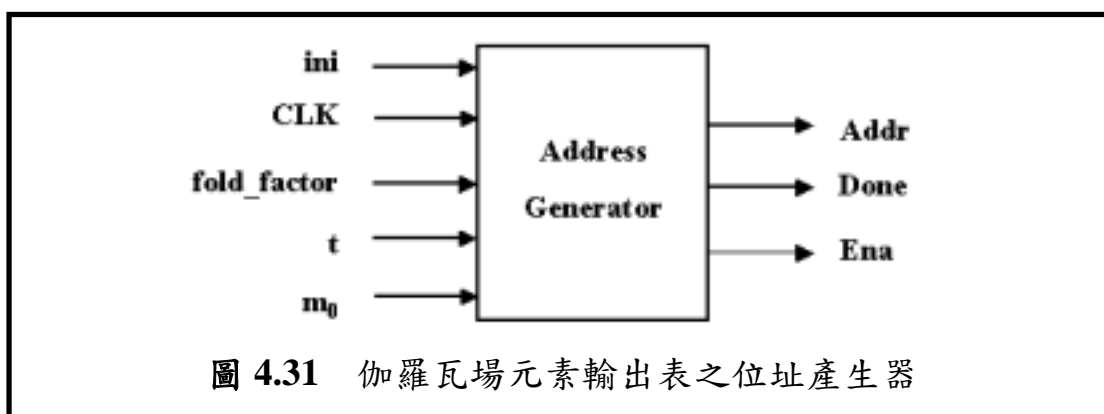


圖 4.31 伽羅瓦場元素輸出表之位址產生器

4.7.3 佛尼演算法

有了先前章節介紹的幾個錯誤修正硬體方塊，接下來就有能力可以進行佛尼演算法的計算，此變形後的佛尼演算法的式子如下：

$$Y_i = - \frac{X_i^{-(m_0+2t-1)} \omega^{(h)}(X_i^{-1})}{\Lambda'(X_i^{-1})} = - \frac{x^{m_0+2t} \omega^{(h)}(x)}{x \Lambda'(x)} \Bigg|_{x=X_i^{-1}} \quad (4.8)$$

配合此佛尼演算法後的錯誤修正器硬體架構如圖 4.32 所示。其中 $\Lambda(x)$ 及 $x \Lambda'(x)$ 方塊均為 t 次多項式的估算方塊，因此延遲時間 (Latency) 為 $t+1$ 個回合，而 $\omega^{(h)}(x)$ 方塊是 $t-1$ 次多項式估算方塊，延遲時間為 t 個回合。至於 x^{m_0+2t} 方塊則將伽羅瓦場元素表的 $(\alpha^i)^{m_0+2t}$ 輸出值延遲為 t 個回合送出，以配合其他方塊運算。此外 x^{-1} 方塊主要是作倒數的運算，這裡是採用查表的方法，依照輸入的值進而查表輸出該值的倒數。另外，圖中在訊號傳遞路徑上的標示數值，表示元素從進入方塊後到流經該路徑所需要的回合數。整個錯誤修正的過程，在多工器經由錯誤位置多項式的計算結果選擇是否為錯誤位置，進一步將儲存在接收訊號緩衝器內的儲存資料，去加上佛尼演算法計算出的錯誤大小作修正，或是加上零不作變動。

此外，在這整個架構中為了管線化所增加的暫存器，每個回合會被致能觸發一次，因為整體的方塊動作是以回合數為基準在流動，所以自元素進入多項式估算方塊後，在第 $t+3$ 個回合就可以對該元素相對於接收訊號的相對位置進行修正，當然接收訊號緩衝器也是依照每回合送出一筆資料的步調來輸出。而前一節中提到伽羅瓦場元素輸出表位址產生器中的致能訊號，就是根據摺疊係數在每回合的最後一個時間點將訊號致能，使得管線化的暫存器能夠被致能觸發，讓資料向前流動。

