

國立交通大學

電信工程學系碩士班

碩士論文

應用於正交多頻通訊系統之
具成本效益傅立葉轉換處理器設計

Cost-effective Fast Fourier Transform Processor Design for
Orthogonal Multi-carrier Communication Systems

研究生：賴昭宏

指導教授：紀翔峰 博士

中華民國九十三年七月

應用於正交多頻通訊系統之
具成本效益傅立葉轉換處理器設計

Cost-effective Fast Fourier Transform Processor Design for Orthogonal
Multi-carrier Communication Systems

研究生:賴昭宏

Student:Zhao-Hong Lai

指導教授:紀翔峰 博士

Advisor: Dr. Hsiang-Feng Chi

國立交通大學

電信工程學系碩士班



A Thesis

Submitted to Department of Communication Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

July 2004

Hsinchu, Taiwan

中華民國九十三年七月

應用於正交多頻通訊系統之 具成本效益傅立葉轉換處理器設計

研究生:賴昭宏

指導教授:紀翔峰 博士

國立交通大學

電信工程學系碩士班

中文摘要



隨著通訊及訊號處理技術的快速進步，以 OFDM 調變技術為主的傳輸系統越來越普遍。一些有線的傳輸系統如非對稱式數位用戶端迴路系統(ADSL)和高速數位用戶端迴路系統(VDSL)，及無線的傳輸系統如數位廣播系統(DAB)、數位視訊地面廣播系統(DVB-T)和 IEEE 802.11a/g 等系統，皆是以 OFDM 技術為傳輸標準。這些屬於正交分頻多工的通訊系統皆需用 FFT/IFFT 來實現，所以如何設計一個高效率低成本的快速(反)傅立葉轉換(FFT/IFFT)處理器一直是受注意的問題。

由於迫切的傳輸寬頻需求，先進的 OFDMA 寬頻無線系統(如 IEEE 802.16a)皆採取多點數之 FFT/IFFT 轉換，使得硬體面積也隨著點數而遞增。目前在無線及有線傳輸系統上，OFDM 技術已廣泛地被應用，然而，這些系統平台上所需運算的 FFT/IFFT 點數不盡相同，所以若能設計一套可變長度的多模 FFT/IFFT 處理器，其 OFDM 技術的應用將會更有彈性。

本論文將實現一個以 memory-based 架構為基礎的多模式可變長度之

RFFT(Real valued FFT)/HS-IFFT(Hermitian Symmetric IFFT)處理器。此架構把 RFFT 和 HS-IFFT 實現於同一平台上，且由於 memory-based 架構最少只需一個 butterfly 運算器即可完成全部轉換的 butterfly 運算，故我們可以實現一個具成本效益的處理器。更進一步地，我們藉由位址產生器的設計，使得處理器具有可變長度的功能。對於 FFT 轉換序列為實數序列或 IFFT 轉換序列為 Hermitian Symmetric 序列，則此處理器有更有效率的處理方式。最後，為了在不增加字元長度的狀況下改善 fixed point 的精確度問題，使用了 block scaling 的方式來改善精確度。



Cost-effective Fast Fourier Transform Processor Design for Orthogonal Multi-carrier Communication Systems

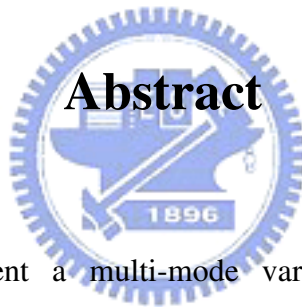
Student: Zhao-Hong Lai

Advisor: Dr. Hsiang-Feng Chi

Department of Communication Engineering

National Chiao Tung University

Hsinchu, Taiwan



In this thesis, we implement a multi-mode variable length RFFT(Real-value FFT)/HS-IFFT(Hermitian Symmetric IFFT) processor which based on memory-based architecture. Because there is only one butterfly process element in our hardware design, the architecture is area-efficient. By designing the address generators, the processor has the variable length character. This architecture has efficient computation for real-value FFT and Hermitian Symmetric IFFT. In order to improve the fixed point precision without increasing the word length, block scaling method is used.

致謝

本篇論文得以順利完成，首先要感謝的是我的指導教授紀翔峰博士。在我論文遇上問題或瓶頸時，紀老師總是可以適時地提供我一個解決的方向，讓我論文得以明確地且不迷失方向地進行下去。除此之外，老師對於學生在獨立思考與分析問題上的培養及教導，讓我受益良多，更使我建立了做研究時所應有的正確態度。

其次，我要由衷的感謝李佳勳、鐘文狀、李昭宏、王志軒及詹謹鴻等五位一起奮鬥的同窗，不管是在課業上、研究上或生活上都給予我諸多的幫助與指導。此外，還得感謝葉桂弘、邱偉茗、謝致遠、沈祐民等一些關心我的同學，及實驗室的學弟妹們，有了各位的支持和相伴，讓我在碩士兩年的生涯中，過得更豐富、更多采多姿，增添了許多生活上的喜悅。

最後，我要感謝一直支持我的父母及姐弟，謝謝父母長久以來對我的栽培與姊弟對我的鼓勵，使我時時感到很窩心。這一切的一切，我都銘記在心，謝謝你們，我最親愛的家人。

目錄：

第一章 簡介	1
1.1 研究動機.....	1
1.2 正交分頻多工系統(OFDM)簡介	2
1.3 論文組織.....	5
第二章 DFT/IDFT 演算法.....	6
2.1 簡介.....	6
2.2 Radix-2 FFT/IFFT 演算法	6
2.2.1 FFT 演算法	6
2.2.2 IFFT 演算法	10
2.3 Radix-4 FFT/IFFT 演算法	13
2.3.1 FFT 演算法.....	13
2.3.2 Radix-4 IFFT 演算法	15
2.4 Radix-2 ² FFT 演算法	17
2.5 Split-Radix 2/4 FFT 演算法	19
2.6 結論.....	21
第三章 FFT/IFFT 架構.....	22
3.1 簡介.....	22
3.2 Pipeline 架構	22
3.2.1 Multiple-Path Delay Commutator (MDC) Pipeline Architecture	23
3.2.2 Single-Path Delay Feedback (SDF) Pipeline Architecture	24
3.2.3 Convergent Block Floating Point Pipeline Architectures	26
3.2.4 不同 Pipeline 架構的比較	29
3.3 Memory-based 架構	30
3.4 結論.....	33
第四章 應用於實數時域之 FFT/IFFT 架構	35
4.1 簡介.....	35
4.2 Real-Value FFT(RFFT)/Hermitian Symmetric IFFT (HS- IFFT)演算法	35
4.2.1 RFFT 演算法	35
4.2.2 HS-IFFT 演算法	39
4.3 Memory-Based Radix-2 雙模可變長度 FFT/IFFT 處理器架構.....	41
4.3.1 Radix-2 DIT FFT 處理器架構.....	41
4.3.2 Radix-2 DIT IFFT 處理器架構.....	43

4.3.3 雙模 Radix-2 DIT FFT/IFFT 處理器架構.....	45
4.3.4 資料位址產生器(DAG)及係數位址產生器(CAG)之探討	47
4.3.5 Conflict Free Memory Addressing.....	53
4.3.6 Block Floating Point Memory-Based 架構	55
4.4 Memory-Based Radix-4 雙模可變長度 FFT/IFFT 架構.....	56
4.5 Memory-Based 多模可變長度 RFFT/HS-IFFT 架構	59
4.6 結論.....	65
第五章 應用於實數時域 FFT/IFFT 處理器	66
之硬體實現與效能分析	66
5.1 簡介.....	66
5.2 Memory-Based Radix-2 多模可變長度 RFFT/HS-IFFT 處理器硬體設計	66
5.2.1 DAG 和 R_DAG 硬體設計.....	67
5.2.2 CAG 和 R_CAG 硬體設計	71
5.2.3 多模 Radix-2 Butterfly 處理器硬體設計	73
5.2.4 Radix-2 多模可變長度 RFFT/HS-IFFT 硬體架構	76
5.3 Memory-Based Radix-4 多模可變長度 RFFT/HS-IFFT 處理器硬體設計	78
5.3.1 DAG 和 R_DAG 硬體設計.....	79
5.3.2 CAG 和 R_CAG 硬體設計	81
5.3.3 多模 Radix-4 Butterfly 運算器硬體設計	82
5.3.4 Radix-4 多模可變長度 RFFT/HS-IFFT 硬體架構	85
5.4 效能分析及硬體需求.....	87
5.4.1 Radix-2 架構之 SQNR 分析	87
5.4.2 Radix-4 架構之 SQNR 分析	94
5.4.3 記憶體大小需求與運算所需 Clock Cycles 數.....	98
5.5 硬體實作與量測結果.....	100
5.6 結論.....	103
第六章 結論與未來展望	104
6.1 結論.....	104
6.2 未來展望.....	105
參考資料.....	106

圖示目錄：

圖 1.1：OFDM 之多載波傳送示意圖	3
圖 1.2：OFDM 之保護區間示意圖	3
圖 1.3：(a)傳統多載波系統的子通道分布，(b)正交分頻多工系統的子通道分布	4
圖 1.4：離散時間正交分頻多工系統模型	4
圖 2.1：八點分時離散傅立葉轉換分解成兩個四點的分時離散傅立葉轉換流程圖	8
圖 2.2：八點之分時快速傅立葉轉換流程圖	8
圖 2.3：八點分頻離散傅立葉轉換分解成兩個四點的分頻離散傅立葉轉換流程圖	10
圖 2.4：八點之分頻快速傅立葉轉換流程圖	10
圖 2.5：八點之反向分時快速傅立葉轉換流程圖	11
圖 2.6：八點之反向分頻快速傅立葉轉換流程圖	13
圖 2.7：(a) Radix-4 FFT 基本 butterfly，(b) Radix-4 butterfly 簡圖	14
圖 2.8：16 點之 radix-4 分時快速傅立葉轉換流程圖	14
圖 2.9：(a) Radix-4 IFFT 基本 butterfly，(b) Radix-4 butterfly 簡圖	16
圖 2.10：16 點之 radix-4 反向分時快速傅立葉轉換流程圖	16
圖 2.11：16 點 radix-2 ² 分頻離散傅立葉轉換分解流程圖	18
圖 2.12：16 點之 radix-2 ² 分頻快速傅立葉轉換流程圖	19
圖 2.13：Split-radix 2/4 DIF 基本 butterfly	20
圖 2.14：16 點 Split-radix 2/4 分頻快速傅立葉轉換流程圖	20
圖 3.1：16 點 R2MDC 快速傅立葉處理器架構圖	23
圖 3.2：256 點 R4MDC 快速傅立葉處理器架構圖	24
圖 3.3：256 點 R4SDC 快速傅立葉處理器架構圖	24
圖 3.4：16 點 R2SDF 快速傅立葉處理器架構圖	25
圖 3.5：256 點 R4SDF 快速傅立葉處理器架構圖	26
圖 3.6：16 點 R2 ² SDF 快速傅立葉處理器架構圖	26
圖 3.7：16 點 radix-2 DIF FFT 之 BFP 架構流程圖	27
圖 3.8：16 點 radix-2 DIF FFT 之 CBFP 架構流程圖	28
圖 3.9：CBFP pipeline 架構的十六點 R2SDF 快速傅立葉處理器架構圖	29
圖 3.10：16 點 DIT FFT 流程圖	31
圖 3.11：單記憶體 memory-based 架構快速傅立葉轉換處理器方塊圖	32
圖 3.12：雙記憶體 memory-based 架構快速傅立葉轉換處理器方塊圖	32
圖 4.1：實數序列之 CFFT 方塊圖	36
圖 4.2：雙實數序列之 CFFT 及後端處理方塊圖	37
圖 4.3：單實數序列之 CFFT 及後端處理方塊圖	38
圖 4.4：Hermitian Symmetric 序列之前端處理及 CIFFT 方塊圖	40
圖 4.5：16 點 radix-2 DIT FFT 運算流程圖	41

圖 4.6 : 16 點 radix-2 DIT FFT 運算等效流程圖	42
圖 4.7 : Memory-based 架構的 radix-2 FFT 處理器方塊圖	42
圖 4.8 : 16 點 radix-2 DIT IFFT 運算流程圖	44
圖 4.9 : 16 點 radix-2 DIT IFFT 運算等效流程圖	44
圖 4.10 : Memory-based 架構 radix-2 IFFT 處理器方塊圖	45
圖 4.11 : 雙模 memory-based 架構之 radix-2 FFT/IFFT 處理器方塊圖	46
圖 4.12 : (a)大矩陣運算(b)分解成三個小矩陣運算	48
圖 4.13 : $i:[a, b] \xrightarrow{S_{3.5}} j:[b, a]$ 圖示	49
圖 4.14 : 資料位址產生器方塊圖	50
圖 4.15 : 16 點 radix-2 DIT FFT/IFFT 之 CAG 示意圖	50
圖 4.16 : 係數位址產生器方塊圖	51
圖 4.17 : 32 點分時快速傅立葉轉換所需之 twiddle factors 圖示	52
圖 4.18 : Radix-r FFT butterfly 方塊圖	53
圖 4.19 : 16 點 radix-2 FFT 之區塊配置示意圖	54
圖 4.20 : BFP memory-based 架構的雙模 radix-2 FFT/IFFT 處理器方塊圖	55
圖 4.21 : 64 點 radix-4 DIT FFT 運算等效流程圖	57
圖 4.22 : 64 點 radix-4 DIT IFFT 運算等效流程圖	58
圖 4.23 : BFP memory-based 架構的雙模 radix-4 FFT/IFFT 處理器方塊圖	59
圖 4.24 : RFFT/HS-IFFT 方塊圖	60
圖 4.25 : RFFT 方塊圖	60
圖 4.27 : HS-IFFT 方塊圖	61
圖 4.26 : 後端處理器流程圖	62
圖 4.28 : 前端處理器流程圖	63
圖 4.29 : BFP memory-based 架構之多模 RFFT/HS-IFFT 處理器方塊圖	64
圖 5.1 : Radix-2 多模可變長度 RFFT/HS-IFFT 處理器方塊圖	66
圖 5.2 : 資料位址產生器方塊圖	67
圖 5.3 : $k:[a, b, c] \xrightarrow{shuffle} address[a, c, b]$ 之二位元表示圖	68
圖 5.4 : $k:[a, b, c] \xrightarrow{shuffle} address[a, c, b]$ 的示意圖	68
圖 5.5 : Radix-2 DAG 邏輯架構圖	68
圖 5.6 : Radix-2 R_DAG 邏輯架構圖	69
圖 5.7 : Radix-2 記憶體區塊配置方塊圖	70
圖 5.8 : Radix-2 記憶體區塊配置方塊圖	71
圖 5.9 : Radix-2 係數位址產生器方塊圖	71
圖 5.10 : Radix-2 R_CAG 邏輯架構圖	72
圖 5.11 : Reduce coefficient memory size 架構圖	73
圖 5.12 : Decoder 內部邏輯架構	73
圖 5.13 : 多模 radix-2 butterfly 運算器的六種運算模式	74

圖 5.14 : 多模 radix-2 butterfly 運算器架構.....	74
圖 5.15 : Single port 記憶體 timing diagram.....	75
圖 5.16 : 以兩個實數乘法器實現在兩個 clock cycles 內運算的複數運算架構圖	75
圖 5.17 : Radix-2 butterfly 輸入範圍和輸出範圍的相關圖	76
圖 5.18 : Radix-2 多模可變長度 RFFT/HS-IFFT 硬體架構圖.....	77
圖 5.19 : Radix-4 多模可變長度 RFFT/HS-IFFT 處理器方塊圖.....	78
圖 5.20 : Radix-4 DAG 邏輯架構圖	79
圖 5.21 : Radix-4 R_DAG 邏輯架構圖.....	80
圖 5.22 : Radix-4 記憶體區塊配置方塊圖.....	81
圖 5.23 : Radix-4 CAG 邏輯架構圖.....	81
圖 5.24 : Radix-4 R_CAG 邏輯架構圖.....	82
圖 5.25 : Basic radix-4 butterfly 運算器.....	83
圖 5.26 : 多模 radix-4 butterfly 運算器之其它運算模式.....	84
圖 5.27 : Radix-4 butterfly 輸入範圍和輸出範圍的相關圖	85
圖 5.28 : Radix-4 多模可變長度 RFFT/HS-IFFT 的硬體架構圖.....	86
圖 5.29 : 計算 SQNR 的示意圖	87
圖 5.30 : FFT/IFFT 輸入訊號功率与其它功率相關圖.....	89
圖 5.31 : FFT/IFFT 輸入訊號功率與 SQNR 相關圖	89
圖 5.32 : RFFT 輸入訊號功率与其它功率相關圖	90
圖 5.33 : RFFT 輸入訊號功率與 SQNR 相關圖.....	90
圖 5.34 : HS-IFFT 輸入訊號功率与其它功率相關圖	91
圖 5.35 : HS-IFFT 輸入訊號功率與 SQNR 相關圖.....	91
圖 5.36 : 不同字元長度下, FFT/IFFT 輸入訊號功率与其它功率相關圖.....	92
圖 5.37 : FFT/IFFT 不同字元長度的 SQNR 圖.....	92
圖 5.38 : RFFT 不同字元長度的 SQNR 圖.....	93
圖 5.39 : HS-IFFT 不同字元長度的 SQNR 圖.....	93
圖 5.40 : FFT/IFFT 輸入訊號功率与其它功率相關圖.....	95
圖 5.41 : FFT/IFFT 輸入訊號功率與 SQNR 相關圖	95
圖 5.42 : 不同字元長度下, FFT/IFFT 輸入訊號功率与其它功率相關圖.....	96
圖 5.43 : FFT/IFFT 不同字元長度的 SQNR 圖.....	96
圖 5.44 : RFFT 不同字元長度的 SQNR 圖.....	97
圖 5.45 : HS-IFFT 不同字元長度的 SQNR 圖.....	97
圖 5.46 : RTL code 階層圖	100
圖 5.47 : DSP development board, Stratix edition	102
圖 5.48 : 驗證方式.....	102

表格目錄：

表 1.1：各通訊系統所需 FFT 處理器轉換大小和資料取樣速率	2
表 2.1：各演算法計算複雜度之比較	21
表 3.1：各不同 pipeline 架構之硬體需求比較	30
表 3.2：各不同 pipeline 架構之硬體使用率比較	30
表 3.3：大點數快速傅立葉轉換之 pipeline 架構與 memory-based 架構比較	33
表 4.1：32 點分時快速傅立葉轉換所需之 twiddle factors 關係表	52
表 5.1：Radix-2 多模處理器操縱模式	67
表 5.2：Radix-4 多模處理器操縱模式	78
表 5.3：字元長度 16 位元的最大可變長度範圍所需記憶體大小	98
表 5.4：各不同轉換點數與運算模式所需的 clock cycles 數	99
表 5.5：各通訊系統所需 FFT 處理器轉換大小和資料取樣速率	99
表 5.6：多模可變長度 RFFT/HS-IFFT 處理器不含記憶體的邏輯閘數目	101
表 5.7：多模可變長度 RFFT/HS-IFFT 處理器的最大操作頻率	102



第一章 簡介

1.1 研究動機

隨著通訊和訊號處理技術的快速進步，以及 VLSI 製程技術的提升，使得過往的一些通訊理論和系統得以真正地被拿來應用，進而帶動了許多無線或有線通訊的相關產業蓬勃發展。因為通訊產品的普及化，使得頻寬的需求與高傳輸速率的期望越來越被重視，這也加速了通訊技術的推進，其中又以具有高傳輸速率與抗多路徑通道特性的正交分頻多工(Orthogonal Frequency Division Multiplexing, OFDM)技術最引人注目。

正交分頻多工早在 1960 年就被提出[1]，其主要的目的在於解決有限頻寬之間的 ISI 問題，但因為當時數位訊號處理技術並不發達和硬體製程技術的侷限，使得 OFDM 技術在當時不受歡迎。直到 1971 年 Weistein 及 Ebert[2]提出了利用快速傅立葉轉換(Fast Fourier Transform, FFT)與反向快速傅立葉轉換(Inverse Fast Fourier Transform, IFFT)來取代類比多載波的技术，OFDM 技術才真正地被重視。目前許多的傳輸技術，包含一些有線的傳輸系統如非對稱式數位用戶端迴路系統(ADSL)和高速數位用戶端迴路系統(VDSL)，及無線的傳輸系統如數位廣播系統(DAB)、數位視訊地面廣播系統(DVB-T)和 IEEE 802.11a/g 等系統，皆是以 OFDM 技術為傳輸標準。而這些屬於正交分頻多工的通訊系統皆需用 FFT/IFFT 來實現，所以如何設計一個高效率低成本的快速(反)傅立葉轉換(FFT/IFFT)處理器一直是受矚目的問題。