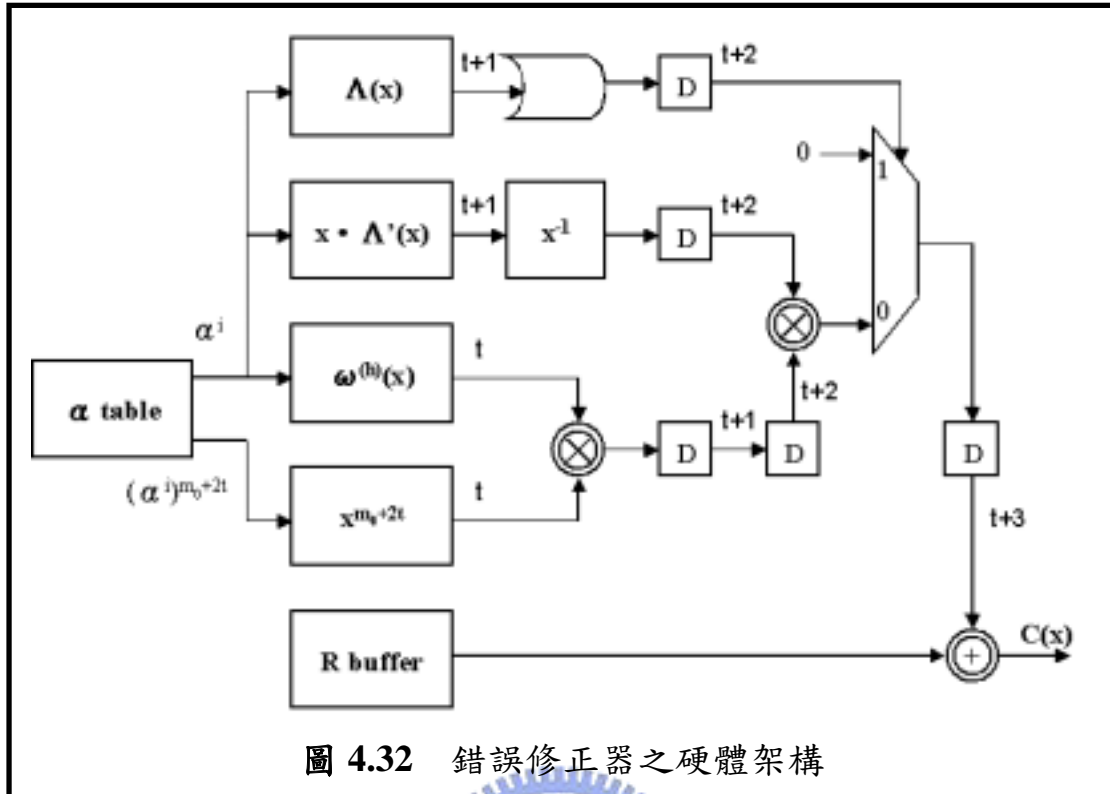
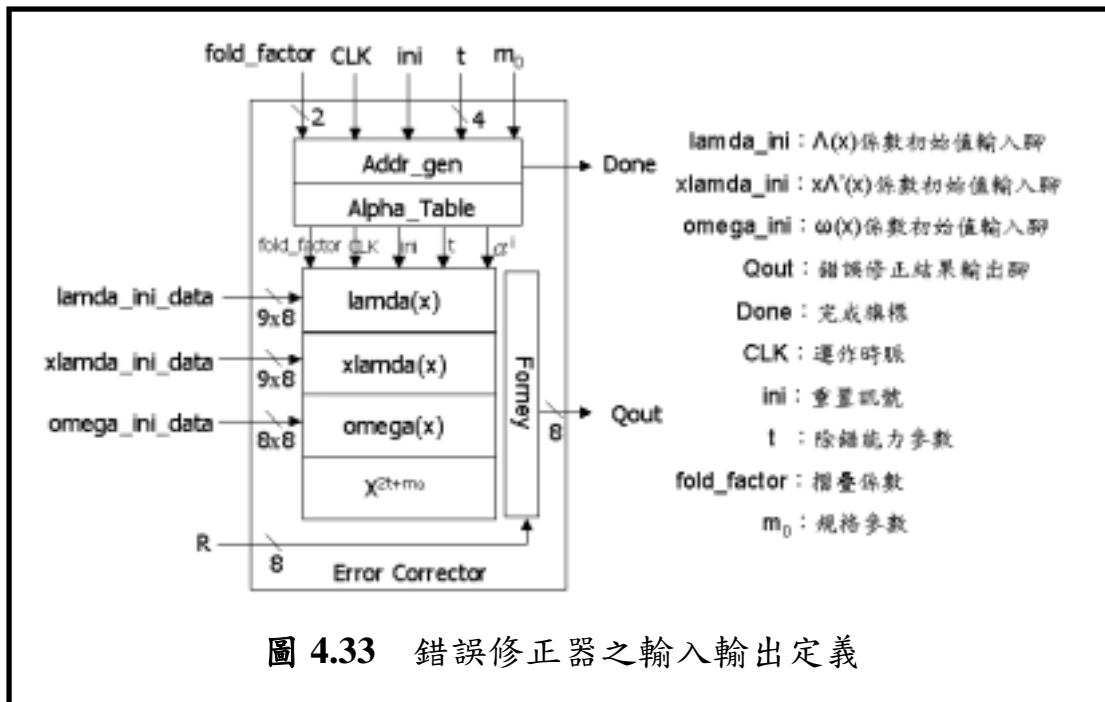


圖 4.32 錯誤修正器之硬體架構

4.7.4 錯誤修正器之輸入輸出定義

錯誤修正器的輸入腳位除了包括重置訊號 (ini)、時脈 (CLK)、摺疊係數 (fold_factor)、錯誤修正能力參數 (t) 與規格參數 m_0 ，另外還有從接收信號緩衝器傳送過來的接收訊號 (R)，以及三組多項式估算的係數初始設定值，其中 $\Lambda(x)$ 及 $x \cdot \Lambda'(x)$ 兩多項式以除錯能力最高為 8 來估算，則各需要九個八位元的係數初始值設定腳位， $\omega^{(h)}(x)$ 多項式則需要八個八位元的初始值設定腳位。至於輸出腳位則包括錯誤修正完成的資料輸出腳位 (Qout)，還有告知修正完成及輸出資料為有效值的完成訊號 (Done)，在其為高準位時表示輸出資料為有效。圖 4.33 對於此方塊的輸入輸出接腳，以圖示的方式加以詳細的說明解釋。



4.8 里德所羅門解碼器之狀態機設計

設計整個系統狀態機的目的，是為了讓解碼器中主要的三大運算方塊可以並行工作、同時處理資料，而不至於讓資料流混亂。在開始介紹狀態機之前，首先整理出各方塊所需要的控制信號，以及一些由方塊回傳用以判斷狀態的訊號，全部在此作個說明：

- **syn_ini** :
錯誤症狀計算器之初始化訊號，輸出至錯誤症狀計算方塊。
- **syn_done** :
錯誤症狀計算完成並輸出之旗標訊號，由症狀計算方塊輸入。
- **syn_ready** :
告知 BM 方塊資料已準備完成之訊號。
- **bm_ini** :
BM 方塊之初始化訊號，輸出至 BM 方塊。

- **bm_done** :
BM 方塊計算完成並輸出之旗標訊號，由 BM 方塊輸入。
- **bm_ready** :
告知錯誤修正器資料已準備完成之訊號。
- **bm_fetch** :
告知錯誤症狀計算器已將資料從 S2B 方塊擷取之訊號。
- **ec_ini** :
錯誤修正器之初始化訊號，輸出至錯誤修正方塊。
- **ec_done** :
錯誤修正器完成錯誤修正之旗標訊號，由錯誤修正器方塊輸入。
- **ec_fetch** :
告知 BM 方塊已將資料從 B2P 方塊擷取之訊號。




圖 4.34 即為里德所羅門解碼器之狀態流程圖。在整個解碼器重置後，每個狀態機就進入第一個狀態中，此時錯誤症狀計算器的狀態為初始化，此刻狀態機會將 **syn_ini** 準位提高，讓錯誤症狀計算方塊重置。接下來在一個時脈後進入計算執行狀態，直到接收 **syn_done** 高準位的告知訊息，表示開始將結果輸出，當 **syn_done** 準位又降低，代表資料輸出完畢，此時告知 BM 方塊來擷取，等擷取後就又回到初始化狀態。

BM 狀態機重置後會呈現發呆狀態，此時狀態機會送出 **bm_ini** 訊號到 BM 方塊，使其發呆等待，直到 **syn_ready** 高準位時，則進入初始化狀態，並告知錯誤症狀計算狀態機已將資料擷取。下一個時脈就進入執行狀態，當 **bm_done** 為高準位時開始將結果輸出，直到降為低準位才完畢，並傳送訊號告訴錯誤修正器資料已準備就緒，等資料被抓取後又回到發呆狀態。

而錯誤修正器在重置後也會呈現發呆狀態，此時會一直對錯誤修正器傳送初始化的訊號，直到接收 BM 的資料準備完成的訊號，才真正進入初始化狀態，並送出資料已接收訊號，接著進入執行狀態，當整個方塊計算完成時，會收到 ec_done 的高準位訊號，表示此時開始輸出資料結果，直到訊號又回到低準位，代表資料輸出完畢，而回歸發呆的狀態。

這三個狀態機彼此會互相牽制，讓資料可以循序處理而不會混亂，也可以讓三個主要的計算方塊能夠齊頭並進的計算資料而不至於閒置荒廢，這也是設計此狀態機最重要的目的及原因。此外，每當整個解碼器重置時，只需花費一個時脈的時間，將狀態機重設進入第一個狀態就可以完成。

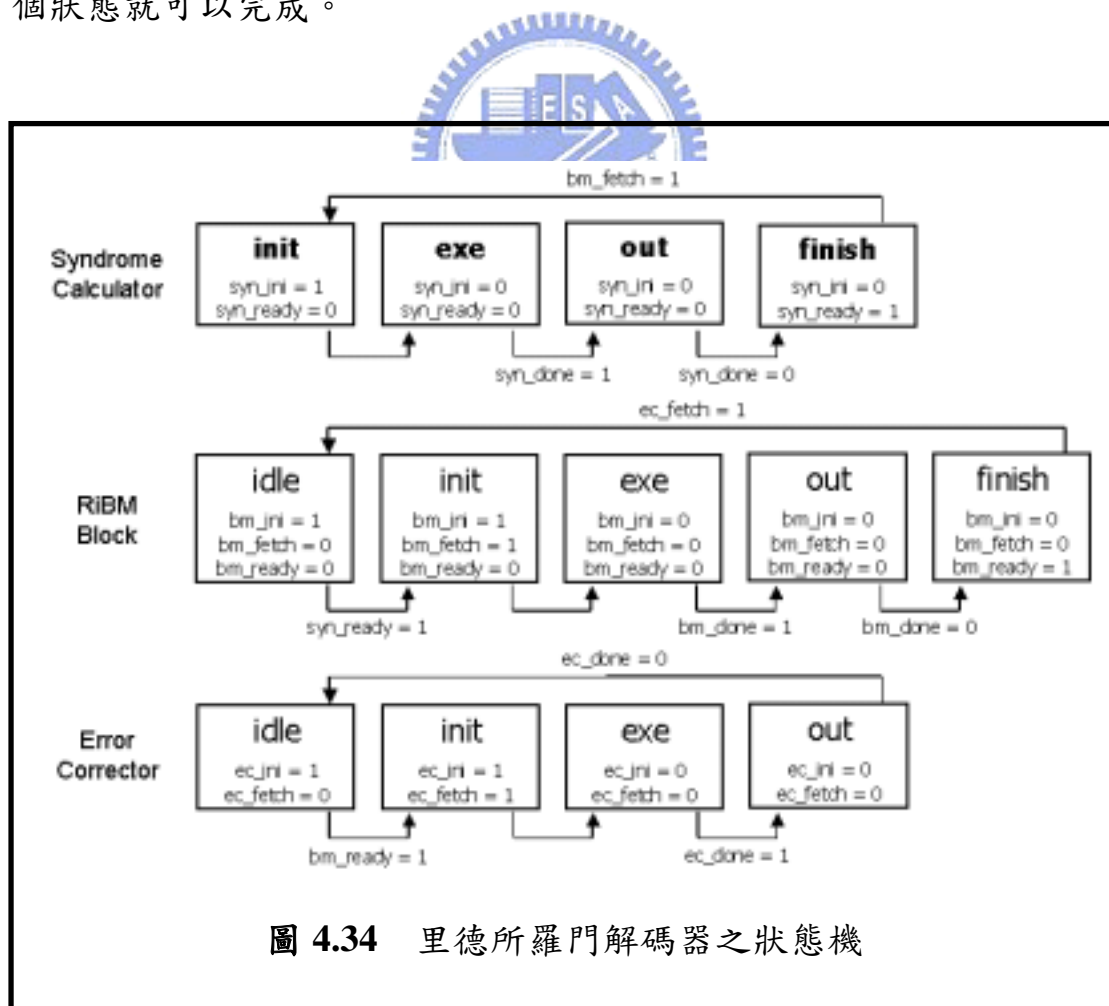


圖 4.34 里德所羅門解碼器之狀態機

4.9 實現結果

在本篇論文 4.1 節設計的合成器中有提到所合成的 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)。而三大主要的計算方塊在完成一組字碼的計算，各自所需要的延遲時間為：