FFT/IFFT 在 OFDM 技術中主要扮演的角色，在於提供其所需的正交多載波，而不同的通訊系統其所需之正交載波數也都不盡相同，表(1.1)[3]提供了不同通訊系統所需 FFT/IFFT 的轉換點數與取樣速度。一般來說，若 FFT/IFFT 轉換的點數越大則所需的運算時間及硬體面積也會跟著變長與增加，所以為了使硬體能達到高效能、運算時間短、低面積與低功率等需求，演算法的選擇及其架構的設計極為重要。常見的 FFT/IFFT 硬體實現架構可分為兩類，pipeline 架構與

memory-based 架構，對於一個要求高速運算的系統來說，pipeline 架構的 FFT/IFFT 是最好的選擇，而若對於以硬體面積大小為考量的話，則 memory-based 架構可符合其需求。此外，在一些使用 OFDM 技術的通訊系統中，若 FFT/IFFT 處理器之轉換點數為可變的，則其應用層面將會更有彈性。

Communication System	FFT Size (Sampling Rate)
802.11a	64(20MHz)
DAB	2048(2MHz)、1024(2MHz)、512(2MHz)、256(2MHz)
DVB-T	8192(8MHz)、2048(8MHz)
ADSL	512(2.2MHz)
VDSL	8192(34.5MHz)、4096(17.3MHz)、2048(8.6MHz)、1024(4.3MHz)、512(2.2MHz)

表 1.1：各通訊系統所需 FFT 處理器轉換大小和資料取樣速率

事實上，FFT/IFFT 其應用的領域相當廣，不只是應用於 OFDM 技術中，舉凡像語音、影像、雷達訊號的處理以及最基本的頻譜分析等數位訊號處理，FFT/IFFT 都佔有重要的一席之地。故設計一個有效率地運算且低硬體需求的可變長度 FFT/IFFT 處理器，將可讓數位訊號處理在硬體上更為實用。

1.2 正交分頻多工系統(OFDM)簡介

多載波傳輸技術原理是將原本高速傳輸的串列資料分割成 N 個低速傳輸(傳輸速度為原本的 $1/N$)的平行資料串列，再將其用 N 個子載波同時傳輸，如圖(1.1)所示。當使用多載波傳輸時，各分割後的低速資料串列其符元區間將會變大 N 倍，相對於頻域而言其在子載波上的信號頻寬將縮小 N 倍，因此可以降低由多重路徑延遲擴散 (Multi-path Delay Spread) 所引起的符號間干擾 (Inter System Interference, ISI)。此外，多路徑延遲擴散除了造成符號之間的干擾外，也會造成

OFDM 系統中不同區塊間的干擾。為了能夠消除此干擾，於每一個 OFDM 方塊中置入了保護區間(Guard Interval)或稱為循環前置碼(Cyclic Prefix)，如圖(1.2)所示，且此保護區間的長度必須大過於所預測之最大多路徑延遲擴散，使得 OFDM 區塊不會受到上一個 OFDM 區塊干擾。雖然欲達到避免相鄰 OFDM 區塊間的干擾，保護區間可以完全不傳送信號，但此情況下，將會產生載波間干擾(Inter-Carrier-Interference, ICI)，導致載波之間不再具正交性。因此，為了消除子載波間的干擾效應，選擇保護區段之信號為 OFDM 區塊的循環延伸(Cyclic Extension)來維持子載波間的正交性。

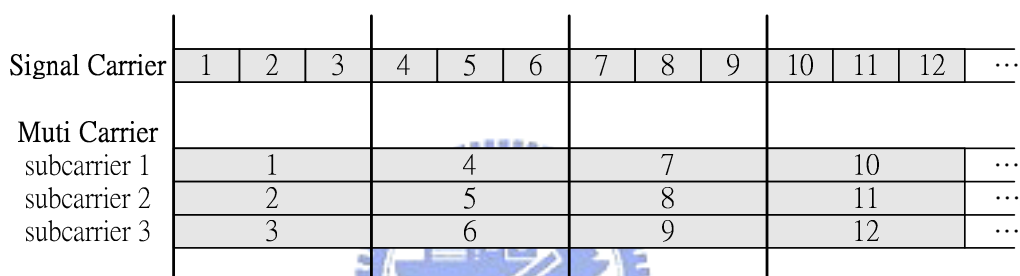


圖 1.1：OFDM 之多載波傳送示意圖

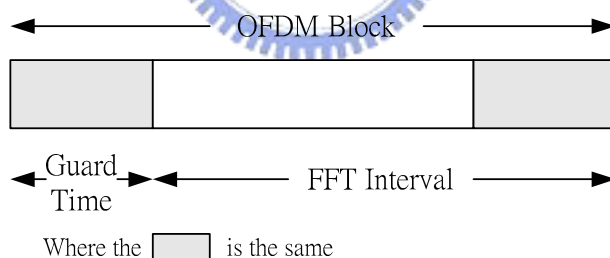


圖 1.2：OFDM 之保護區間示意圖

傳統的分頻多工(Frequency Division Multiplexing, FDM)是利用頻帶間不交疊的子載波來讓子載波彼此間不互相干擾，如圖(1.3(a))所示，而 OFDM(Orthogonal Frequency Division Multiplexing)則是利用子載波彼此之間的正交性(Orthogonality)讓即使頻帶相互交疊的子載波也能達到互不干擾的狀態，這將使使用多載波傳輸模式的頻寬效益提升，如圖(1.3(b))所示。

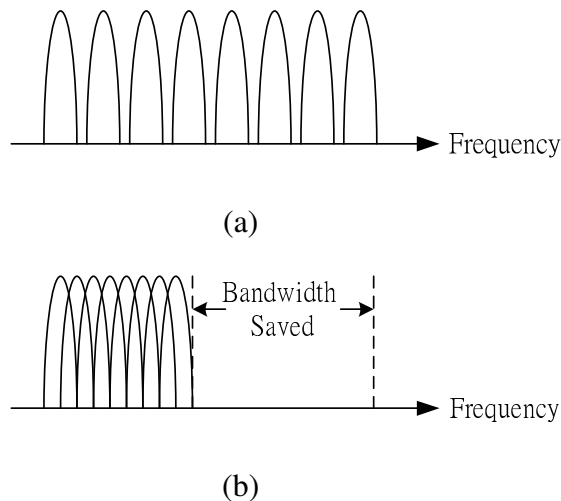


圖 1.3：(a)傳統多載波系統的子通道分布，(b)正交分頻多工系統的子通道分布

要實現正交分頻多工調變技術，最好的方式是利用 IDFT/DFT 所產生的正交子載波來達到，圖(1.4)為利用 IDFT/DFT 的 OFDM 調變示意圖。其中 X_k 為第 k 個符元的 N 個欲調變資料， \bar{X}_k 為第 k 個符元的 N 個解條後資料，利用 IDFT 與 DFT 的關係式可知 $\bar{X}_k = DFT(IDFT(X_k))$ 。而實做一個正交分頻多工系統時，IDFT 與 DFT 可由 IFFT 和 FFT 來實現。

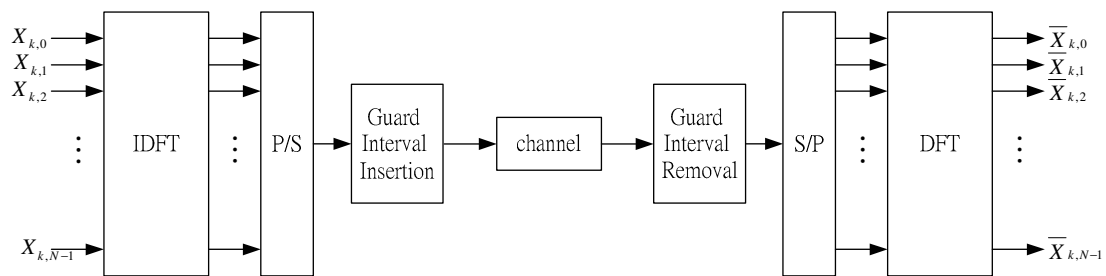


圖 1.4：離散時間正交分頻多工系統模型

在許多通訊傳輸系統中，OFDM 技術已被採為傳輸標準，在有線環境中，如非對稱式數位用戶端迴路系統(ADSL)和高速數位用戶端迴路系統(VDSL)，在無線環境中，如數位廣播系統(DAB)、數位視訊地面廣播系統(DVB-T)和 IEEE 802.11a/g 等系統都是藉由 OFDM 技術來傳輸。

1.3 論文組織

本論文主要為實現一個以 memory-based 架構為基礎的多模可變長度快速(反)傅立葉轉換處理器，其內容包含處理器的演算法、架構設計、效能分析、硬體實現與測試。在第二章首先我們介紹了一般最常被用來有效率計算離散傅立葉轉換(Discret Fourier Transform, DFT)的快速傅立葉轉換(Fast Fourier Transform, FFT)演算法，並比較其運算複雜度。第三章將提及兩種常見的 FFT/IFFT 處理器硬體實現架構，分別為 pipeline 架構和 memory-based 架構。第四章首先將針對輸入序列為實數序列的 FFT、輸入序列為 Hermitian Symmetric 序列的 IFFT 所採用的演算法做介紹，之後將以 memory-based 架構為主要重點，來設計一個低硬體需求的多模可變長度 RFFT/HS-IFFT 轉換處理器架構。第五章主要在執行 memory-based 架構多模可變長度 RFFT/HS-IFFT 處理器的硬體實現與測試，並分析處理器的效能。最後，將在第六章做個結論與未來的展望。



第二章 DFT/IDFT 演算法

2.1 簡介

在許多數位訊號處理與通訊系統等領域上，離散傅立葉轉換(Discrete Fourier Transform, DFT)一直扮演著重要的角色。然而，在一個得運用到離散傅立葉轉換的即時訊號處理系統中，快速地得到其運算後的結果是必要的，因此也發展出了許多快速計算 DFT 的演算法。有許多方法可用來衡量一演算法或實施的複雜度和效率，當演算法在一般的微處理器上實現時，乘法數及加法數將直接和計算的速度有關，故我們將使用乘法數來衡量其計算複雜度。一個最常見且有效率的演算法為快速傅立葉轉換(FFT)演算法，因為它不只可以將直接 DFT 運算的計算複雜度從 $O(N^2)$ 減少至 $O(N \log N)$ ，並且由於其演算法的規律性，使 FFT 演算法非常適合於在 VLSI 中實現。

FFT 演算法的基本原理是將“長度為 N 之序列的離散傅立葉轉換的計算分解成許多小的離散傅立葉轉換”來達到有效的運算。FFT(IFFT)演算法有兩種形式的分解，第一種為採分時(Decimation-in-time, DIT)的演算法，即將原序列 $x[n]$ 連續分解成許多小的子序列，之後再算每個小序列的轉換，另一種為採分頻(Decimation-in-frequency, DIF)的演算法，即離散傅立葉轉換 $X[k]$ 被分解成許多小的子序列。本章將介紹 FFT(IFFT)演算法的一些不同行式，如 radix-2、radix-4、radix-2² 和 split-radix 2/4 等[4][5][6]。並在最後比較各不同演算法的計算複雜度。

2.2 Radix-2 FFT/IFFT 演算法

2.2.1 FFT 演算法

I. Decimation-in-time 演算法

當我們計算 DFT 時，高度的效率可從分解計算成許多連續的小 DFT 計算而得到，在這個過程，複數指數 $W_N^{kn} = e^{-j(2\pi/N)kn}$ 的對稱性及週期性均被利用到，而演算法的分解就是基於把離散序列 $x[n]$ 分成許多小序列，故稱之為分時演算法。

一個長度為 N 的離散序列 $x[n]$ ，其離散傅利葉轉換可以表示為：

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.1)$$

藉由將 $x[n]$ 分成奇數點和偶數點兩個部分，我們可以得到：

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] W_N^{2nk} + \sum_{n=0}^{N/2-1} x[2n+1] W_N^{(2n+1)k}, \quad k = 0, 1, 2, \dots, N-1 \quad (2.2)$$

令 $g[n] = x[2n]$ 且 $h[n] = x[2n+1]$ ，則：

$$\begin{aligned} X[k] &= \sum_{n=0}^{N/2-1} g[n] W_N^{2nk} + \sum_{n=0}^{N/2-1} h[n] W_N^{(2n+1)k} \\ &= \sum_{n=0}^{N/2-1} g[n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} h[n] W_{N/2}^{nk} \\ &= G[k] + W_N^k H[k] \end{aligned} \quad (2.3)$$

其中 $G[k]$ 和 $H[k]$ 分別為 $x[n]$ 之偶數點和奇數點的 $N/2$ 點離散傅立葉轉換。

$$\begin{aligned} X[k + N/2] &= \sum_{n=0}^{N/2-1} g[n] W_{N/2}^{nk} - W_N^k \sum_{n=0}^{N/2-1} h[n] W_{N/2}^{nk} \\ &= G[k] - W_N^k H[k] \end{aligned} \quad (2.4)$$

圖(2.1)為八點之分時離散傅立葉轉換分解成兩個四點的分時離散傅立葉轉換的流程圖。同理，我們可以繼續把 $N/2$ 點的離散傅立葉轉換再分解成兩個點數為 $N/4$ 點的離散傅立葉轉換，逐次分解下去，則可得分時快速傅立葉轉換。下圖(2.2)為八點之分時快速傅立葉轉換流程圖。

一個長度為 N 之 radix-2 DIT FFT 的計算複雜度如下所示：

$$M(N) = (N/2) \log_2 N \quad (2.5)$$

$$A(N) = N \log_2 N \quad (2.6)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

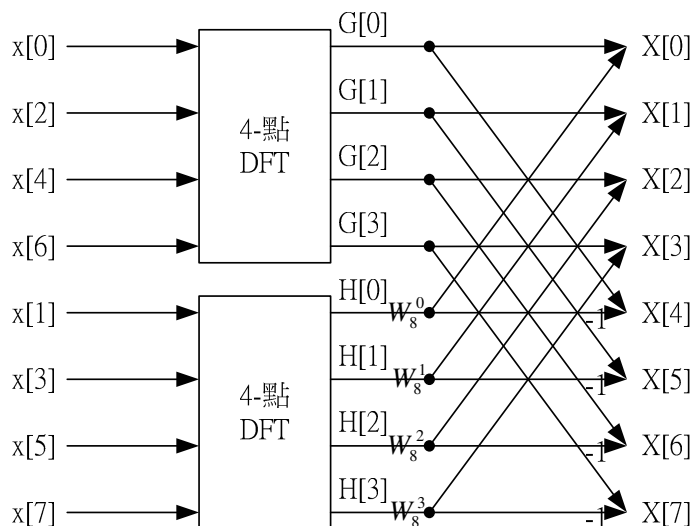


圖 2.1：八點分時離散傅立葉轉換分解成兩個四點的分時離散傅立葉轉換流程圖

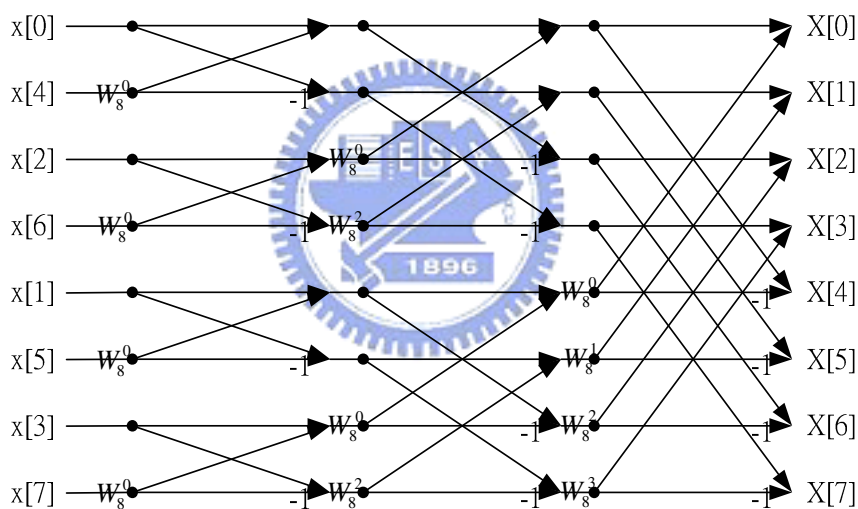


圖 2.2：八點之分時快速傅立葉轉換流程圖

II. Decimation-in-frequency 演算法

分時的 FFT 演算法是把輸入序列 $x[n]$ 分解成許多小序列的 DFT 來運算。同樣地，我們亦可把輸出序列 $X[k]$ 分解成許多子序列來快速計算 DFT，這種 FFT 演算法稱為分頻演算法。

一個長度為 N 的離散序列 $x[n]$ ，其離散傅利葉轉換可以表示為：

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, k = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.7)$$

將其 $x[n]$ 分成兩半，則：

$$\begin{aligned} X[k] &= \sum_{n=0}^{N/2-1} x[n]W_N^{nk} + \sum_{n=N/2}^{N-1} x[n]W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x[n]W_N^{nk} + (-1)^k \sum_{n=0}^{N/2-1} x[n+N/2]W_N^{nk} \end{aligned} \quad (2.8)$$

其中 $X[k]$ 的偶數點和奇數點分別為：

$$\begin{aligned} X[2r] &= \sum_{n=0}^{N/2-1} x[n]W_N^{2nr} + (-1)^{2r} \sum_{n=0}^{N/2-1} x[n+N/2]W_N^{2nr} \\ &= \sum_{n=0}^{N/2-1} x[n]W_{N/2}^{nr} + \sum_{n=0}^{N/2-1} x[n+N/2]W_{N/2}^{nr} \\ &= \sum_{n=0}^{N/2-1} (x[n] + x[n+N/2])W_{N/2}^{nr} \end{aligned} \quad (2.9)$$

$$\begin{aligned} X[2r+1] &= \sum_{n=0}^{N/2-1} x[n]W_N^n W_N^{2nr} + (-1)^{2r+1} \sum_{n=0}^{N/2-1} x[n+N/2]W_N^n W_N^{2nr} \\ &= \sum_{n=0}^{N/2-1} x[n]W_N^n W_{N/2}^{nr} - \sum_{n=0}^{N/2-1} x[n+N/2]W_N^n W_{N/2}^{nr} \\ &= \sum_{n=0}^{N/2-1} (x[n] - x[n+N/2])W_N^n W_{N/2}^{nr} \end{aligned} \quad (2.10)$$

令 $g[n] = x[n] + x[n+N/2]$ 、 $h[n] = (x[n] - x[n+N/2])W_N^n$ ，則：

$$X[2r] = \sum_{n=0}^{N/2-1} g[n]W_{N/2}^{nr} \quad \text{且} \quad X[2r+1] = \sum_{n=0}^{N/2-1} h[n]W_{N/2}^{nr} \quad (2.11)$$

圖(2.3)為八點之分頻離散傅立葉轉換分解成兩個四點的分頻離散傅立葉轉換的流程圖。同理，我們也可以跟分時離散傅立葉轉換一樣逐次分解，則可得分頻快速傅立葉轉換。下圖(2.4)為八點之分頻快速傅立葉轉換流程圖。

一個長度為 N 之 radix-2 DIF FFT 具有跟 DIT 相同的計算複雜度：

$$M(N) = (N/2) \log_2 N \quad (2.12)$$

$$A(N) = N \log_2 N \quad (2.13)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