- 錯誤症狀計算： $255 \times syn_fold \times clk$
- RiBM 計算方塊： $2t \times bm_fold \times clk$
- 錯誤修正方塊： $(t+3+255) \times poly_fold \times clk$

其中 fold 參數是各方塊在該除錯能力 t 下所需的摺疊係數。

接下來利用合成器所產生之 RTL 程式以軟體 Modelsim 進行模擬測試，其測試數據列於附錄，而所設定的一些規格如下：

- 合成器輸入規格：最大除錯能力參數 t 為 8
三大計算方塊使用運算單元數皆為 4
- 模擬測試輸入值：t=6, m₀=1

1. 錯誤症狀計算器 (Syndrome Calculator)

$$\begin{aligned} S_0 &= r(\alpha^1) = \alpha^{167} = 0x7E & S_6 &= r(\alpha^7) = \alpha^{172} = 0xDE \\ S_1 &= r(\alpha^2) = \alpha^{200} = 0x38 & S_7 &= r(\alpha^8) = \alpha^{229} = 0x5E \\ S_2 &= r(\alpha^3) = \alpha^1 = 0x40 & S_8 &= r(\alpha^9) = \alpha^{252} = 0xB5 \\ S_3 &= r(\alpha^4) = \alpha^{120} = 0xDC & S_9 &= r(\alpha^{10}) = \alpha^{157} = 0xAB \\ S_4 &= r(\alpha^5) = \alpha^{159} = 0xCE & S_{10} &= r(\alpha^{11}) = \alpha^{89} = 0x87 \\ S_5 &= r(\alpha^6) = \alpha^{148} = 0x4A & S_{11} &= r(\alpha^{12}) = \alpha^{207} = 0x65 \end{aligned}$$

從圖 4.35 中可以看出一個回合為三個時脈週期，這是當 $t=6$ 時摺疊係數是 3 的緣故。在最後一個回合中輸出的十二個錯誤症狀的值，跟原先經過程式計算的相同。

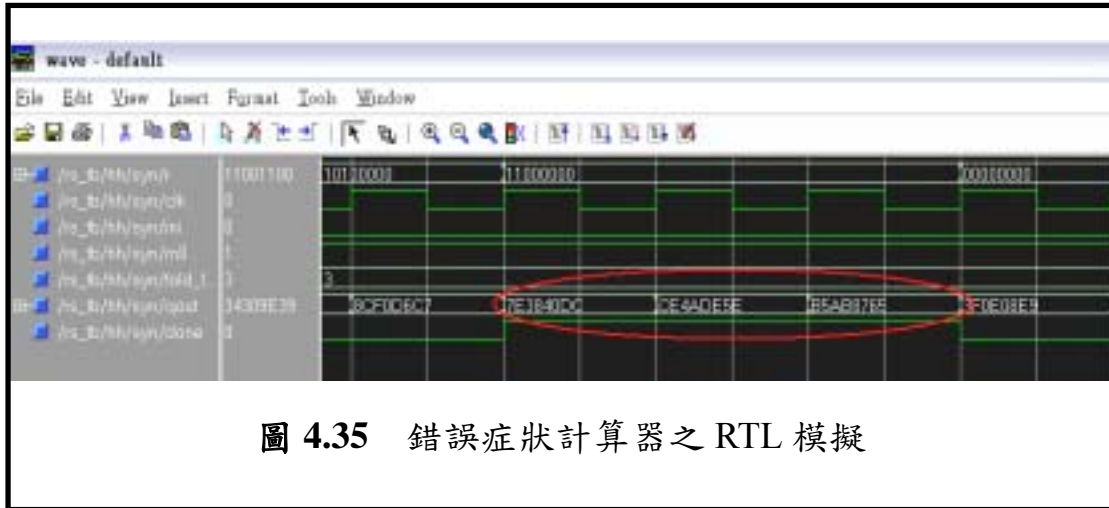


圖 4.35 錯誤症狀計算器之 RTL 模擬

2. BM 方塊 (BM Block)



$$\Lambda_6 = 0$$

$$\omega_5^{(h)} = 0$$

$$\Lambda_5 = 0$$

$$\omega_4^{(h)} = 0$$

$$\Lambda_4 = \alpha^{246} = 0xF3$$

$$\omega_3^{(h)} = \alpha^{198} = 0xE0$$

$$\Lambda_3 = \alpha^{116} = 0x1F$$

$$\omega_2^{(h)} = \alpha^{195} = 0x26$$

$$\Lambda_2 = \alpha^{150} = 0xAA$$

$$\omega_1^{(h)} = \alpha^{86} = 0x8D$$

$$\Lambda_1 = \alpha^{245} = 0x97$$

$$\omega_0^{(h)} = \alpha^{236} = 0xD3$$

$$\Lambda_0 = \alpha^{237} = 0xD1$$

在圖 4.36 中可以發現，RiBM 疊代演算法的硬體架構在最後一個回合的輸出裡，從前面開始的第 $t+1$ 個值起，連續 $t+1$ 個值即為錯誤位置多項式係數，再下來的 t 個值是錯誤大小評估多項式係數。

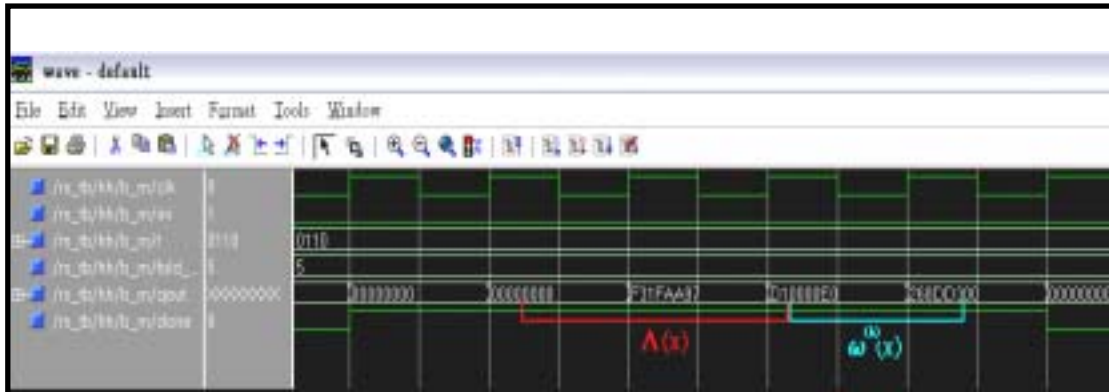


圖 4.36 RiBM 方塊之 RTL 模擬

3. 錯誤修正器 (Error Corrector)

圖 4.37 為錯誤修正器的模擬，圖中的 r 訊號為接收信號緩衝器所送出的接收信號值，qout 為修正後的正確值。由此可見，含有原本含有錯誤的一組字碼已被完整的修正復原。

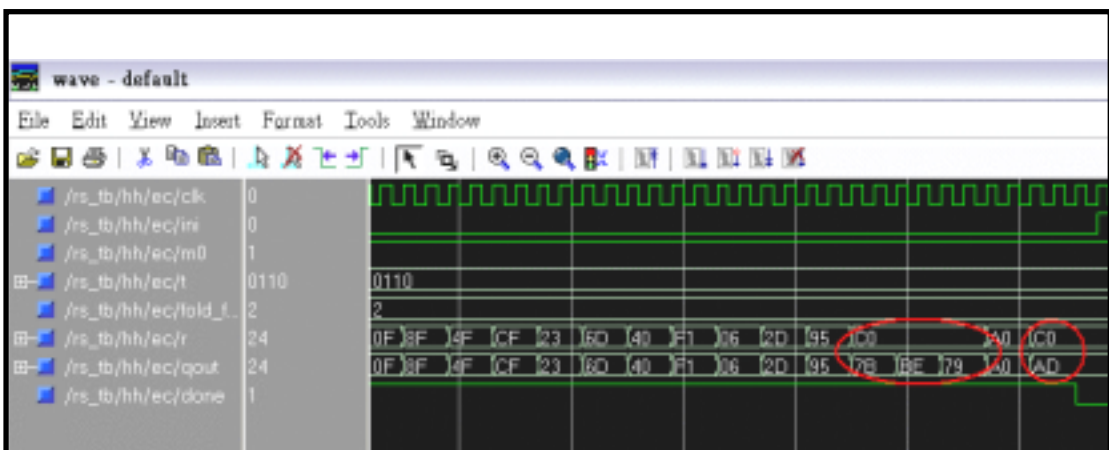
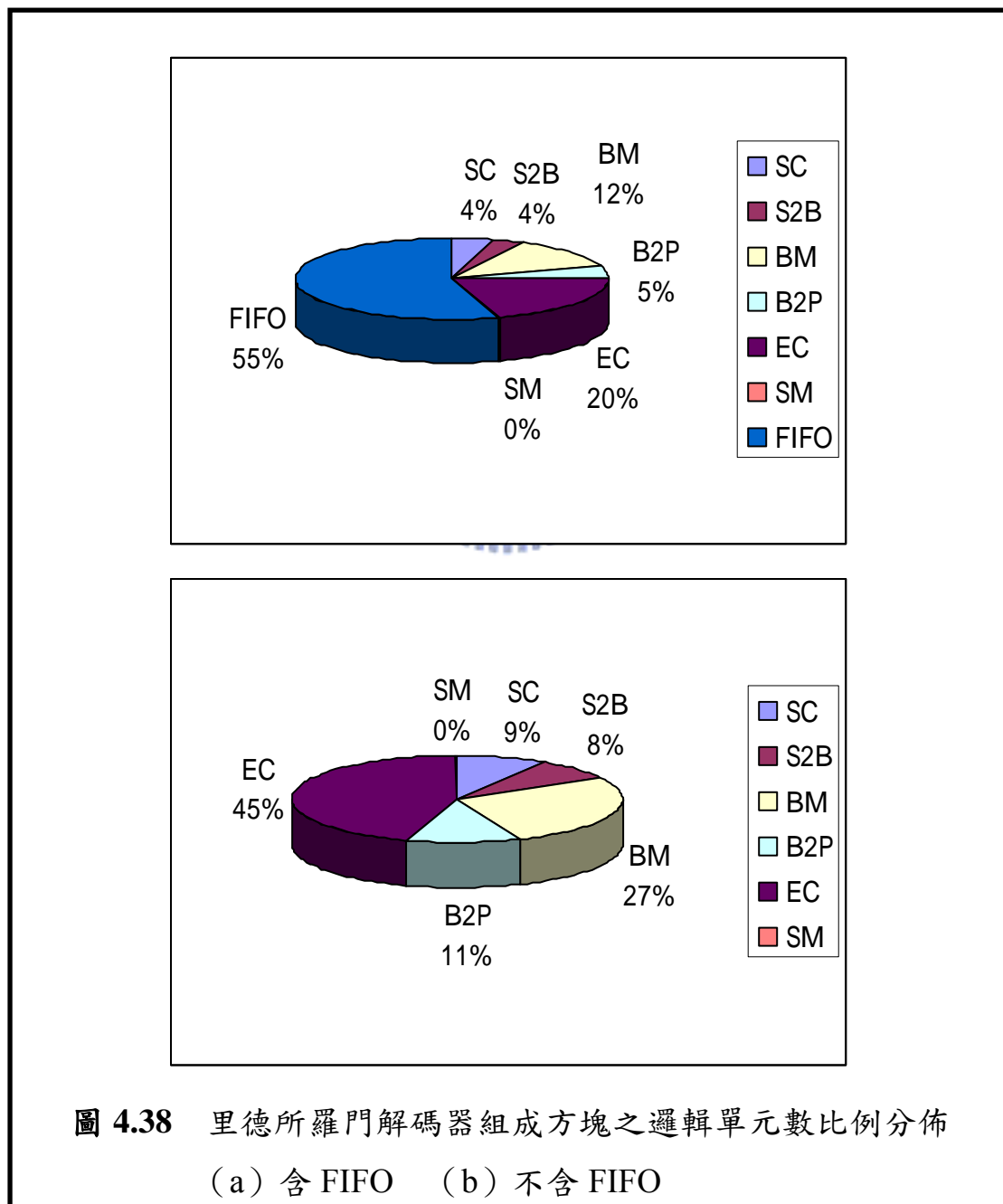