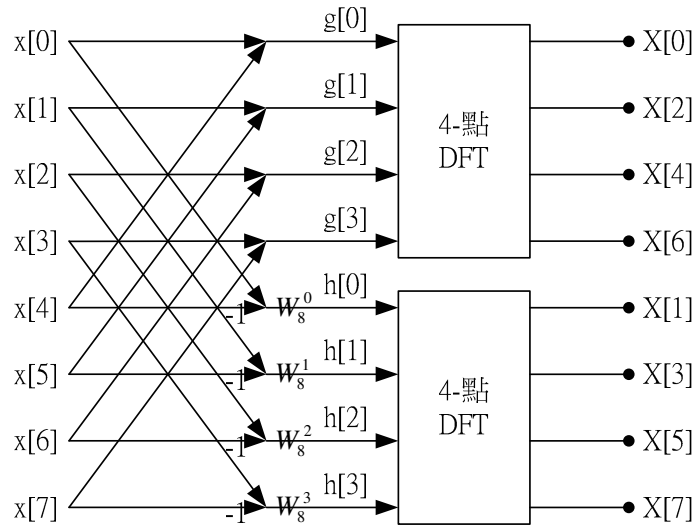


圖 2.3：八點分頻離散傅立葉轉換分解成兩個四點的分頻離散傅立葉轉換流程圖

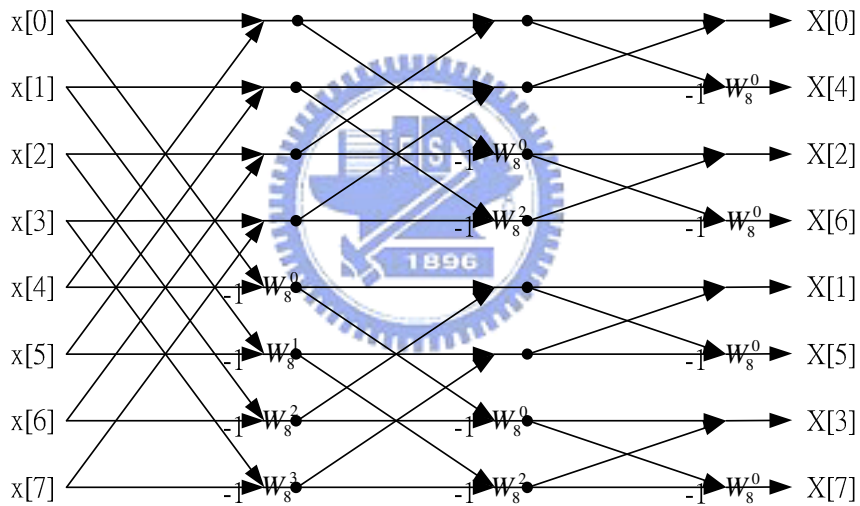


圖 2.4：八點之分頻快速傅立葉轉換流程圖

2.2.2 IFFT 演算法

I. Decimation-in-time 演算法

一個長度為 N 的離散序列 $X[k]$ ，其反向離散傅利葉轉換可以表示為：

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-nk}, n = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.14)$$

與分時 FFT 一樣，藉由將 $X[k]$ 分成奇數點和偶數點兩個部分，我們可以得到：

$$x[n] = \sum_{k=0}^{N/2-1} X[2k]W_N^{-2nk} + \sum_{k=0}^{N/2-1} X[2k+1]W_N^{-(2k+1)n}, n=0,1,2,\dots,N-1 \quad (2.15)$$

令 $G[k] = X[2k]$ 且 $H[k] = X[2k+1]$ ，則：

$$\begin{aligned} x[n] &= \sum_{k=0}^{N/2-1} G[k]W_N^{-2nk} + \sum_{k=0}^{N/2-1} H[k]W_N^{-(2k+1)n} \\ &= \sum_{k=0}^{N/2-1} G[k]W_{N/2}^{-nk} + W_N^{-n} \sum_{k=0}^{N/2-1} H[k]W_{N/2}^{-nk} \\ &= g[n] + W_N^{-n}h[n] \end{aligned} \quad (2.16)$$

其中 $g[n]$ 和 $h[n]$ 分別為 $X[k]$ 之偶數點和奇數點的 $N/2$ 點反向離散傅立葉轉換。

$$\begin{aligned} x[n+N/2] &= \sum_{k=0}^{N/2-1} G[k]W_{N/2}^{-nk} - W_N^{-n} \sum_{k=0}^{N/2-1} H[k]W_{N/2}^{-nk} \\ &= g[n] - W_N^{-n}h[n] \end{aligned} \quad (2.17)$$

我們可以繼續將 $N/2$ 點的反向離散傅立葉轉換再分解成兩個點數為 $N/4$ 點的反向離散傅立葉轉換，逐次分解下去，則可得反向分時快速傅立葉轉換。下圖(2.5)

為八點之反向分時快速傅立葉轉換流程圖。

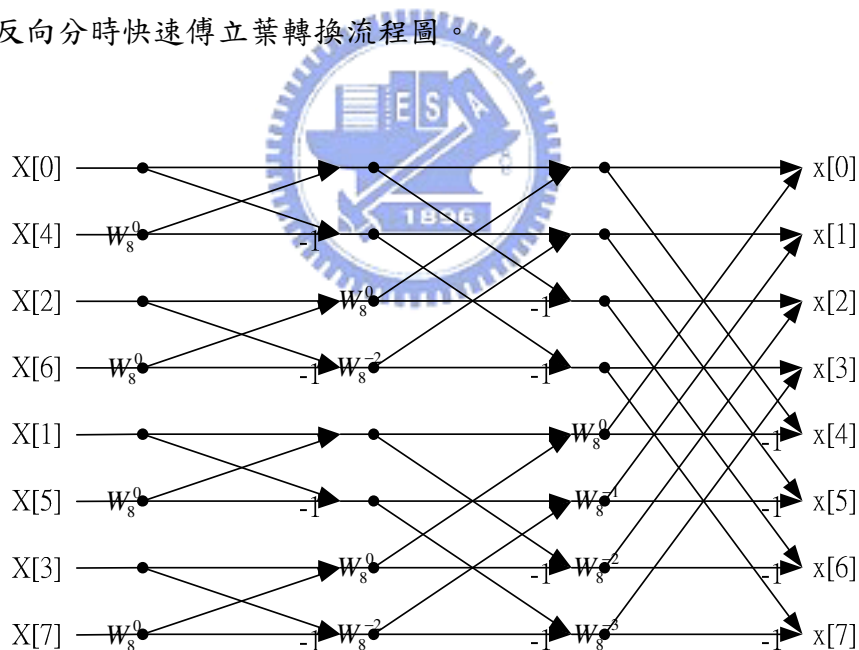


圖 2.5：八點之反向分時快速傅立葉轉換流程圖

一個長度為 N 之 radix-2 DIT IFFT 的計算複雜度為：

$$M(N) = (N/2)\log_2 N \quad (2.18)$$

$$A(N) = N\log_2 N \quad (2.19)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

II. Decimation-in-frequency 演算法

一個長度為 N 的離散序列 $X[k]$ ，其離散傅利葉轉換可以表示為：

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-nk}, \quad n = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.20)$$

與分頻 FFT 一樣，將其 $X[k]$ 分成兩半，則：

$$\begin{aligned} x[n] &= \sum_{k=0}^{N/2-1} X[k] W_N^{-nk} + \sum_{k=N/2}^{N-1} X[k] W_N^{-nk} \\ &= \sum_{k=0}^{N/2-1} X[k] W_N^{-nk} + (-1)^n \sum_{k=0}^{N/2-1} X[k + N/2] W_N^{-nk} \end{aligned} \quad (2.21)$$

其中 $x[n]$ 的偶數點和奇數點分別為：

$$\begin{aligned} x[2r] &= \sum_{k=0}^{N/2-1} X[k] W_N^{-2kr} + (-1)^{2r} \sum_{k=0}^{N/2-1} X[k + N/2] W_N^{-2kr} \\ &= \sum_{k=0}^{N/2-1} X[k] W_{N/2}^{-kr} + \sum_{k=0}^{N/2-1} X[k + N/2] W_{N/2}^{-kr} \\ &= \sum_{k=0}^{N/2-1} (X[k] + X[k + N/2]) W_{N/2}^{-kr} \end{aligned} \quad (2.22)$$

$$\begin{aligned} x[2r+1] &= \sum_{k=0}^{N/2-1} X[k] W_N^{-k} W_N^{-2kr} + (-1)^{2r+1} \sum_{k=0}^{N/2-1} X[k + N/2] W_N^{-k} W_N^{-2kr} \\ &= \sum_{k=0}^{N/2-1} X[k] W_N^{-k} W_{N/2}^{-kr} - \sum_{k=0}^{N/2-1} X[k + N/2] W_N^{-k} W_{N/2}^{-kr} \\ &= \sum_{k=0}^{N/2-1} (X[k] - X[k + N/2]) W_N^{-k} W_{N/2}^{-kr} \end{aligned} \quad (2.23)$$

令 $G[k] = X[k] + X[k + N/2]$ 、 $H[k] = (X[k] - X[k + N/2]) W_N^{-k}$ ，則：

$$x[2r] = \sum_{k=0}^{N/2-1} G[k] W_{N/2}^{-kr} \quad \text{且} \quad x[2r+1] = \sum_{k=0}^{N/2-1} H[k] W_{N/2}^{-kr} \quad (2.24)$$

同理，可將 $N/2$ 點的反向離散傅立葉轉換繼續分解成兩個點數為 $N/4$ 點的反向離散傅立葉轉換，逐次分解下去，則可得反向分頻快速傅立葉轉換。下圖(2.6)為八點之反向分頻離散傅立葉轉換流程圖。

一個長度為 N 之 radix-2 DIF IFFT 的計算複雜度：

$$M(N) = (N/2) \log_2 N \quad (2.25)$$

$$A(N) = N \log_2 N \quad (2.26)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

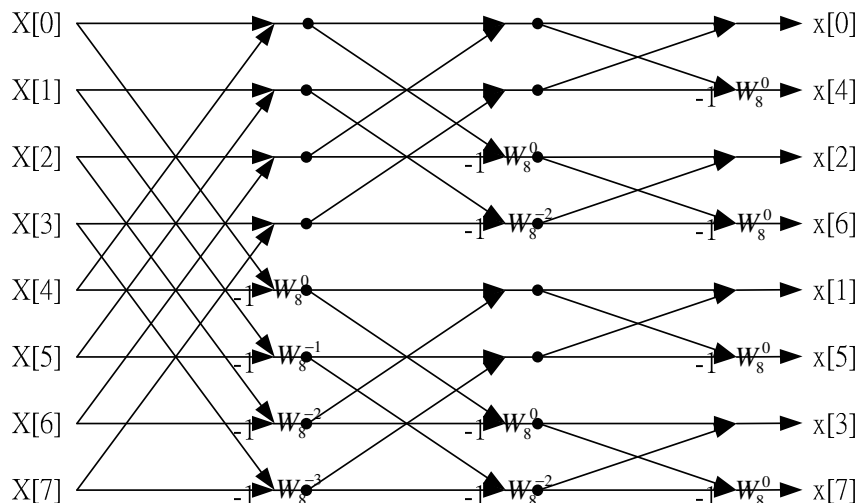


圖 2.6：八點之反向分頻快速傅立葉轉換流程圖

2.3 Radix-4 FFT/IFFT 演算法

2.3.1 FFT 演算法

Decimation-in-time 演算法

若 DFT 的輸入序列點數 N 為 4 的冪次方時 ($N = 4^v$)，當然我們還是可以選擇用 radix-2 演算法來計算。此外，我們也可以選擇更有效率的 radix-4 演算法來計算。一個長度為 N 的離散序列 $x[n]$ ，其離散傅立葉轉換可以表示為：

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, k = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.27)$$

如同 Radix-2 DIT 的做法可得以下四式，利用下列四式，我們可以得到 radix-4 的基本 butterfly 運算。下圖 2.7(a) 為 radix-4 DIT FFT butterfly 之圖示，下圖 2.7(b) 為其簡圖。

$$X[k] = F_1[k] + W_N^k F_2[k] + W_N^{2k} F_3[k] + W_N^{3k} F_4[k] \quad (2.28)$$

$$X[k + N/4] = F_1[k] - jW_N^k F_2[k] - W_N^{2k} F_3[k] + jW_N^{3k} F_4[k] \quad (2.29)$$

$$X[k + 2N/4] = F_1[k] - W_N^k F_2[k] + W_N^{2k} F_3[k] - W_N^{3k} F_4[k] \quad (2.30)$$

$$X[k + 3N/4] = F_1[k] + jW_N^k F_2[k] - W_N^{2k} F_3[k] - jW_N^{3k} F_4[k] \quad (2.31)$$

其中 $f_1[m] = x[4m]$ 、 $f_2[m] = x[4m+1]$ 、 $f_3[m] = x[4m+2]$ 、 $f_4[m] = x[4m+3]$ 且 $F_1[k]$ 、 $F_2[k]$ 、 $F_3[k]$ 、 $F_4[k]$ 分別為其傅立葉轉換。

同理，可將 $F_1[k]$ 、 $F_2[k]$ 、 $F_3[k]$ 與 $F_4[k]$ 等 $N/4$ 點的離散傅立葉轉換繼續分解，逐次分解下去，則可得 radix-4 分時快速傅立葉轉換。下圖(2.8)為十六點之 radix-4 分時快速傅立葉轉換流程圖。

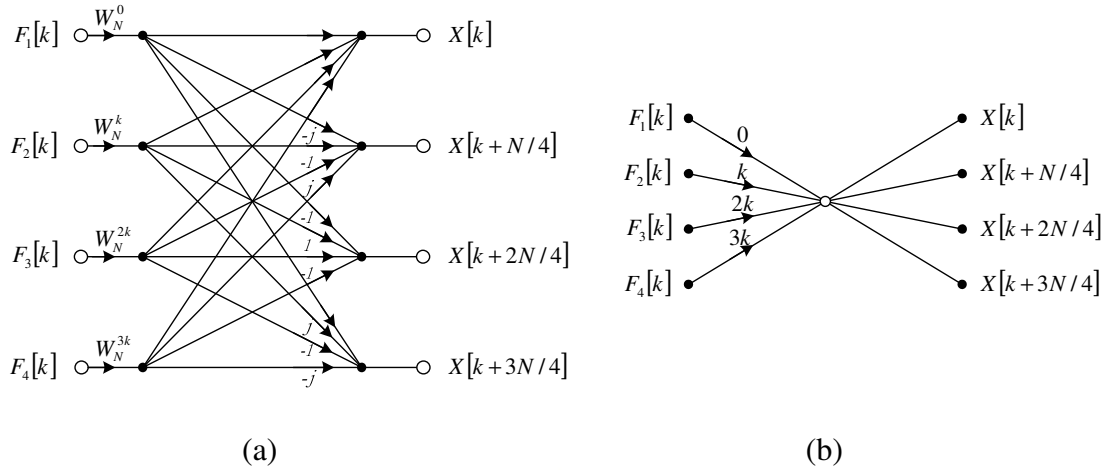


圖 2.7：(a) Radix-4 FFT 基本 butterfly，(b) Radix-4 butterfly 簡圖

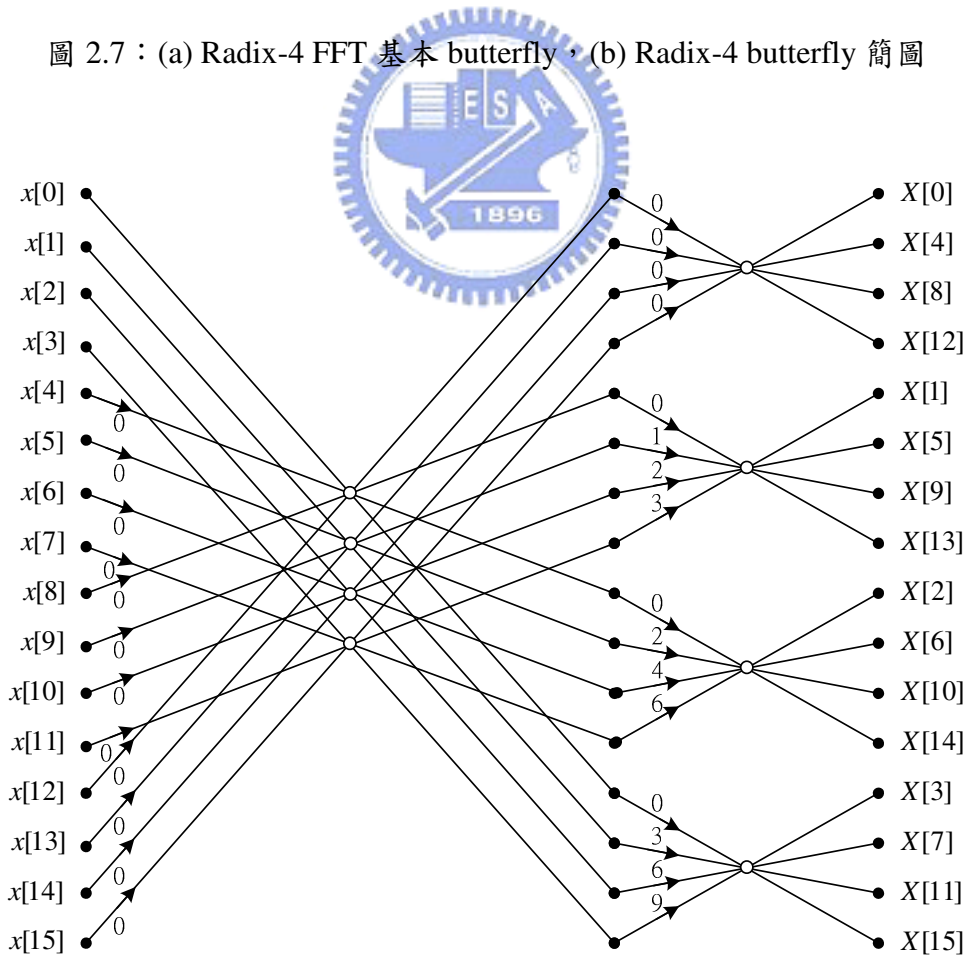


圖 2.8：16 點之 radix-4 分時快速傅立葉轉換流程圖

一個長度為 N 之 radix-4 DIT FFT 的計算複雜度：

$$M(N) = (3N/8) \log_2 N \quad (2.32)$$

$$A(N) = (3N/2) \log_2 N \quad (2.33)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

2.3.2 Radix-4 IFFT 演算法

Decimation-in-time 演算法

一個長度為 N 的離散序列 $x[n]$ ，其反向離散傅立葉轉換可以表示為：

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-nk}, n = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.34)$$

與 radix-4 DIT FFT 一樣，利用下列四式，我們可以得到 radix-4 的基本 butterfly 運算。下圖 2.9(a) 為 radix-4 DIT IFFT butterfly 之圖示，下圖 2.9(b) 為其簡圖。

$$x[n] = f_1[n] + W_N^{-n} f_2[n] + W_N^{-2n} f_3[n] + W_N^{-3n} f_4[n] \quad (2.35)$$

$$x[n + N/4] = f_1[n] + jW_N^{-n} f_2[n] - W_N^{-2n} f_3[n] - jW_N^{-3n} f_4[n] \quad (2.36)$$

$$x[n + 2N/4] = f_1[n] - W_N^{-n} f_2[n] + W_N^{-2n} f_3[n] - W_N^{-3n} f_4[n] \quad (2.37)$$

$$x[n + 3N/4] = f_1[n] - jW_N^{-n} f_2[n] - W_N^{-2n} f_3[n] + jW_N^{-3n} f_4[n] \quad (2.38)$$

其中 $F_1[k] = X[4k]$ 、 $F_2[k] = X[4k+1]$ 、 $F_3[k] = X[4k+2]$ 、 $F_4[k] = X[4k+3]$ 且

$f_1[n]$ 、 $f_2[n]$ 、 $f_3[n]$ 、 $f_4[n]$ 分別為其反向傅立葉轉換。

同理，可將 $f_1[n]$ 、 $f_2[n]$ 、 $f_3[n]$ 與 $f_4[n]$ 等 $N/4$ 點的反向離散傅立葉轉換繼續分解，逐次分解下去，則可得 radix-4 反向分時快速傅立葉轉換。下圖(2.10)為十六點之 radix-4 反向分時快速傅立葉轉換流程圖。

一個長度為 N 之 radix-4 DIT IFFT 的計算複雜度：

$$M(N) = (3N/8) \log_2 N \quad (2.39)$$

$$A(N) = (3N/2) \log_2 N \quad (2.40)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

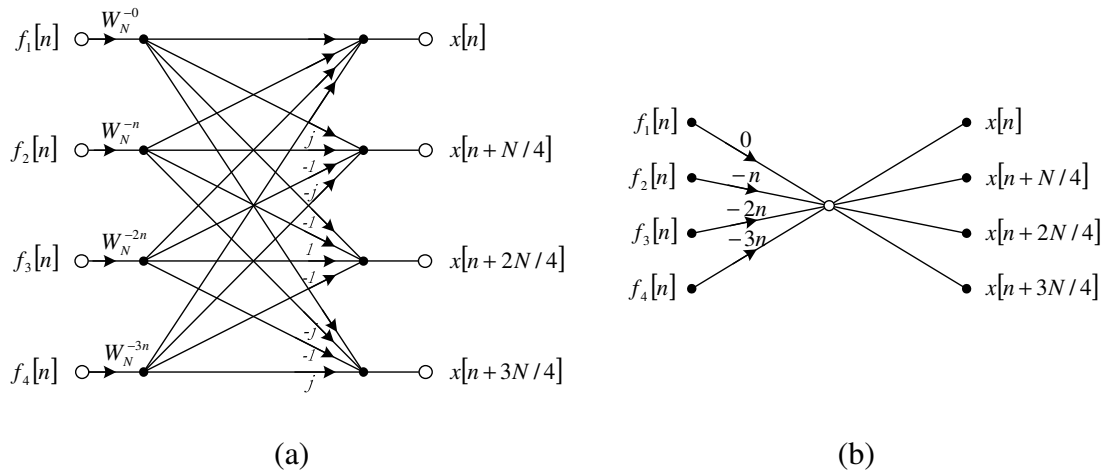


圖 2.9 : (a) Radix-4 IFFT 基本 butterfly，(b) Radix-4 butterfly 簡圖

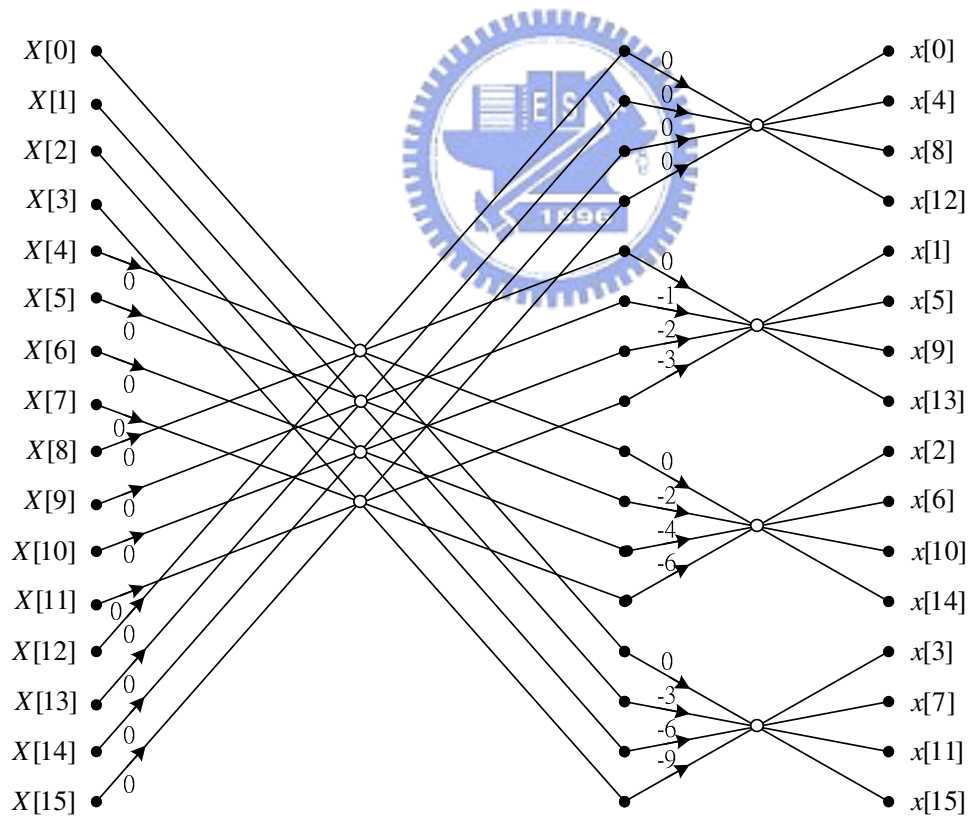


圖 2.10 : 16 點之 radix-4 反向分時快速傅立葉轉換流程圖

2.4 Radix-2² FFT 演算法

一個長度為 N 的離散序列 $x[n]$ ，其離散傅利葉轉換可以表示為：

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, 2, \dots, N-1 \quad \text{其中 } W_N = e^{-j(2\pi/N)} \quad (2.41)$$

對上式採用 DIF 演算法分解，令 $n = \frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3$ 、 $k = k_1 + 2k_2 + 4k_3$ 代入上式(2.41)，則：

$$\begin{aligned} X[k_1 + 2k_2 + 4k_3] &= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^1 \sum_{n_1=0}^1 x\left[\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3\right] W_N^{\left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3\right)(k_1 + 2k_2 + 4k_3)} \\ &= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^1 \left\{ B_{N/2}^{k_1} \left(\frac{N}{4}n_2 + n_3\right) W_N^{\left(\frac{N}{4}n_2 + n_3\right)k_1} \right\} W_N^{\left(\frac{N}{4}n_2 + n_3\right)(2k_2 + 4k_3)} \end{aligned} \quad (2.42)$$

$$\text{其中 } B_{N/2}^{k_1} \left(\frac{N}{4}n_2 + n_3\right) = x\left(\frac{N}{4}n_2 + n_3\right) + (-1)^{k_1} x\left(\frac{N}{4}n_2 + n_3 + \frac{N}{2}\right) \quad (2.43)$$

我們可將 twiddle factor $W_N^{\left(\frac{N}{4}n_2 + n_3\right)(k_1 + 2k_2 + 4k_3)}$ 將其分解成式(2.44)，之後代入式(2.42)中並把 $\sum_{n_2=0}^1$ 展開可得式(2.45)。

$$\begin{aligned} W_N^{\left(\frac{N}{4}n_2 + n_3\right)(k_1 + 2k_2 + 4k_3)} &= W_N^{N n_2 k_3} W_N^{\frac{N}{4} n_2 (k_1 + 2k_2)} W_N^{n_3 (k_1 + 2k_2)} W_N^{4 n_3 k_3} \\ &= (-j)^{n_2 (k_1 + 2k_2)} W_N^{n_3 (k_1 + 2k_2)} W_N^{4 n_3 k_3} \end{aligned} \quad (2.44)$$

$$X[k_1 + 2k_2 + 4k_3] = \sum_{n_3=0}^{\frac{N}{4}-1} \left[H(k_1, k_2, n_3) W_N^{n_3 (k_1 + 2k_2)} \right] W_N^{n_3 k_3} \quad (2.45)$$

其中

$$H(k_1, k_2, n_3) = \underbrace{\left\{ x[n_3] + (-1)^{k_1} x[n_3 + N/2] \right\}}_{BF \text{ I}} + (-j)^{(k_1 + 2k_2)} \underbrace{\left\{ x[n_3 + N/4] + (-1)^{k_1} x[n_3 + 3N/4] \right\}}_{BF \text{ II}} \quad (2.46)$$

由上式(2.46)可知， $H(k_1, k_2, n_3)$ 可分成 BF I 和 BF II 兩個階段的 butterflies 來計算，圖(2.11)為十六點之 radix-2² 分頻離散傅立葉轉換分解成四個四點的分頻離散傅立葉轉換流程圖，可將圖(2.11)繼續分解下去可得十六點之 radix-2² 分頻快速傅立葉轉換流程圖，圖(2.12)。

一個長度為 N 之 radix-2² DIF FFT 之計算複雜度：

$$M(N) = (3N/8) \log_2 N \quad (2.47)$$

$$A(N) = N \log_2 N \quad (2.48)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

Radix-2² 演算法的優點在於它有和 radix-4 演算法一樣的複數乘法計算複雜度，且也仍然保留著 radix-2 的 butterfly 架構。

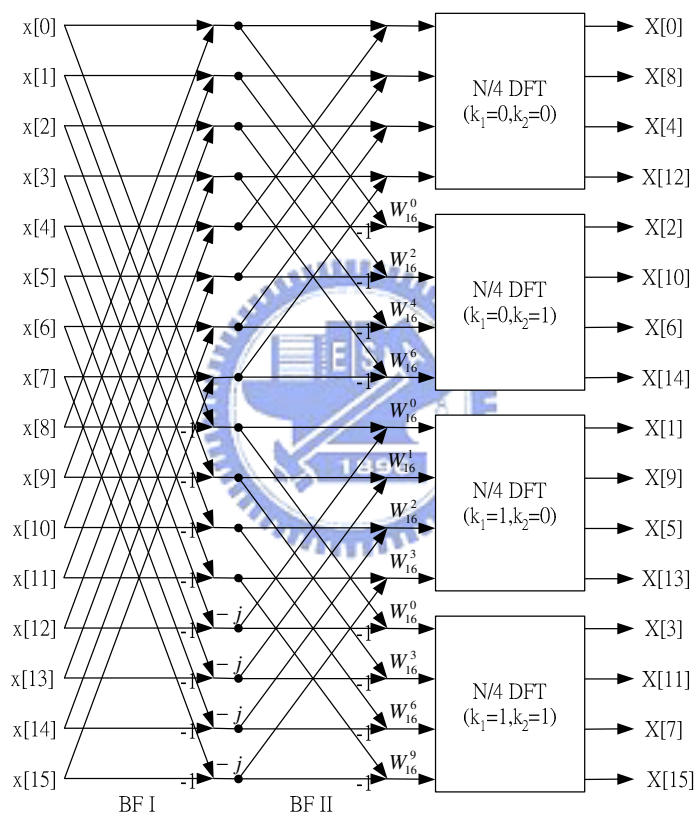


圖 2.11：16 點 radix-2² 分頻離散傅立葉轉換分解流程圖

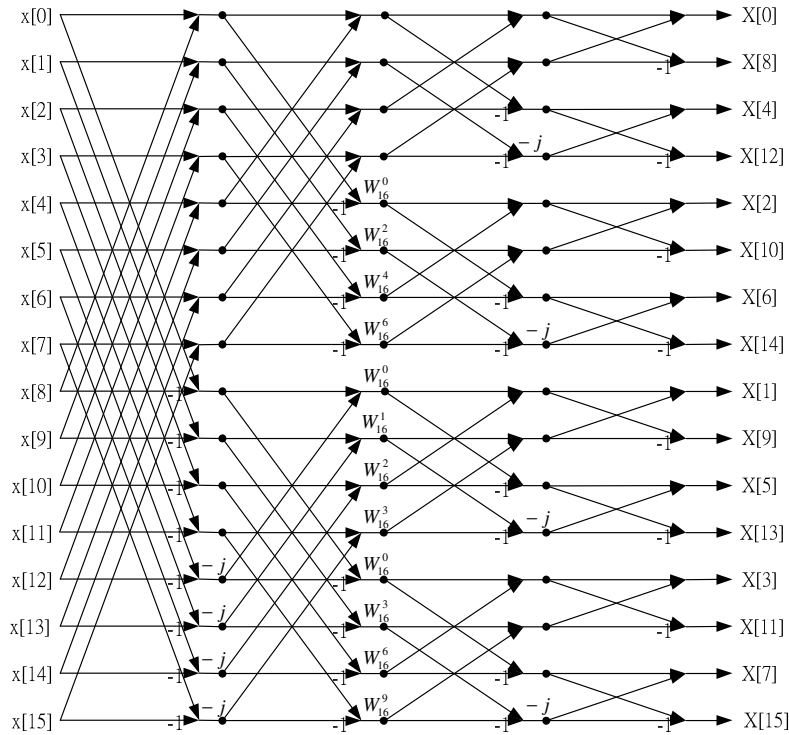


圖 2.12：16 點之 radix-2² 分頻快速傅立葉轉換流程圖

2.5 Split-Radix 2/4 FFT 演算法

我們可以藉由將 radix-2 DIF 分解中的偶數頻率項與 radix-4 DIF 解中的奇數頻率項結合，來獲得較低的計算複雜度。此方式稱為 split-radix 2/4 DIF 演算法，其基本分解如下所示。

$$X(2r) = \sum_{n=0}^{N/4-1} (x[n] + x[n + 2N/4]) W_{N/2}^{nr} \quad (2.49)$$

$$X(4s + 1) = \sum_{n=0}^{N/4-1} [x[n] - x[n + 2N/4] - j(x[n + N/4] - x[n + 3N/4])] W_N^n W_N^{4ns} \quad (2.50)$$

$$X(4s + 3) = \sum_{n=0}^{N/4-1} [x[n] - x[n + 2N/4] + j(x[n + N/4] - x[n + 3N/4])] W_N^{3n} W_N^{4ns} \quad (2.51)$$

其中 $0 \leq r \leq N/2 - 1$ 、 $0 \leq s \leq N/4 - 1$ 。圖(2.13)為 split-radix 2/4 DIF butterfly 之圖示，圖(2.14)為 16 點 Split-Radix 2/4 分頻快速傅立葉轉換流程圖。

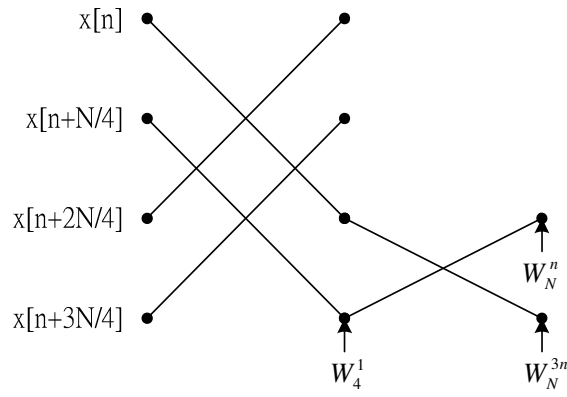


圖 2.13 : Split-radix 2/4 DIF 基本 butterfly

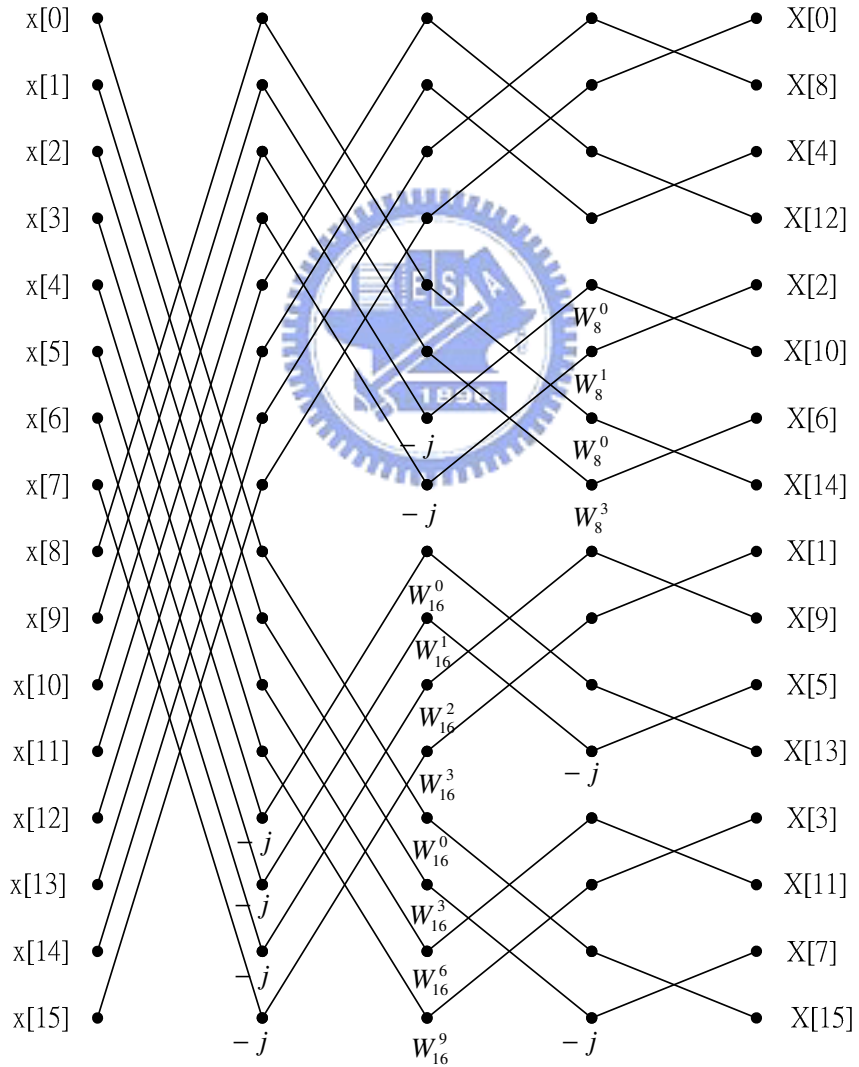


圖 2.14 : 16 點 Split-radix 2/4 分頻快速傅立葉轉換流程圖

一個長度為 N 之 radix-2² DIF FFT 之計算複雜度：

$$M(N) = [(3\log_2 N - 2) \times N + 2 \times (-1)^{\log_2 N}] / 9 \quad (2.52)$$

$$A(N) = N \log_2 N \quad (2.53)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

2.6 結論

在本章中，我們分析了一些不同 radix 的 FFT 和 IFFT 演算法，這些演算法的目的都是為了使離散傅立葉轉換可以被快速的計算出來。表(2.1)為各演算法的所需複雜度。從表(2.1)中，我們可以知道 high-radix 有較低的乘法複雜度，對於想設計高速運算 FFT 的處理器來說，是個不錯的選擇。此外，split-radix 演算法雖然具有較 fixed radix 低的計算複雜度，但相較於 fixed radix 來說 split-radix 較難有規律性。



complexity algorithm	No. of complex multiplications	No. of complex additions
Direct computation	N^2	$N^2 - N$
Radix-2	$(N/2)\log_2 N$	$N \log_2 N$
Radix-4	$(3N/8)\log_2 N$	$(3N/2)\log_2 N$
Radix-2 ²	$(3N/8)\log_2 N$	$N \log_2 N$
Split-Radix 2/4	$[(3\log_2 N - 2) \times N + 2 \times (-1)^{\log_2 N}] / 9$	$N \log_2 N$

表 2.1：各演算法計算複雜度之比較

第三章 FFT/IFFT 架構

3.1 簡介

隨著快速傅立葉轉換演算法的演變，以及半導體製程技術不斷的進步，使得快速傅立葉轉換已可以被廣泛的應用且實現在數位訊號處理以及通訊系統上。並且根據這些不同的演算法，也衍生出許多不同的硬體實現架構。一般來說，常見的 FFT 硬體架構實現方式，大致可分為兩類，分別為 pipeline-based 架構與 memory-based 架構[5][7][8][9][10][11]。Pipeline-based 架構的最大優點，在於有非常高的 throughput rate 以及非常規律性的硬體實現方式。而 memory-based 架構，通常只需利用到一個 radix-r butterfly 運算器，所以可以達到非常低的硬體需求。且藉由單一 radix-r butterfly 運算器的運算方式，可以很容易地設計出可變長度快速傅立葉轉換處理器，但其缺點在於偏低的 throughput rate。且若處理器是採 fixed point 架構來實現，則在不增加內部運算資料字元長度的狀況下，當換點數為大點數轉換時其所得運算結果的精確度甚糟，一種改善精確度的方法為使用 block scaling 的方式[12]，此方式可在不增加內部運算資料字元長度的行情下提升精確度。

本章將敘述常見的 pipeline-based 架構和 memory-based 架構的幾種不同做法，並比較各個做法的優缺。

3.2 Pipeline 架構

在快速傅立葉處理器的硬體實現方面，最常見的做法就是 pipeline 架構的做法[5][8][9]。因為此架構具有很高的 throughput rate，很適合應用於需要高速傅立葉轉換的系統中，加上此架構的硬體複雜度並不高，所以可以很容易實現於硬體上。但其硬體的 demand 會依據轉換點數的變大而增加硬體，且硬體的使用率也會根據不同的架構而有所不同，對於欲設計一個可變長度的快速傅立葉處理器來說，硬體的 demand 是蠻大的。本節將介紹運用 DIF FFT 演算法所衍生出的幾種不同 pipeline 架構，並討論其硬體的使用率。pipeline 架構可細分為兩類，其一為 delay-commutator 架構，另一種為 delay-feedback 架構。兩架構不同之處在於

delay-commutator 架構有較高的 throughput rate，而 delay-feedback 架構則是硬體需求較低且也有較好的硬體使用率。

3.2.1 Multiple-Path Delay Commutator (MDC) Pipeline Architecture

此方式主要是將輸入序列適當地分成 r 個平行序列（ r 依據 radix- r 來決定），並且將每個序列經過適當的延遲後輸入 butterfly 運算器，藉此以達到正確的計算方式。此做法可得到較高的 throughput rate，但所需的記憶體與複數乘法器等硬體需求也較高。

● Radix-2 MDC

對於 radix-2 FFT 演算法來說，R2MDC 算是一個非常易懂的 pipeline 實現架構。將輸入序列分成兩個平行的資料流，再經由適當的延遲後進入 butterfly 運算器運算。圖(3.1)為一個具有 pipeline 架構的十六點 R2MDC 快速傅立葉處理器。其中每個 butterfly 運算器與複數乘法器的使用率均為 50%，共需要 $\log_2 N - 2$ 個複數乘法器， $\log_2 N$ 個 radix-2 butterfly 運算器和 $3/2N - 2$ 個暫存器。