圖 4.37 錯誤修正器之 RTL 模擬

確定所設計之硬體在功能上沒有問題之後，將合成器所產生的硬體語言在 Altera 所提供之設計工具 Quartus II 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。在初略的估計之下，其總邏輯單元（Logic Element）數為 15326，其中一個邏輯單元近似於 12 個邏輯閘數，而各組成方塊之邏輯單元數列於表 4.1，其所佔有之百分比在圖 4.38 中清楚的表示。



Unit	SC		S2B	BM		B2P	EC				FIFO	SM
Percentage of Unit	M&R	Addr		M&R	Addr		Chien	Forney	Poly			
	97.3%	2.7%		98.8%	1.2%		22.3%	11.1%	M&R	Addr		
									66%	0.6%		
Logic Element	600		524	1838		733	3018				8152	20

表 4.1 里德所羅門解碼器組成方塊之邏輯單元數

由各組成方塊之邏輯單元數比例分佈圖可以發現，接收信號緩衝器佔有超過一半的總邏輯單元數，這是因為緩衝器在此是以 D 型正反器的方式合成，其總共必須儲存 3x255x8 位元的資料，一般在業界會使用高密度的儲存裝置來取代這一個記憶功能的方塊，所以為了更仔細的分析，在比例分配的計算上，採取包含此方塊及不包含此方塊兩種型式呈現。

在邏輯單元數列表中，M&R 代表計算方塊內部所包含的運算器及暫存器，而 Addr 則表示位址產生器的部分，這兩者於三大計算方塊中所涵蓋的比率，有很大的差距，為了能夠動態調整不同的除錯能力值所設計的位址產生器，只佔有整體計算方塊的些微比例，另外再加上資料傳遞用途的 S2B 及 B2P 兩個方塊，若不將緩衝器計算在內，大約是總邏輯單元數的 20% 以下，所增加的這些電路邏輯，換取了可動態變換的規格參數 t。

接下來在 APEX EP20K1500EBC652-1X 之可程式化邏輯陣列中進行驗證，驗證方法是先用軟體程式編碼計算 t = 2、4、6、8 之 RS(255,251)、RS(255,247)、RS(255,243)、RS(255,239) 各組字碼 (codeword)，接著於各組字碼中任意挑選小於 t 個記號 (symbol) 並修改內容值，之後將其全部存入實驗板的內建 ROM 模組中，這些字碼的資料輸出會根據 t 值來選擇。最後將佈局與繞線完成的里德所