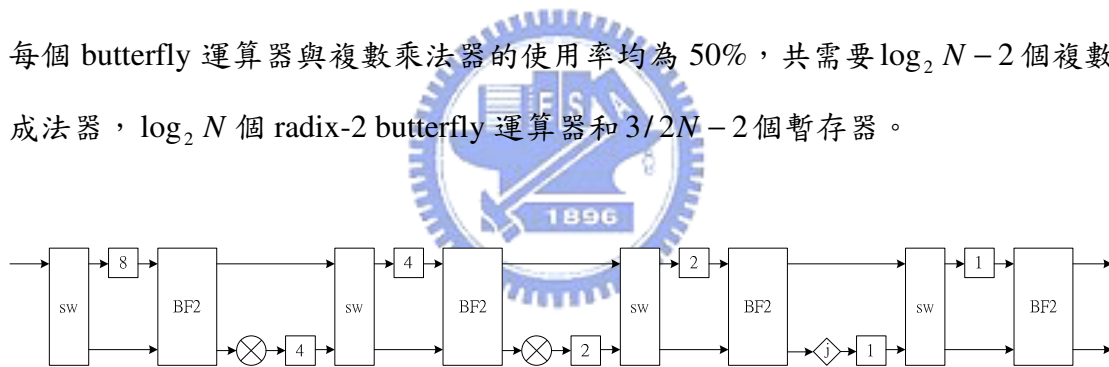


圖 3.1：16 點 R2MDC 快速傅立葉處理器架構圖

● Radix-4 MDC

R4MDC 架構跟 R2MDC 架構的原理很像，主要不同之處在 radix 的不同。對於欲計算一個 N 點 FFT 而言，R4MDC 架構有較 R2MDC 架構快的運算速度，然而它所需要的硬體複雜度相對的也較高。此架構的每個 butterfly 運算器與複數乘法器使用率只有 25%，且需要 $3\log_4 N - 3$ 個複數乘法器， $\log_4 N$ 個 radix-4 butterfly 運算器和 $5/2N - 4$ 個暫存器。圖(3.2)為一個具有 pipeline 架構的 256 點 R4MDC 快速傅立葉處理器。

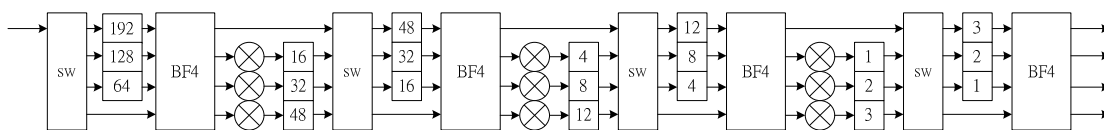


圖 3.2：256 點 R4MDC 快速傅立葉處理器架構圖

● Radix-4 Single-Path Delay Commutator

Radix-4 Single-path Delay Commutator 是使用一個改良式的 radix-4 演算法，將原來 radix-4 butterfly 運算器改為四分之一可程式化的 radix-4 butterfly 運算器(每次運算只做 radix-4 butterfly 運算器的其中一個輸出)，使得每一個 butterfly 運算器只需銜接一個複數乘法器在其後面即可，故所需複數乘法器數目為 $\log_4 N - 1$ ，且複數乘法器的使用率也提高到 75%。且此架構也將所需暫存器之個數從 $5/2N - 4$ 減至 $2N - 2$ 。圖(3.3)為一個具有 pipeline 架構的 256 點 Radix-4 Single-path Delay Commutator 快速傅立葉處理器。

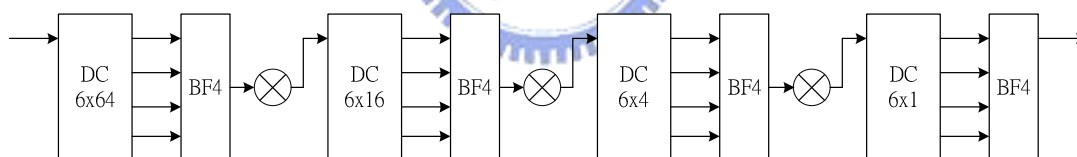


圖 3.3：256 點 R4SDC 快速傅立葉處理器架構圖

3.2.2 Single-Path Delay Feedback (SDF) Pipeline Architecture

本節將介紹 pipeline 架構的另一種硬體實現方式。此方式稱為 Single-path Delay Feedback(SDF)方式，此方式可適用於不同的 radix FFT 演算法。此架構的處理方式，主要是將輸入序列經過迴授位移暫存器的適當延遲後，再交由 butterfly 運算器處理。SDF 架構有極高的儲存元件使用率(100%)和較高複數乘法器使用率，其經過運算後所獲得輸出序列之順序為 digit-reversed(輸入序列為 in-order)。

● Radix-2 SDF

R2SDF 主要是利用迴授位移暫存器來減少暫存器的使用數目，以達到更有效率地使用暫存器之目的。Butterfly 運算器將其部分輸出存回一個迴授位移暫存器內，並在每個階段中，都只會有一個從 Butterfly 運算器輸出的資料經過複數乘法器，所以此架構所使用的複數乘法器及 Butterfly 運算器的個數與 R2MDC 相同。但此架構可將暫存器個數減至只需 $N-1$ 個暫存器，其記憶體的需要量是最小的。複數乘法器的使用率為 50%。圖(3.4)為一個具有 pipeline 架構的十六點 Radix-2 Single-path Delay feedback 快速傅立葉處理器。

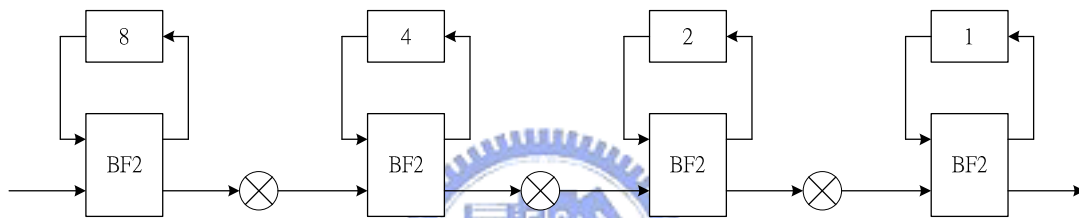


圖 3.4：16 點 R2SDF 快速傅立葉處理器架構圖

● Radix-4 SDF

同樣地，R4SDF 架構的操縱方式與 R2SDF 架構相似。因為 R4SDF 架構是以 radix-4 butterfly 處理器來運作，所以其 butterfly 處理器有四個輸出資料，將其中三個輸出存回迴授暫存器內，並把剩餘的一個輸出資料經過複數乘法器運算，此方式可讓複數乘法器的使用率增為 75%。然而，radix-4 butterfly 運算器相對的也較複雜，裡面至少有八個加法器，其加法使用率僅 25%。所需硬體單元，需要 $\log_4 N - 1$ 個複數乘法器， $\log_4 N$ 個 radix-4 butterfly 運算器和 $N - 1$ 個迴授位移暫存器。圖(3.5)為一個具有 pipeline 架構的 256 點 Radix-4 Single-path Delay feedback 快速傅立葉處理器。

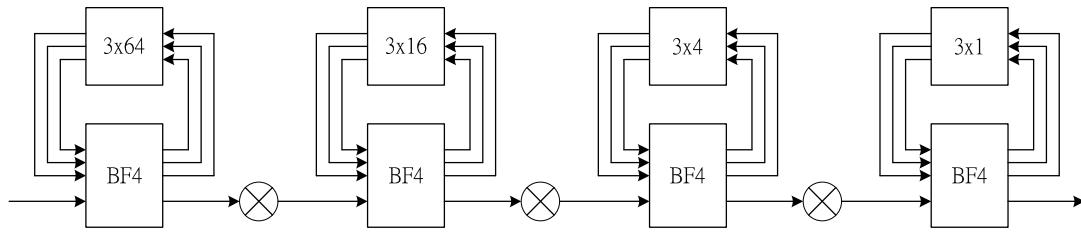


圖 3.5：256 點 R4SDF 快速傅立葉處理器架構圖

● Radix-2² SDF

根據 radix-2² DIF FFT 演算法，我們可以得到一個非常類似 R2SDF 的 R2²SDF 架構。此架構的 butterfly 處理器有兩種形式 BF2 I 和 BF2 II，在每階段中 BF2 I 和 BF2 II 運算器都會將兩輸出的其中之一存回迴授位移暫存器內，另一個輸出則為下一階段的輸入，且因為只有在 BF2 II 處理器那階段的輸出資料才需要經過複數乘法器運算，故此架構使用到的複數乘法器數目較 R2SDF 架構少。此架構的複數乘法器的使用率增與 R4SDF 同(75%)，所需硬體單元為 $\log_4 N - 1$ 個複數乘法器， $\log_2 N$ 個 radix-2 butterfly 運算器和 $N - 1$ 個迴授位移暫存器。圖(3.6)為一個具有 pipeline 架構的十六點 Radix-2² Single-path Delay feedback 快速傅立葉處理器。

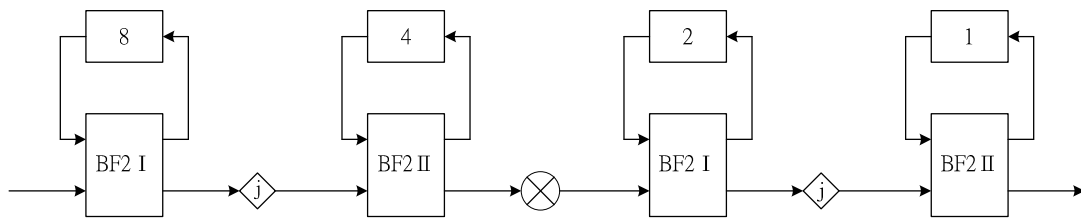


圖 3.6：16 點 R2²SDF 快速傅立葉處理器架構圖

3.2.3 Convergent Block Floating Point Pipeline Architectures

一般來說，pipeline FFT 處理器最簡易也最常見的做法都是採用 fixed point 架構來實現。然而，我們都知道因為 fixed point 的字元長度有限，所以 fixed point FFT 處理器最重要的問題在於其計算後的精確度問題。一種可以達到較高的訊號雜訊比方式為在 pipeline FFT 的每個階段中增加其內部的字元長度，但如此的話，我

們知道所得到的輸出字元長度將會較輸入字元長度寬。另一種改善訊號雜訊比且不需增加其內部字元長度的方式為使用 block scaling 的方式[12]。Block scaling 這方式是利用類似 floating point 的概念，它將每階段 fixed point 形式的輸入資料，依所在的不同階段分成不同的 block 群組並搭配相對應的 scaling factor，以此類似 floating point 的概念來獲得較高的精確度。這裡將提到兩種 Block scaling 方式，分別為 Block Floating Point (BFP)與 Convergent Block Floating Point (CBFP)。

Block Floating Point (BFP)，此方法是以每個階段的運算資料都屬於同一個 block 群組的概念，先將每階段 block 群組所運算出的最大輸出值偵測出來，找到每階段的 scaling factor 後，再將所有的輸出值經過相同地 scaled 來當作下一階段的輸入，以此方式來改善精確度。圖(3.7)為 16 點 radix-2 DIF FFT 之一個例子。但此方式不太適用於 pipeline FFT 架構，因為 BFP 在下一階段開始之前所需的 scaling factor，是必須等到前一階段運算完畢才知道其 scaling factor，這將會導致 pipeline FFT 架構的輸出 latency 過長。

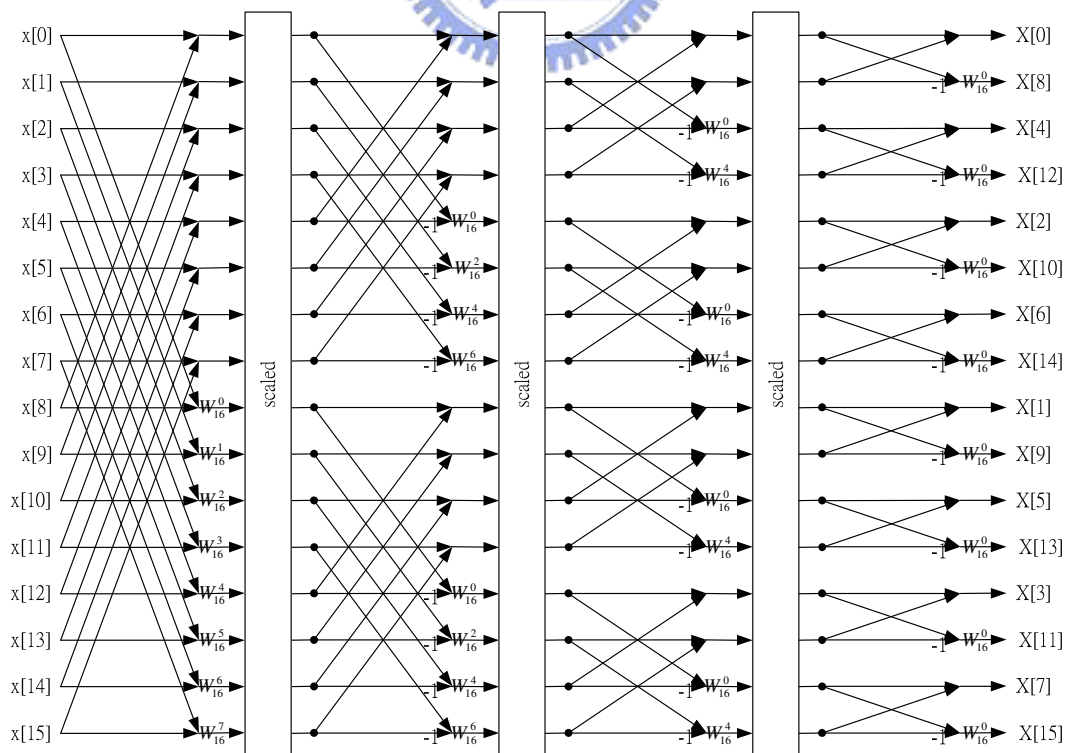


圖 3.7：16 點 radix-2 DIF FFT 之 BFP 架構流程圖

若要等每個階段算完後才可得到下一階段所需的 scaling factor，對 pipeline FFT 架構來說，此方式是不可行的。所幸，有另一種可行且可應用在 pipeline FFT 架構的改良方法為 Convergent Block Floating Point (CBFP) 方式。此方式主要的想法，以 radix-2 FFT 演算法為例，是將 radix-2 每階段的輸出，從第一階段開始分成兩個各具有不同 scaling factor 的輸出獨立 block 群組，經過適當的 butter 之後，再把此兩個具有不同 scaling factor 的輸出獨立 block 群組 scaled，當做第二階段的輸入。經過第二階段後可得到 4 個各具有不同 scaling factor 的輸出獨立 block 群組，然後再將其 scaled，依此方式處理下去。算到最後，我們可以發現 FFT 的每個輸出 block 群組都各具有一個專屬的 scaling factor。圖(3.8)為 16 點 radix-2 DIF FFT 之一個例子。圖(3.9)為一個具有 CBFP pipeline 架構的十六點 Radix-2 Single-path Delay feedback 快速傅立葉處理器。

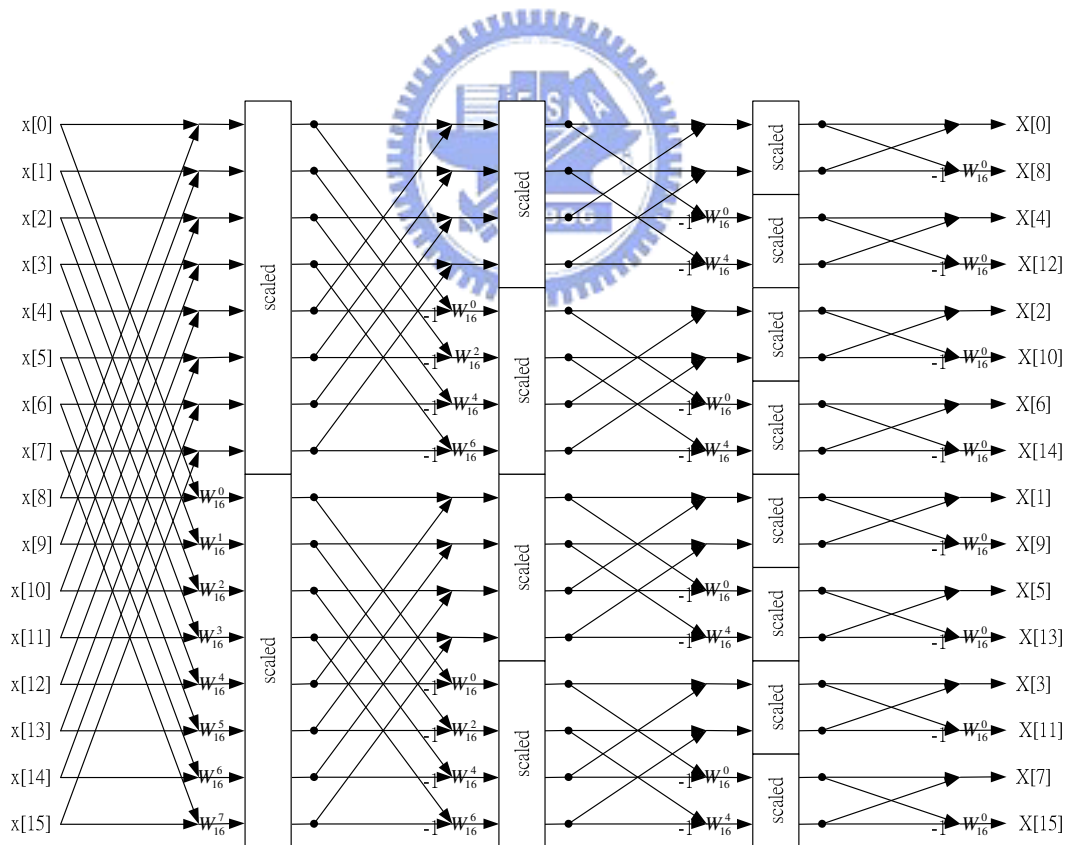


圖 3.8：16 點 radix-2 DIF FFT 之 CBFP 架構流程圖

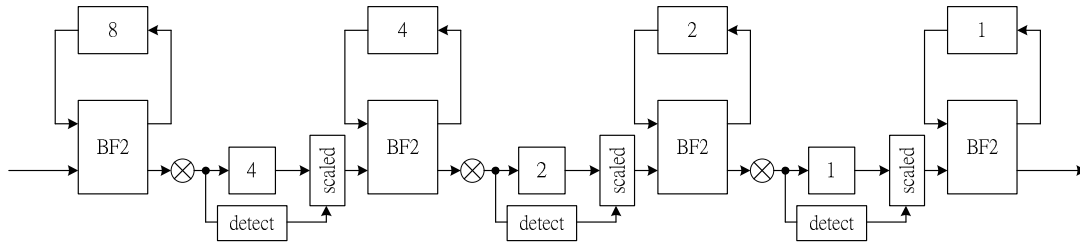


圖 3.9：CBFP pipeline 架構的十六點 R2SDF 快速傅立葉處理器架構圖

CBFP 的優點在於，可在不增加輸出字元長度的狀況下，讓輸出訊號雜訊比獲得改善。即使當欲轉換之輸入訊號值過小時，CBFP 也會在第一個階段就將其輸入訊號 scaled 至適當的準位，以增加其运算精確度。但相對的 CBFP 需耗費較多的記憶體且其輸出 latency 也將會延長。

3.2.4 不同 Pipeline 架構的比較

我們將針對本節所述的各種 pipeline FFT 處理器架構，radix-2 multiple-path delay commutator、radix-4 multiple-path delay commutator、radix-2 single-path delay feedback、radix-4 single-path delay feedback、radix-2² single-path delay feedback 以及 CBFP pipeline 架構的 radix-2 single-path delay feedback 等架構，比較其乘法器、butterfly 運算器和暫存器的所需數目及其使用率。從表(3.1)與表(3.2)中，我們可以知道 delay feedback 架構較 delay commutator 架構所需的硬體需求較少，且其各硬體的使用率也較 delay commutator 架構的各硬體使用率高。但我們由其架構來看，很明確地，delay commutator 具有較高的 throughput rate。此外，雖然使用 CBFP 架構可以改善 fixed point 的精確度問題，但相對的其記憶體需求較原本架構增加 1.5 倍。

Architecture	No. of Complex Multipliers	No. of Complex Adders	Memory size	control
R2MDC	$2(\log_4 N - 1)$	$4\log_4 N$	$3N/2 - 2$	simple
R4MDC	$3(\log_4 N - 1)$	$8\log_4 N$	$5N/2 - 4$	simple
R4SDC	$\log_4 N - 1$	$3\log_4 N$	$2N - 2$	complex
R2SDF	$2(\log_4 N - 1)$	$4\log_4 N$	$N - 1$	simple
(CBFP) R2SDF	$2(\log_4 N - 1)$	$4\log_4 N$	$3N/2 - 2$	simple
R4SDF	$\log_4 N - 1$	$8\log_4 N$	$N - 1$	medium
R2 ² SDF	$\log_4 N - 1$	$4\log_4 N$	$N - 1$	simple

表 3.1：各不同 pipeline 架構之硬體需求比較

Architecture	Utilization rate of Multipliers	Utilization rate of Adders	Utilization rate of Registers
R2MDC	50%	50%	50%
R4MDC	25%	20%	25%
R2SDF	50%	50%	100%
(CBFP) R2SDF	50%	50%	100%
R4SDF	75%	25%	100%
R2 ² SDF	75%	50%	100%

表 3.2：各不同 pipeline 架構之硬體使用率比較

3.3 Memory-based 架構

另外一個有別於 pipeline 架構的常見做法為 memory-based 架構[10][11]。memory-based 架構可達到非常低的硬體需求，通常只需利用到一個 radix-r butterfly 處理器來運算所有的 radix-r butterfly。如同圖(3.10)所示，圖(3.10)為 16 點 DIT FFT 的流程圖，其 butterfly 處理器的運算順序為，從每階段的第一個 butterfly 開始計算(由上而下)，算完這階段的 butterfly 後接著算下一階段(從左至右)，依此方式運算下去。此方式所需硬體運算單元較 pipeline 架構少許多。且藉由單一 radix-r butterfly 處理器的運算方式，可以很容易地設計出可變長度快速傅立葉轉換處理器。此外，因為 memory-based 架構對於演算法的選擇需要有非常

好的規律性，故不適用於 split-radix 等規律性較差的演算法，常用於 fixed radix 等有規律性的演算法的實現。

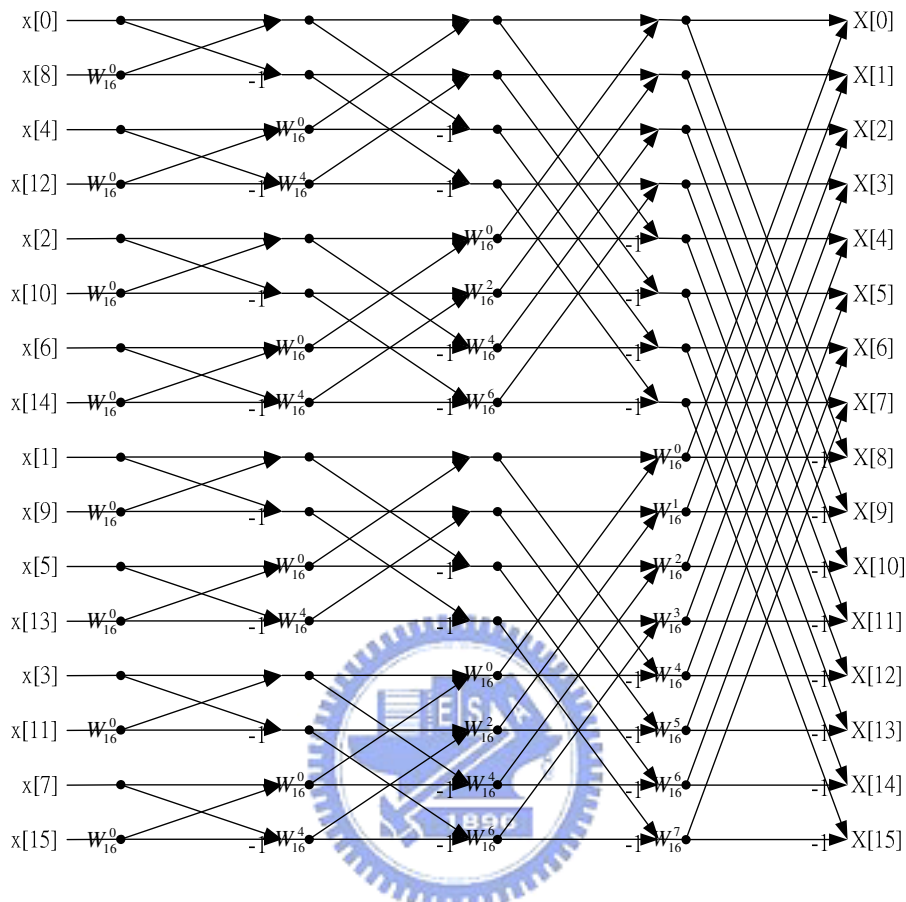


圖 3.10：16 點 DIT FFT 流程圖

與 pipeline 架構一樣，memory-based 架構也需要記憶體來儲存 FFT 運算時的值，其 radix-r butterfly 處理器每次運算所需之 twiddle factor 都來自於唯讀記憶體中。又因為在運算時，我們知道儲存運算資料的記憶體一直是在被使用的狀況，所以若想要處理連續的 FFT 轉換，則需格外有另一套記憶體來暫存住。另外，memory-based 架構又可分為單記憶體架構與雙記憶體架構。單記憶體架構的記憶體主要是採 in-place 模式，也就是說經由 butterfly 運算器運算後之輸出值所存回記憶體的位址為當初所擷取的位址，圖(3.11)為單記憶體 memory-based 架構的方塊圖。而雙記憶體架構的存取模式主要是利用兩組記憶體的切換存取，使運算資料能不互衝地讓 butterfly 運算器運算，圖(3.12)為雙記憶體 memory-based 架構

的方塊圖。很明顯地，採 in-place 模式的單記憶體架構其硬體所需之記憶體較雙記憶體架構可少一半的大小。

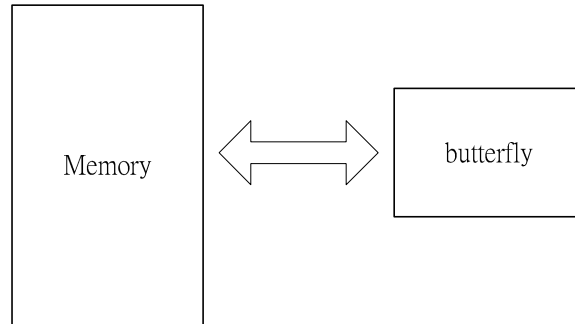


圖 3.11：單記憶體 memory-based 架構快速傅立葉轉換處理器方塊圖

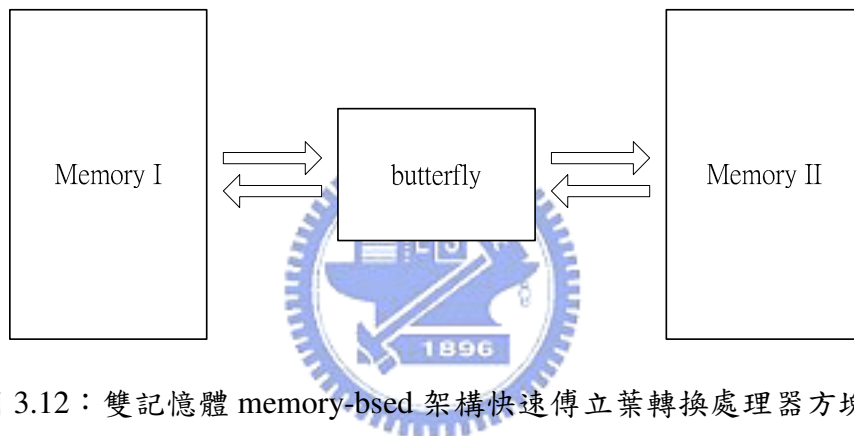


圖 3.12：雙記憶體 memory-based 架構快速傅立葉轉換處理器方塊圖

一般來說，一個長度為 N 的 FFT 轉換需要經過 $\frac{N}{r} \times \log_r N$ 個 radix-r butterfly 運算，且每個 radix-r butterfly 運算必須經由 $2r$ 次的記憶體讀寫來完成。所以對於一個要運算長度為 N 的 memory based FFT/IFFT 處理器來說，總共需要的記憶體讀寫次數為 $2N \times \log_r N$ 次。然而，就一個有效率的 memory based FFT/IFFT 處理器來說，其記憶體讀寫所需花費的總 clock cycle 數越少越能達到其運算效率。有兩種方式可以減少其記憶體的讀寫所需的總 clock cycle 數：

1. 使用 high-radix 的架構，此方式可降低讀寫次數，但相對的會增加其硬體複雜度。
2. 藉由增加記憶體擷取頻寬來減少讀寫時所需的總 clock cycle 數。

增加記憶體擷取頻寬的方式有兩種，一為使用可以同時存取多筆資料的單一 multiple-port 記憶體來達到，但此方式會讓硬體面積增大；另一種方式，則是將記憶體分割成多組 (r) single-port 記憶體來達到同時讀取 r 筆資料和同時寫回 r 筆資料的目的。

3.4 結論

在本章中，我們討論到了兩個最常被用來實現 FFT/IFFT 演算法的架構，分別為 pipeline 架構與 memory-based 架構。

Pipeline 架構又可分為 MDC 或 SDF 兩方式來實現，其中 MDC 的 throughput rate 較高，SDF 所需要記憶體與計算單元需求較低。另一個架構為 Memory-based 架構，此架構只需一個 radix- r butterfly 處理器來執行其運算，使我們可以很容易在不增加運算單元的狀況下，達到可變長度的 FFT/IFFT 運算，但其需要較多的 clock cycle 數來完成運算。表(3.3)為大點數轉換 pipeline 架構與 memory-based 架構的比較，其中關於記憶體大小部分， N 代表欲轉換的點數，且 memory-based 架構對於處理連續轉換時，需有格外的儲存元件(buffer)來儲存下次欲轉換的值。

FFT架構	記憶體大小	throughput rate	運算單元	SQNR
一般 Pipeline	$N-1$	高	多	差
(CBFP)Pipeline	$3N/2-2$	高	多	好
一般 Memory-based	$N+buffer$	低	少	差
(BFP)Memory-based	$N+buffer$	低	少	好

表 3.3：大點數快速傅立葉轉換之 pipeline 架構與 memory-based 架構比較

基本上，pipeline 架構非常適用於需要高 throughput rate 的系統中，但當轉換點數為大點數時，pipeline 架構的精確度將非常糟糕，可藉由逐級增加字元長度的方式來改善精確度，但相對的其運算單元及記憶體所處理的字元也得逐級變長，故硬體面積將會變大且硬體速度也會稍微降低。在不增加字元長度下，CBFP pipeline 架構可改善運算精確度，但需格外多付出記憶體來達到。而對於需要中、

低 throughput rate 且大點數的系統來說(如 ADSL、VDSL 等系統)，memory-based 架構是個不錯的選擇。因其運算單元並不會根據點數變大而增加，故可達到較低的硬體面積，並且可在不增加記憶體儲存字元長度下，使用 BFP 方式來改善精確度。



第四章 應用於實數時域之 FFT/IFFT 架構

4.1 簡介

雖然在高 throughput rate 的系統中，memory-based 架構很難比 pipeline 架構達到高 throughput rate 需求，但對於應用在中、低 throughput rate 系統中(如 ADSL、VDSL 等系統)，由於 memory-based 架構的 throughput rate 可以符合其需求且具有較低的硬體需求、較高的精確度(BFP)以及轉換點數可任意變動等優點，使得在中、低 throughput rate 系統中，memory-based 架構是個不錯的選擇。本論文的主題，在於實現一個低硬體需求且可變長度的快速傅立葉轉換處理器。而 memory-based 架構的硬體實現方式正好符合所需，故本論文將針對 memory-based 架構的快速傅立葉轉換處理器做更深入的探討。

本章將介紹可以應用於計算實數時域的 memory-based 架構處理器。此處理器對於處理實數 FFT 或 Hermitian Symmetric IFFT 可達到更有效率的處理方式，且也可以用於計算一般的 FFT 和 IFFT。本章首先將提到專門處理實數 FFT 的 Real-Valued FFT(RFFT)演算法，及專門處理 Hermitian Symmetric IFFT 的 Hermitian Symmetric IFFT(HS-IFFT) 演算法，此兩種演算法都是利用其輸入序列的特性來達到更有效率的運算[13][14][15]。之後將分別提到 radix-2 雙模可變長度 FFT/IFFT 架構、radix-4 雙模可變長度 FFT/IFFT 架構，最後在以多模可變長度 RFFT/ HS-IFFT 架構做個總結。

4.2 Real-Value FFT(RFFT)/Hermitian Symmetric IFFT

(HS- IFFT)演算法

4.2.1 RFFT 演算法

正如我們所知道的，欲得到長度為 N 之複數序列的離散傅立葉轉換(DFT)頻譜，可藉由快速傅立葉轉換(FFT)來達到有效率的運算。然而，大多數的 FFT 演算法都是被設計來計算複數序列的 DFT，但在有些應用(如 ADSL、VDSL 等系統)，

欲轉換的序列實際上是實數序列。其實，我們可以利用實數序列轉換後頻譜的一些對稱特性，來達到更有效率的運算。本小節將介紹針對實數序列離散傅立葉轉換的幾種廣為人知的方式。

法一：

若要用 N 點 Complex-valued FFT(CFFT)演算法來計算 N 點實數序列的離散傅立葉轉換。最簡單的方式就是把實數序列擴展成虛部為零的複數序列後，直接算其 N 點 CFFT 即可。如下圖(4.1)所示：

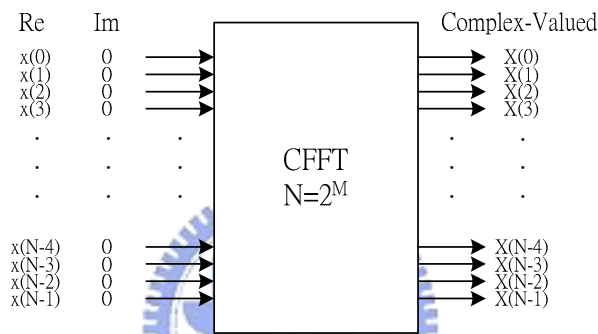


圖 4.1：實數序列之 CFFT 方塊圖

很明顯的，我們發現此作法所需要的計算複雜度，其實跟同長度複數序列的計算複雜度是一樣的。

法二：

第二種計算實數序列離散傅立葉轉換的方式，是運用一個 N 點的 CFFT 來同時處理兩個等長的 N 點實數序列離散傅立葉轉換。之後，再利用實數訊號轉換後的頻譜對稱特性加上一些後端的處理，即可獲得個別的離散傅立葉轉換頻譜。如圖(4.2)所示：

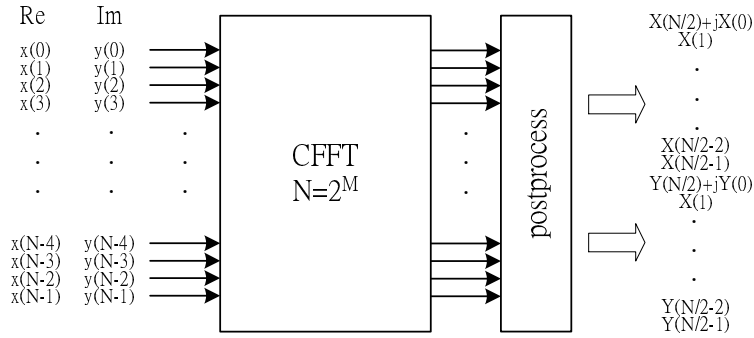


圖 4.2：雙實數序列之 CFFT 及後端處理方塊圖

若有一序列 $x[n]$ 為實數序列，則其轉換後的 $X[k]$ ，實部必為偶對稱，虛部必為奇對稱。所以假設現有兩個等長度為 N 的實數序列 $x[n]$ 與 $y[n]$ ，令一複數序列 $z[n]$ 其實部為 $x[n]$ 序列、虛部為 $y[n]$ 序列。

$$z[n] = x[n] + jy[n] \quad , n = 0, 1, \dots, N-1 \quad (4.1)$$

因為離散傅立葉轉換是線性運算，對 $z[n]$ 做傅立葉轉換後可得到：

$$\begin{aligned} Z[k] &= X[k] + jY[k] \\ &= \{X_r[k] - Y_i[k]\} + j\{X_i[k] + Y_r[k]\} \quad , k = 0, 1, \dots, N-1 \end{aligned} \quad (4.2)$$

其中下標 r 和 i 各別代表實部與虛部。藉由實數轉換的對稱性可得下式：

$$Z[N-k] = \{X_r[k] + Y_i[k]\} - j\{X_i[k] - Y_r[k]\} \quad (4.3)$$

從上兩個式子，很容易的我們可將 $X[k]$ 與 $Y[k]$ 分解出來。

$$\begin{aligned} X[k] &= X_r[k] + jX_i[k] \\ &= \frac{1}{2}\{Z_r[k] + Z_r[N-k]\} + j\frac{1}{2}\{Z_i[k] - Z_i[N-k]\} \quad , k = 0, 1, \dots, N/2 \end{aligned} \quad (4.4)$$

$$\begin{aligned} Y[k] &= Y_r[k] + jY_i[k] \\ &= \frac{1}{2}\{Z_i[N-k] + Z_i[k]\} + j\frac{1}{2}\{Z_r[N-k] - Z_r[k]\} \quad , k = 0, 1, \dots, N/2 \end{aligned} \quad (4.5)$$

就如之前所說的， $x[n]$ 與 $y[n]$ 皆為實數序列，傅立葉轉換後之 $X[k]$ 與 $Y[k]$ 有 Hermitian Symmetric 的特性。所以只需計算 $X[k]$ 與 $Y[k]$ 在區間 $0 \leq k \leq N/2$ 即可，且很明顯的可知在 $k=0$ 與 $k=N/2$ 時， $X_r[k] = Z_r[k]$ 、 $Y_r[k] = Z_i[k]$ 及 $X_i[k] = 0$ 、 $Y_i[k] = 0$ 。由此可見，若欲同時轉換兩個相同長度的實數序列，我們可用此方式有效率的達到。

法三：

最後一種計算 N 點實數序列傅立葉轉換的方式，為運用一個 $N/2$ 點的 CFFT 來有效地運算單一 N 點實數序列的離散傅立葉轉換。就如同下圖(4.3)所示，將長度為 N 的實數序列 $x[n]$ 分成偶數序列 $x[2n]$ 及奇數序列 $x[2n+1]$ 兩部分後，令一長度為 $N/2$ 的複數序列 $y[n]$ ， $y[n]=x[2n]+jx[2n+1]$ 其實部為 $x[2n]$ 、虛部為 $x[2n+1]$ 。經由 $N/2$ 點的 CFFT 運算，後端再藉由利用實數訊號轉換後的特性加以處理，即可得 $x[n]$ 的離散傅立葉轉換 $X[k]$ 。

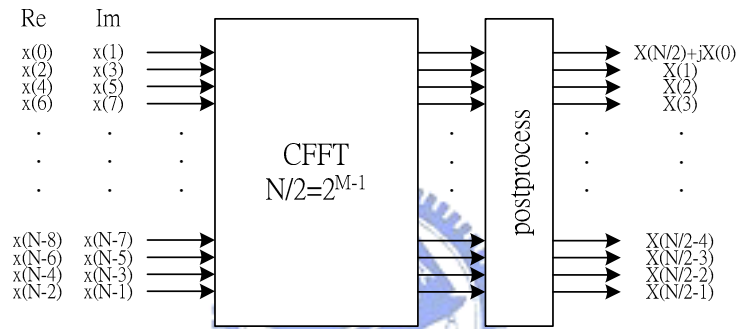


圖 4.3：單實數序列之 CFFT 及後端處理方塊圖

一個長度為 N 的實數序列 $x[n]$ ，將其偶數點和奇數點分開成長度為 $N/2$ 的兩實數序列 $h[n]$ 和 $g[n]$ 。

$$h[n] = x[2n], \quad 0 \leq n \leq N/2 - 1 \quad (4.6)$$

$$g[n] = x[2n+1], \quad 0 \leq n \leq N/2 - 1 \quad (4.7)$$

將其帶入離散傅立葉轉換：

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n] W_N^{nk} = \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{nk} \\ &= \sum_{n=0}^{N/2-1} h[n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} g[n] W_{N/2}^{nk} = H[k] + W_N^k G[k], \quad 0 \leq k \leq N-1 \end{aligned} \quad (4.8)$$