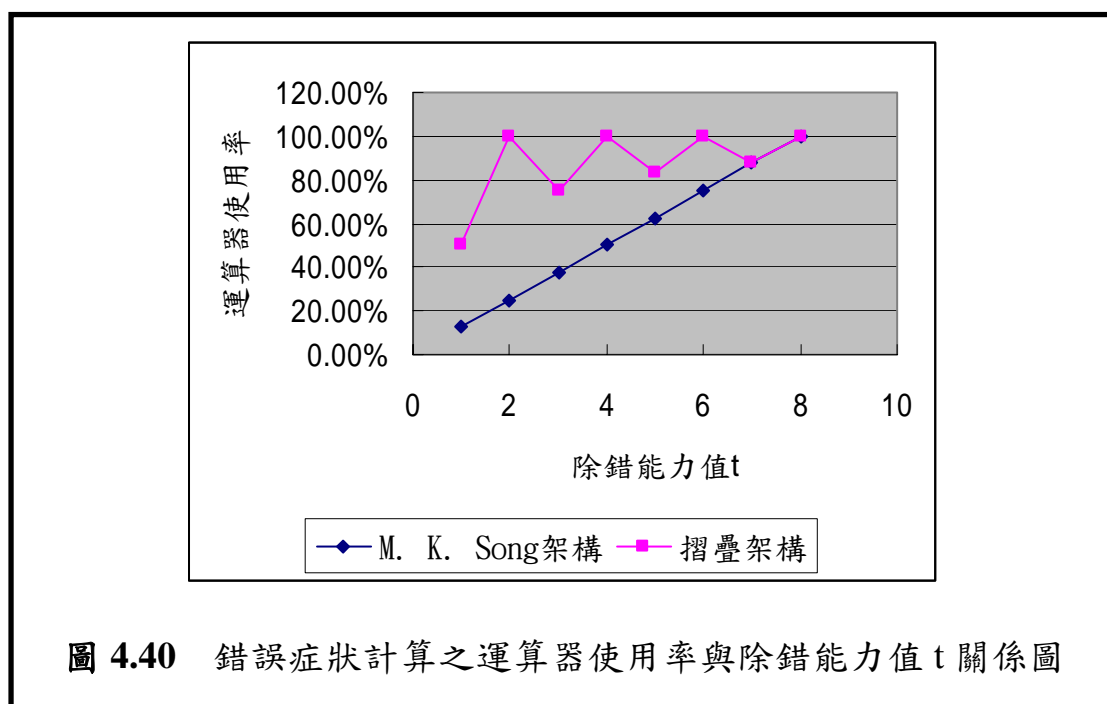
羅門解碼硬體配合 ROM 模組，在輸入不同 t 值的情況下，檢視解碼輸出的結果是否修正先前字碼中記號 (symbol) 被修改的內容，測試結果四組字碼均正確完成修正工作。圖 4.39 為模擬測試驗證環境。



圖 4.39 模擬測試驗證環境

2002 年 M. K. Song [12] 提出 n 與 t 兩參數可變的里德所羅門解碼器硬體，為了達成 n 參數可變，其尋根硬體架構的特殊設計會出現解碼前的輸入是由一個字碼組的最高位字碼開始到最低位字碼，但解碼後的輸出卻是反相，由最低位字碼到最高位字碼，由於這個改變，所以必須多加一個使字碼位置前後顛倒的硬體，其結果才可以再被超高速數位用戶線路的下一級所採用，而要完成這個動作，必須等到所有的字碼都已經準備就緒，才能一次把位置調換過來，這似乎會把整個解碼的時間拖垮。此外，其硬體是根據最大除錯能力參數 t 所需之最多運算單元個數來實現，在處理非最大除錯能力參數時，一些不需使用的運算單元依舊會運作，只是其計算的結果所要存入的暫存器是被除能而不存入，如此一來，會造成在功率上不必要的消耗，而且所處理資料的除錯能力參數越小，會有越多的運算單元作不必要的運算。

對於超高速數位用戶線路應用來說，里德所羅門解碼器需要的 n 規格是固定為 255，而除錯能力參數 t 是 1 到 8 之間的可變範圍，本篇論文所提出的摺疊架構完全符合其需求，且其處理資料是依據運算單元個數為一個單位來分時運算，因此平均來說，硬體的使用率較高，也產生較少的功率浪費。圖 4.40 為摺疊架構與 M. K. Song 所提架構中錯誤症狀計算方塊之運算器使用率與除錯能力參數值之間的關係，其中除錯能力最大為 8，而摺疊架構是採用四個運算單元。根據圖中顯示，M. K. Song 的架構隨著除錯能力值的變化，硬體使用率有很大的變動，當除錯能力值越小，使用率越低；而本篇論文所提出的架構，使用率比較平均而且比較高。



第五章

結論

5.1 主要貢獻



本篇論文提出一套硬體架構之摺疊演算法，此演算法可應用於各種採用基本運作單元（PE）所構成之陣列架構，其具備可重複使用的特性，特別適合用在陣列的基本運作單元個數會因規格而變動的應用上。將此摺疊演算法套用在里德所羅門解碼的設計上，使其成為一套可重複規劃之里德所羅門解碼器，可以動態調整除錯能力參數，而且也減少了運算單元使用個數與整體架構的面積，使硬體具有較高的使用率，對於非對稱數位用戶線路（ADSL）與超高速數位用戶線路（VDSL）這類型需要動態調整里德所羅門解碼器的除錯能力參數之應用，有極大的貢獻。

此外論文中還提供一套可讓使用者輸入參數之合成器，使用者根據需要，可以設定最大除錯能力參數，以及解碼器中各方塊所需採用的運算單元個數，而合成器便會自動產生整個里德所羅門解碼器之

VHDL 硬體語言，其皆為可合成 (Synthesizable) 之 RTL 程式。這麼一來，在各種需要里德所羅門解碼器的應用中，均可以依據其規定之處理速率估算運算單元需要的個數，以及選擇最大的除錯能力參數，進而使用此合成器來產生適合於該應用之里德所羅門解碼架構。

5.2 未來展望

- 發展具備使用者介面之 IP 合成器

本篇論文設計的合成器僅有簡陋的輸入介面，對於所合成產生的硬體架構只能一成不變的使用固定數目的運算器，未能動態選擇運算器使用的數量，而且規格參數 m_0 只能有 0 或 1 的選擇，另外也必須限定於 $m=8$ 的 $GF(2^8)$ 之中，在使用者的角度看來，並沒有充份的利用空間，因此可以增添一些參數規格及硬體使用的操作範圍，附加更多的彈性。

- 加入消除 (Erasure) 的處理

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。

從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：

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

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。

● 降低功率的浪費

本篇設計的摺疊架構中，在每回合的最後一個時間點難免會有一些運算單元是不必運算的，但架構中沒有關閉運算器計算的機制，導致功率的消耗，故可以在硬體設計中加入運算器自動調整計算開啟或關閉的切換開關，以節省電力的浪費。