其中 $H[k]$ 與 $G[k]$ 分別為 $h[n]$ 和 $g[n]$ 的離散傅立葉轉換。

令 $y[n]$ 為一長度等於 $N/2$ 的複數序列，且由 $h[n]$ 與 $g[n]$ 兩實數序列所組成，其實部為 $h[n]$ 、虛部為 $g[n]$ 。

$$y[n] = h[n] + jg[n], \quad 0 \leq n \leq N/2 - 1 \quad (4.9)$$

下式為其離散傅立葉轉換：

$$Y[k] = H[k] + jG[k], \quad 0 \leq k \leq N/2 - 1 \quad (4.10)$$

且 $Y^*[N/2 - k] = H[k] - jG[k], \quad 0 \leq k \leq N/2 - 1 \quad (4.11)$

由上兩式，可以堆得 $H[k]$ 和 $G[k]$ ：

$$H[k] = (Y[k] + Y^*[N/2 - k])/2, \quad 0 \leq k \leq N/2 - 1 \quad (4.12)$$

$$jG[k] = (Y[k] - Y^*[N/2 - k])/2, \quad 0 \leq k \leq N/2 - 1 \quad (4.13)$$

最後，再藉由 $H[k]$ 和 $G[k]$ 來求得 $X[k]$ ：

$$X[k] = H[k] + W_N^k G[k], \quad 0 \leq k \leq N - 1 \quad (4.14)$$

因為 $x[n]$ 為實數序列，其轉換後的 $X[k]$ 有 Hermitian Symmetric 特性，我們只需得到在此 $0 \leq k \leq N/2$ 區間的 $X[k]$ 值即可。且當 $k = 0$ 與 $k = N/2$ 時， $X[k]$ 都只為實數 $X[0] = Y_r[0] + Y_i[0]$ 、 $X[N/2] = Y_r[0] - Y_i[0]$ 。由此可知，此方式可以有效的針對單一長度為 N 的實數序列做其傅立葉轉換。也就是說所運用到的 CFFT 只需 $N/2$ 點數，較原本的點數 N 少一半。

一個長度為 N 之 radix-2 DIT RFFT 的計算複雜度：

$$M(N) = (N/4 + 1) \log_2(N/2) \quad (4.15)$$

$$A(N) = (N/2 + 4) \log_2(N/2) \quad (4.16)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

4.2.2 HS-IFFT 演算法

就如前面所述，若欲轉換的序列 $x[n]$ 為實數序列，則其離散傅立葉轉換後之序列必具有 Hermitian Symmetric 的性質。故我們也可以針對其反向傅立葉轉換之序列若為 Hermitian Symmetric 序列，利用此 Hermitian Symmetric 的特性來反推出有效率的反向傅立葉轉換。

若一長度為 N 的離散序列 $X[k]$ 具有 Hermitian Symmetric 的性質，則：

$$X[N - k] = X^*[k], \quad 0 < k \leq N/2 - 1 \quad \text{且 } X[0] \text{ 與 } X[N/2] \text{ 為純實數}$$

將此 $X[k]$ 序列做其反向傅立葉轉換：

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-nk} \quad (4.17)$$

$$= \sum_{k=0}^{N/2-1} X[k] W_N^{-nk} + (-1)^n \sum_{k=0}^{N/2-1} X^*[N/2-k] W_N^{-nk}, \quad 0 \leq n \leq N-1$$

將上式分成奇偶兩序列：

$$x[2n] = \sum_{k=0}^{N/2-1} (X[k] + X^*[N/2-k]) W_{N/2}^{-nk} \quad (4.18)$$

$$x[2n+1] = \sum_{k=0}^{N/2-1} (X[k] - X^*[N/2-k]) W_N^{-k} W_{N/2}^{-nk} \quad (4.19)$$

由之前 RFFT 的假設，得知：

$$x[2n] + jx[2n+1] = y[n] = \sum_{k=0}^{N/2-1} Y[k] W_{N/2}^{-nk} \quad (4.20)$$

故可由 $X[k]$ 來求得 $Y[k]$ ，再將其經過 $N/2$ 點數的 Complex-valued IFFT(CIFFT) 後，即可求得 $X[k]$ 的反向離散傅立葉轉換 $x[n]$ ，如下圖(4.4)所示。

$$Y[k] = (X[k] + X^*[N/2-k]) + j(X[k] - X^*[N/2-k]) W_N^{-k} \quad (4.21)$$

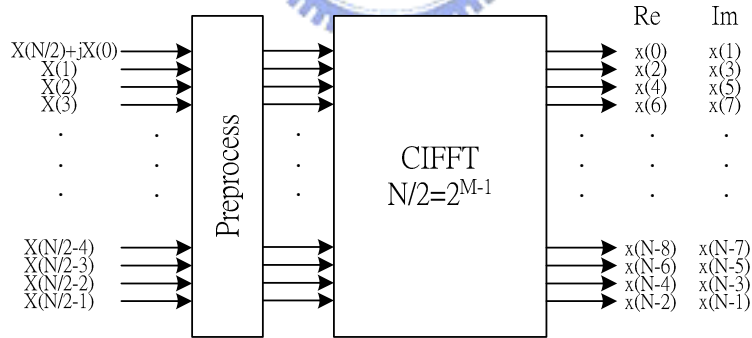


圖 4.4：Hermitian Symmetric 序列之前端處理及 CIFFT 方塊圖

一個長度為 N 之 radix-2 DIT HS-IFFT 的計算複雜度：

$$M(N) = (N/4 + 1) \log_2(N/2) \quad (4.22)$$

$$A(N) = (N/2 + 4) \log_2(N/2) \quad (4.23)$$

其中， $M(N)$ 和 $A(N)$ 分別代表複數乘法和複數加法的個數。

4.3 Memory-Based Radix-2 雙模可變長度 FFT/IFFT 處理器

架構

4.3.1 Radix-2 DIT FFT 處理器架構

採用 Radix-2 Decimation-in-Time(DIT) FFT 演算法來實現此架構，其基本運算單元為 radix-2 DIT butterfly 運算器。圖(4.5)為 16 點 radix-2 DIT FFT 運算流程圖，而圖(4.6)為其等效流程圖，其輸入序列的儲存順序為位元反向順序 (digit-reverse)，輸出序列則為正常順序(in-order)，butterfly 運算器的運算順序為從每階段的第一個 butterfly 開始計算(由上而下)，算完此階段的 butterfly 後接著算下一階段(從左至右)依此方式運算下去。且每個 butterfly 計算完後，會將結果採 in-place 方式存回原本所在的記憶體位址。整個 16-點 DIT FFT 流程圖共有 4 個階段與 4 個 shuffle permutation，其中 shuffle permutation 為將輸入序列重新排列的重排矩陣，且不同階段有不同的 shuffle permutation，之後將有詳細的介紹。

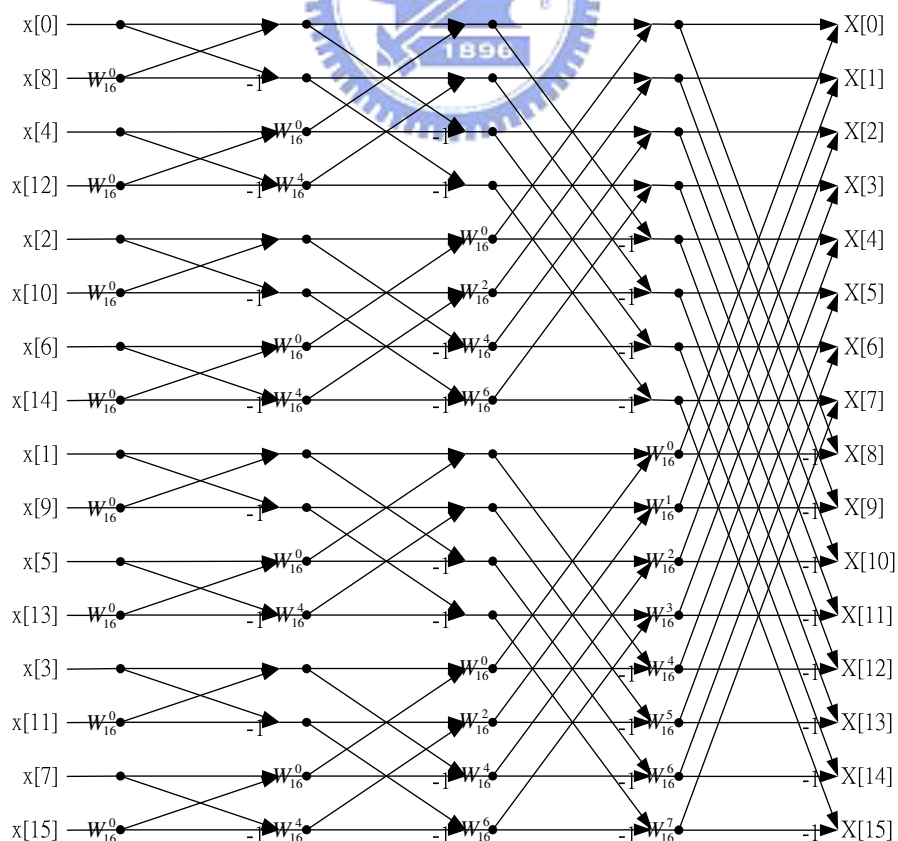


圖 4.5：16 點 radix-2 DIT FFT 運算流程圖

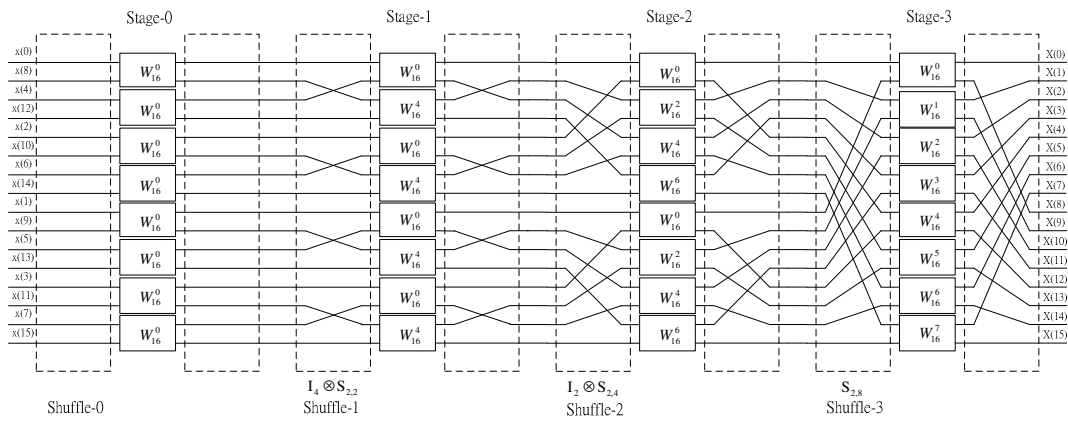


圖 4.6：16 點 radix-2 DIT FFT 運算等效流程圖

下圖(4.7)為 memory-based 架構的 radix-2 FFT 方塊圖。整個架構只採用一個 radix-2 butterfly 運算器作為運算單元，其次，FFT 的點數是可變動的，可以藉由控制訊號 M 來選擇做 N 點的分時快速傅立葉轉換，其中 $M = \log_2 N$ 。

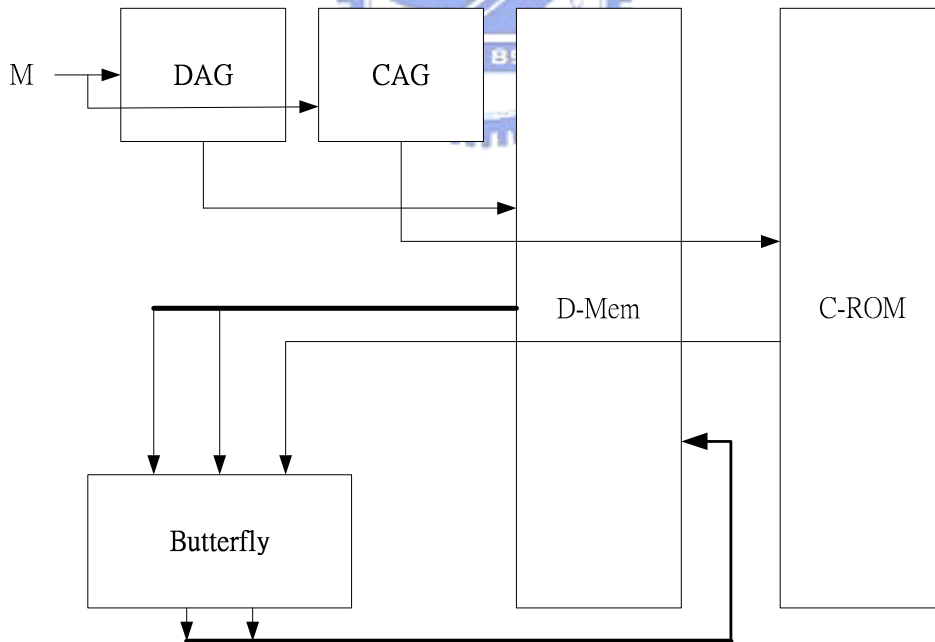


圖 4.7：Memory-based 架構的 radix-2 FFT 處理器方塊圖

Butterfly 運算器有三個輸入運算元，這三個運算元分別依照資料位址產生器(Data Address Generator, DAG)和係數位址產生器(Coefficient Address Generator, CAG)所產生出來之位址去記憶體裡抓取，經由運算後會產生兩個輸出結果，輸出結果儲存的位址為原本抓取的輸入資料記憶體位址。DAG 主要是根據每個階段之前的 shuffle permutation 運算，用來產生 butterfly 運算器的輸入資料所在記憶體位址，CAG 則是提供 butterfly 運算器的 twiddle factor 所在唯讀記憶體位址。D-Mem 與 C-ROM 分別為儲取輸入資料與 twiddle factors 的儲存元件。

4.3.2 Radix-2 DIT IFFT 處理器架構

至於 IFFT，一般常見的作法都是直接利用 FFT 的運算模式來完成，差別只是要另外將輸入及輸出取共軛運算。但若欲運算 RFFT(Real-valued FFT)及 HS-IFFT(Hermitian Symmetric IFFT)的話，此作法是不恰當的。因為 RFFT 及 HS-IFFT 是利用 half-size 之 Complex FFT/Complex IFFT(CFFT/CIFFT)並配合適當的 post-processing 及 pre-processing 來達到有效率之運算方式，然而此方式所需之 CFFT 的輸出得為 in-order 輸出，CIFFT 的輸入得為 in-order 輸入，故我們必須另外設計出一套具有 in-order 輸入的 CIFFT 架構。

圖(4.8)為 16 點 radix-2 DIT IFFT 運算流程圖，而圖(4.9)為其等效流程圖，其輸入序列的儲存順序為正常順序(in-order)，輸出序列則為位元反向順序(digit-reverse)，同樣地，butterfly 運算器運算順序為，從每階段的第一個 butterfly 開始計算(由上而下)，算完此階段的 butterfly 後接著算下一階段(從左至右)，依此方式運算下去。每個 butterfly 計算完後，一樣的會將結果採 in-place 方式存回原本所在的記憶體位址。整個 16-點分時快速傅立葉轉換流程圖共有 $4(\log_2 16)$ 個階段與 4 個 shuffle permutations，且不同的階段也具有不同的 shuffle permutation。

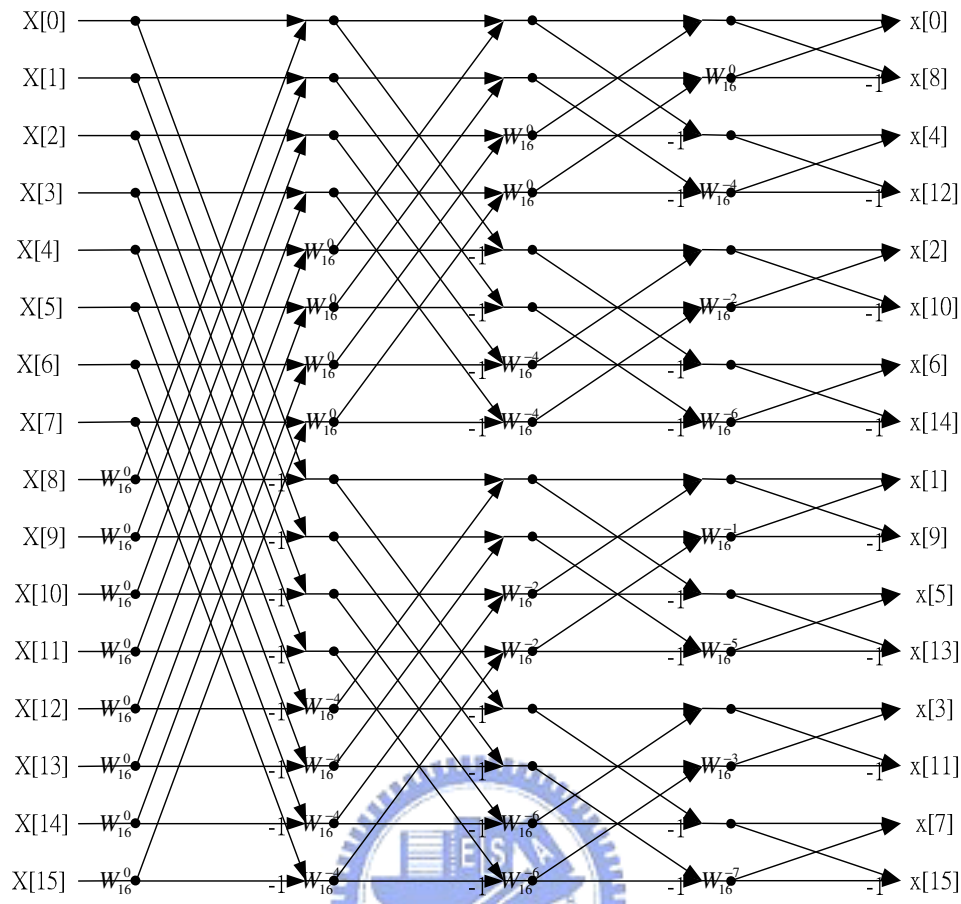


圖 4.8：16 點 radix-2 DIT IFFT 運算流程圖

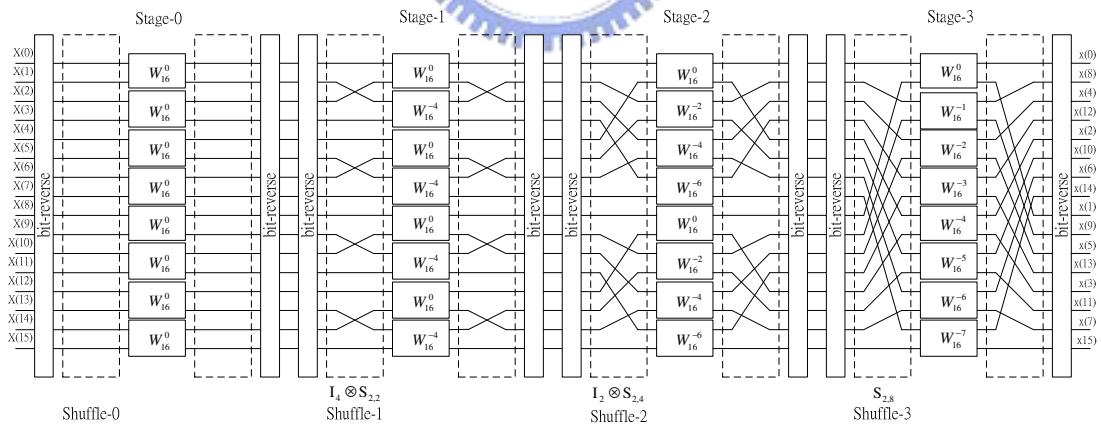


圖 4.9：16 點 radix-2 DIT IFFT 運算等效流程圖

下圖(4.10)為 memory-based 架構 radix-2 IFFT 方塊圖。整個架構也是採用一個 radix-2 butterfly 運算器作為運算單元，且 IFFT 的點數也相同地透過控制訊號 M 來決定其轉換點數的大小，其中 $M = \log_2 N$ 。