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] S. Y. Kung, “VLSI Array Processors,” Prentice-Hall, Inc. published, 1988.
- [4] Keshab K. Parhi, “VLSI Digital Signal Processing Systems: Design and Implementation,” Wiley-Interscience, Inc. published, 1998.
- [5] Lijun Gao and Keshab K. Parhi, “Custom VLSI Design of Efficient Low Latency and Low Power Finite Field Multiplier for Reed-Solomon Codec,” The 2001 IEEE International Symposium on Circuits and Systems, 2001. ISCAS 2001, Vol. 4, Page: 574-577, 6-9 May 2001.
- [6] Surendra K. Jain, Leilei Song, and Keshab K. Parhi, “Efficient Semisystolic Architectures for Finite-Field Arithmetic,” IEEE Transactions on Very Large Scale Integration System, Vol. 6, No. 1, pp. 101-113, March 1998.
- [7] Khaled M. Elleithy and Magdy A. Bayoumi, “A Systolic Architecture for Modulo Multiplication,” IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 42, No. 11, pp. 725-729, November 1995.
- [8] Chin-Liang Wang and Jung-Lung Lin, “Systolic Array Implementation of Multipliers for Finite Field $GF(2^m)$,” IEEE Transactions on Circuits and Systems, Vol. 38, No. 7, pp. 796-800, July 1991.

- [9] R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Polynomial GCD Computation," Tech. Rep., Carnegie-Mellon University, Computer Science Department, May 1982.
- [10] 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.
- [11] Dilip V. Sarwate and Naresh R. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders," IEEE Transactions on Very Large Scale Integration Systems, Vol. 9, No. 5, pp. 641-655, October 2001.
- [12] I. S. Reed, M. T. Shih, and T. K. Truong, "VLSI design of inverse-free Berlekamp-Massey Algorithm," IEE Proceedings-E Vol. 138, No. 5, pp. 295-298, September 1991.
- [13] Jin-Chuan Huang, Chien-Ming Wu, Ming-Der Shieh, and Chien-Hsing Wu, "An Area-Efficient Versatile Reed-Solomon Decoder for ADSL," Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, 1999. ISCAS '99, Vol. 1, Page: 517-520, 6-9 May 1999.
- [14] Mook Kyou Song, Eung Bae Kim, Hee Sun Won, and Min Han Kong, "Architecture for Decoding Adaptive Reed-Solomon Codes with Variable Block Length," IEEE Transactions on Consumer Electronics, Vol. 48, No. 3, pp. 631-637, August 2002.
- [15] Huai-Yi Hsu and An-Yeu Wu, "VLSI Design of A Reconfigurable Multi-mode Reed-Solomon Codec for High-Speed Communication Systems," IEEE Asia-Pacific Conference on ASIC, pp. 359-362, August 2002.

附錄

本論文模擬測試數據之 $R_i = \alpha^j$ 對應表整理如下：

i	j	i	j	i	j	i	j
0	25/42	32	90	64	162	96	46
1	50	33	62	65	65	97	137
2	25/137	34	204	66	109	98	32
3	25/243	35	187	67	71	99	35
4	25/62	36	177	68	83	100	217
5	135	37	134	69	57	101	146
6	20	38	96	70	60	102	68
7	30	39	251	71	132	103	17
8	24	40	170	72	158	104	124
9	1	41	85	73	93	105	180
10	93	42	157	74	42	106	184
11	183	43	41	75	20	107	38
12	233	44	82	76	171	108	119
13	213	45	59	77	211	109	153
14	174	46	161	78	86	110	165
15	79	47	108	79	242	111	227
16	215	48	246	80	97	112	24
17	44	49	111	81	190	113	254
18	117	50	12	82	252	114	197
19	122	51	127	83	220	115	49
20	235	52	236	84	178	116	237
21	22	53	73	85	151	117	222
22	245	54	23	86	135	118	74
23	11	55	196	87	144	119	103
24	81	56	118	88	205	120	13
25	160	57	164	89	207	121	99
26	169	58	123	90	188	122	128
27	156	59	183	91	149	123	140
28	176	60	216	92	91	124	247
29	95	61	67	93	209	125	192
30	89	62	45	94	63	126	112
31	203	63	31	95	55	127	7

i	j	i	j	i	j	i	j
128	87	160	64	192	166	224	113
129	167	161	70	193	114	225	76
130	243	162	56	194	228	226	8
131	115	163	131	195	77	227	200
132	172	164	92	196	120	228	248
133	229	165	19	197	9	229	105
134	212	166	210	198	154	230	193
135	78	167	241	199	201	231	28
136	43	168	189	200	185	232	129
137	121	169	219	201	249	233	239
138	21	170	150	202	39	234	141
139	10	171	143	203	106	235	52
140	159	172	206	204	125	236	14
141	155	173	148	205	194	237	224
142	94	174	208	206	181	238	100
143	202	175	54	207	29	239	4
144	61	176	136	208	69	240	75
145	186	177	34	209	130	241	199
146	133	178	145	210	18	242	104
147	250	179	16	211	240	243	27
148	84	180	179	212	218	244	238
149	40	181	37	213	142	245	51
150	58	182	152	214	147	245	223
151	107	183	226	215	53	247	3
152	110	184	253	216	33	248	198
153	126	185	48	217	15	249	26
154	72	186	221	218	36	250	50
155	195	187	102	219	225	251	2
156	163	188	98	220	47	252	25
157	182	189	139	221	101	253	1
158	66	190	191	222	138	254	0
159	30	191	6	223	5		