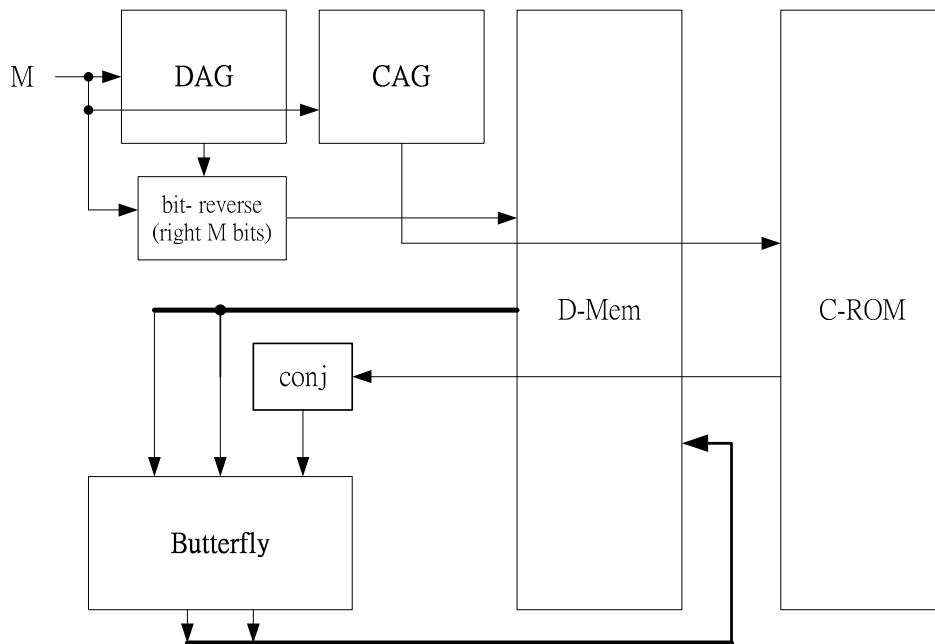


圖 4.10：Memory-based 架構 radix-2 IFFT 處理器方塊圖

與 FFT 架構相同地，butterfly 運算器有三個輸入運算元，這三個運算元分別依照經過 bit-reverse 的 DAG 和 CAG 所產生出來之位址去記憶體裡抓取，並把從唯讀記憶體擷取出來的 twiddle factor 取共軛後餵入 butterfly 運算器運算，再將運算後產生的兩個輸出結果存回原本抓取的輸入資料記憶體位址。IFFT 的點數也是可變動的，可以藉由控制訊號 M 來選擇做 N 點的 IFFT，其中 $M = \log_2 N$ 。

4.3.3 雙模 Radix-2 DIT FFT/IFFT 處理器架構

我們把圖(4.7)分時快速傅立葉轉換(FFT)流程圖與圖(4.10)反向分時快速傅立葉轉換(IFFT)流程圖做個比較之後，會發現 FFT 與 IFFT 在相同階段內所做的 shuffle permutation 都是一樣的。只是 IFFT 在產生輸入資料位址時，須比 FFT 多做一個 bit-reverse 的步驟。且可由相同的 CAG 來產生 butterfly 運算器所需的 twiddle factor 位址，IFFT 只需在將其所擷取出的 twiddle factor 取共軛即可。由於 FFT 與 IFFT 具有相同的 DAG、CAG、butterfly 運算器與記憶體需求，故可將其整合至同個平台中。圖(4.11)為雙模 memory-based 架構的 radix-2 FFT/IFFT 方塊圖。

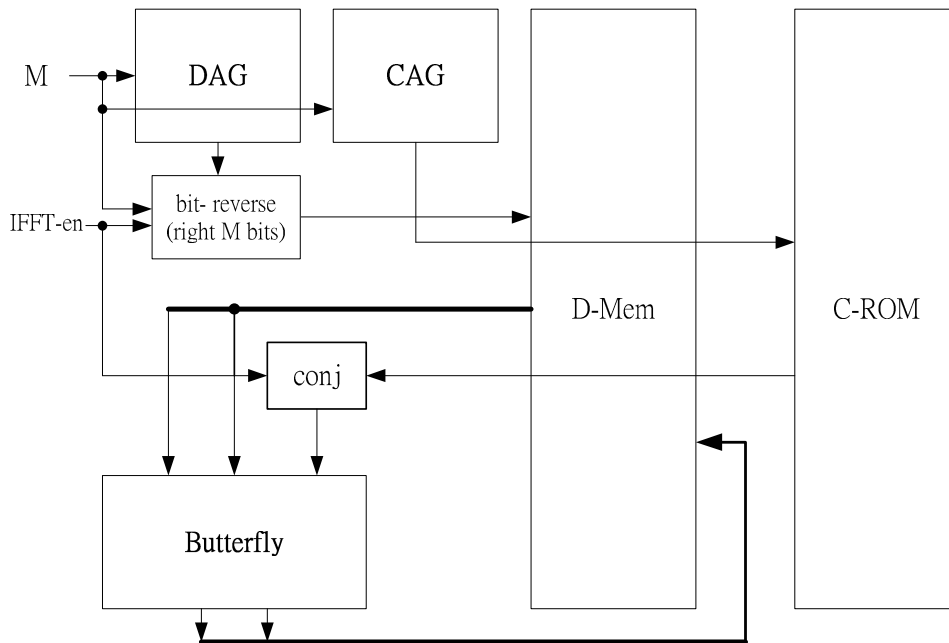


圖 4.11：雙模 memory-based 架構之 radix-2 FFT/IFFT 處理器方塊圖

此 memory-based 架構的雙模 radix-2 FFT/IFFT 處理器，藉由控制訊號 IFFT-en 來做 FFT 與 IFFT 模式的切換，當控制訊號 IFFT-en 為 high 時，bit-reverse 與 conj(共軛)運作，做 IFFT 運算；反之，IFFT-en 為 low 時，bit-reverse 與 conj(共軛)停止運作，做 FFT 運算。當運算模式為 FFT 模式時，輸入序列是採位元反向順序 (digit-reverse) 的儲存方式存在記憶體中，運算完畢後的輸出，則是屬於正常順序 (in-order) 地被存放在記憶體中；若為 IFFT 模式時，輸入序列採正常順序 (in-order) 存放在記憶體中，其運算完畢後的輸出，則被位元反向順序 (digit-reverse) 地存放在記憶體中。且可以藉由控制訊號 M 來選擇做 N 點的 FFT/IFFT，其中 $M = \log_2 N$ 。

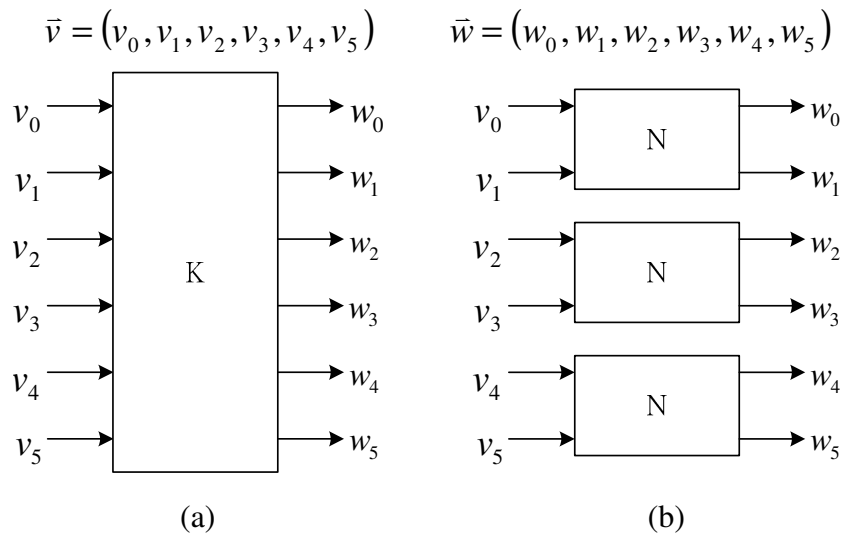


圖 4.12：(a)大矩陣運算(b)分解成三個小矩陣運算

而 S_{B_0, B_1} 為重排矩陣，也就是說一個含有 $B_0 \times B_1$ 個元素之向量 \bar{v} ，經過此矩陣運算後，其內部元素將會互相調動重新排列成為一個相樣含有 $B_0 \times B_1$ 個元素的新向量 \bar{v}' 。此重排矩陣 S_{B_0, B_1} 可用另一種較簡單的方式來描述其重排方式，假設一個具有 15 個元素的向量 \bar{v} ，其向量元素所在位置分別為 $i = 0$ 至 $i = 14$ ，經重排矩陣 $S_{3,5}$ 運算後向量 \bar{v} 內的元素將重新排列成為向量 \bar{v}' 。如圖(4.13)所示，原本在向量 \bar{v} 中 $i = 2$ 位置的元素經由 $S_{3,5}$ 運算後，將移至新向量 \bar{v}' 中的 $j = 6$ 位置。我們可將元素所在位置 i ，根據重排矩陣 $S_{3,5}$ 的下標 [3,5] 作為其 i 的基底來表示，經由 $S_{3,5}$ 轉換可得重排後由基底 [5,3] 所表示的位置 j ，如下式所示：

$$i : [a, b] \xrightarrow{S_{3,5}} j : [b, a] \quad (4.27)$$

藉由此式子，我們可以知道經過重排後位置之間的相對關係。

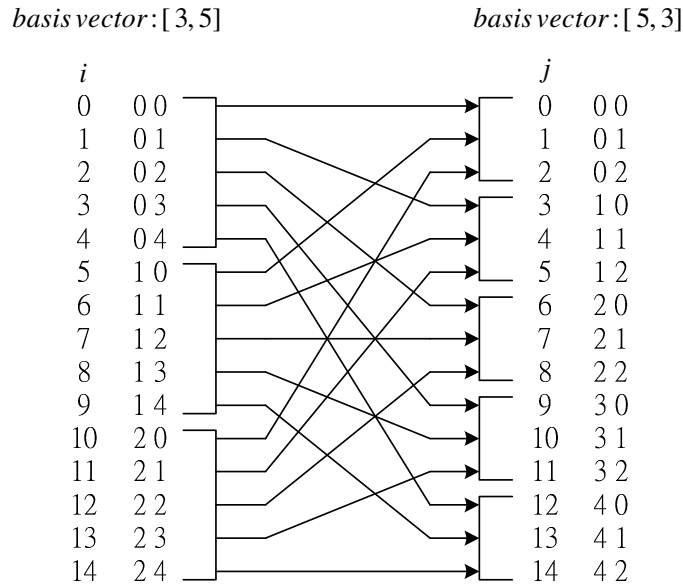


圖 4.13 : $i:[a, b] \xrightarrow{S_{3,5}} j:[b, a]$ 圖示

最後，我們可將之前所提到的 Kronecker Product 與 shuffle permutation 結合在一起運用。若有一重排矩陣為 $I_{B_0} \otimes S_{B_1, B_2}$ 形式，則重排後位置的對應關係如式子 (4.28) 所示，其中 i 為轉換前之位置且基底為 $[B_0, B_1, B_2]$ ， j 為轉換之後的位置，其基底為 $[B_0, B_2, B_1]$ 。

$$i:[a, b, c] \xrightarrow{I_{B_0} \otimes S_{B_1, B_2}} j:[a, c, b] \quad (4.28)$$

由之前的流程圖可知，一個 $N = 2^M$ 點的 radix-2 分時快速傅立葉轉換共有 M 個 stages，且每個 stage 對應到不同形式的 shuffle permutation，而第 i 個 stage 的 shuffle permutation 為：

$$I_{2^{M-1-i}} \otimes S_{2^i, 2} \quad i = 0, 1, \dots, M-1 \quad (4.29)$$

故我們可由上式，來得知 butterfly 運算器所欲擷取資料的記憶體對應位址。藉由將計數器所計數出來的值 k ($k = 0, 1, \dots, N-1$)，經過第 i 個 stage 的 shuffle permutation ($I_{2^{M-1-i}} \otimes S_{2^i, 2}$) 運算後，如下式 (4.30) 所示，即可產生 butterfly 運算器的輸入資料擷取位址。圖 (4.14) 為 DAG 之方塊圖。

$$k:[a, b, c] \xrightarrow{shuffle} address[a, c, b] \quad (4.30)$$

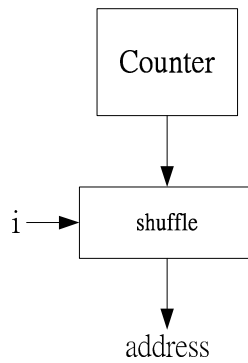


圖 4.14：資料位址產生器方塊圖

● 係數位址產生器(CAG)

一個 $N = 2^M$ 點的 radix-2 DIT FFT/IFFT 運算共有 M 個 Stages，且每個 Stage 之 twiddle factor 位址的抓法都是不同的。我們將以一個 16 點 radix-2 DIT FFT/IFFT 為例子，來簡單地說明 CAG 的作法。圖(4.15)為 16 點 FFT 之 CAG 推導範例。

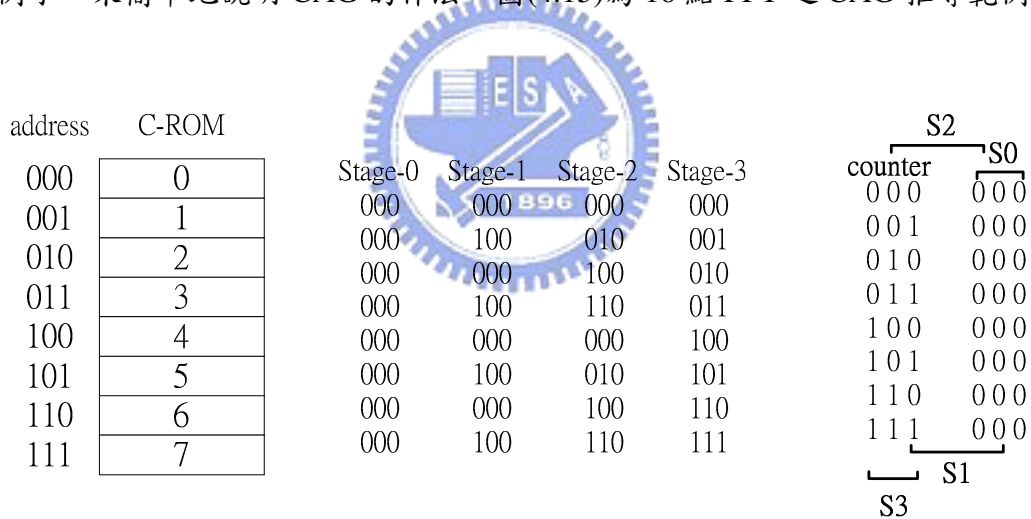


圖 4.15：16 點 radix-2 DIT FFT/IFFT 之 CAG 示意圖

C-ROM 裡存的為 twiddle factor 值，0 表示 W_N^0 、1 表示 W_N^1 依此類推。由上所示的 16 點 FFT/IFFT 中 Stage-0 所要抓取的位址可由三位元計數器所產生之值左移 3 位元後的 S0 得之，Stage-1 所抓取的位址可由計數器所產生之值左移 2 位元後的 S1 得之，同理 Stage-2 則需由左移 1 位元後的 S2 得之，最後的 Stage-3 則是左移 0 位元後的 S3。經由上面的分析，我們可以很簡單的找到其規律性。一個

$N = 2^M$ 點的 FFT 在第 i 個 Stage 只需把計數器產生之值左移 $M-i-1$ 個位元就可得到所需的 twiddle factor 位址，如圖(4.16)所示。

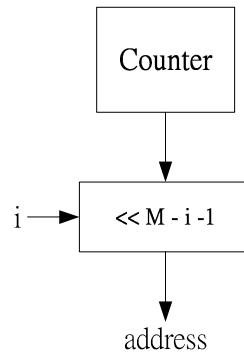


圖 4.16：係數位址產生器方塊圖

- **Reduce Coefficient Memory size**

一般來說，一個長度為 N 的 radix-2 分時快速傅立葉轉換需要儲存的 twiddle factor 個數為 $N/2$ 。但事實上，我們可以利用其 twiddle factors 相互間的映射關係來減少所需儲存的數目，藉由此相互間的相關性可將 twiddle factor 儲存個數減至為 $N/8+1$ [18]。此方式可以有效地減少 FFT 處理器的硬體及功率耗費。以下圖(4.17)32 點分時快速傅立葉轉換所需之 twiddle factors 圖為例。我們可藉由第四象限的 twiddle factors 映射得到分佈在第三象限上的 twiddle factors，且可再進一步地將第四象限分成兩份 block I 和 block II，同樣地它們相互間也都存在著映射的關係。故我們最後只需儲存 block I 內的 twiddle factors，再利用 block 彼此間的關係即可得到 FFT 運算所需的所有 twiddle factors。我們由表(4.1)可知，block II 中的值可由 block I 中藉由實虛變號且互換得到，block III 可由 block I 實虛互換後將其虛部變號得之，block IV 只需將 block I 中的實部變號即可求得。

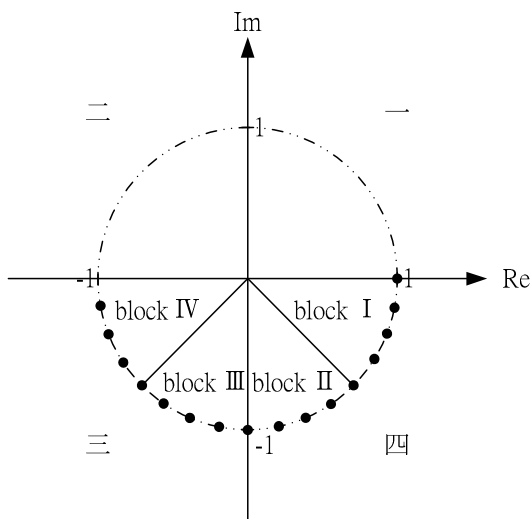


圖 4.17：32 點分時快速傅立葉轉換所需之 twiddle factors 圖示

address (b_3, b_2, b_1, b_0)	twiddle factor	quadrant	block
0000	1.0, 0.0	四	block I (00)
0001	.98, -.19		
0010	.92, -.38		
0011	.83, -.55		
0100	.71, -.71		
0101	.55, -.83		
0110	.38, -.92		
0111	.19, -.98		
1000	0.0, -1.0	三	block III (10)
1001	-.19, -.98		
1010	-.38, -.92		
1011	-.55, -.83		
1100	-.71, -.71		
1101	-.83, -.55		
1110	-.92, -.38		
1111	-.98, -.19		
			block IV (11)

表 4.1：32 點分時快速傅立葉轉換所需之 twiddle factors 關係表

就如前面所述，對於一個長度為 N 的 radix-2 分時快速傅立葉轉換所需的 twiddle factors 可由 block I 得之，所以我們得知其 twiddle factor 所屬 block 與其對應至 block I 位址的關係式。twiddle factor 所屬的 block 可由最原始的 twiddle factor 儲存位址所決定，如式(4.31)所示。

$$block = (b_{MSB}, b_{MSB-1} \& (b_{MSB-2} | b_{MSB-3} | \dots | b_1 | b_0)) \quad (4.31)$$

決定了所屬的 block 之後，接下去要尋找在 block I 中所對應的位址(因為記憶體現在只存 block I 中的值，所以其所對應的位址要忽略 b_{MSB})，所對應的位址如下所示。

$$block\ I、block\ III : address = (b_{MSB-1}, b_{MSB-2}, \dots, b_2, b_1, b_0) \quad (4.32)$$

$$block\ II、block\ IV : address = \sim (b_{MSB-1}, b_{MSB-2}, \dots, b_2, b_1, b_0) + 1 \quad (4.33)$$

最後，再根據此所屬的 block 及對應位址求得 block I 所映射的 twiddle factor。依此方式，我們只需儲存 $N/8+1$ 個 twiddle factor 即可。

4.3.5 Conflict Free Memory Addressing

一般來說，一個長度為 N 的 FFT 轉換需要經過 $\frac{N}{r} \times \log_r N$ 個 radix- r butterfly 運算，且每個 radix- r butterfly 運算必須經由 $2r$ 次的記憶體讀寫來完成。所以對於一個要運算長度為 N 的 memory based FFT/IFFT 處理器來說，總共需要的記憶體讀寫次數為 $2N \times \log_r N$ 次。然而，就一個有效率的 memory based FFT/IFFT 處理器來說，其記憶體讀寫所需總 clock cycle 數越少越能達到其運算效率(運算時所需的 clock cycle 數越少越好)。有兩種方式可以減少其記憶體讀寫所需的總 clock cycle 數，一為使用 high-radix 的架構，此方式相對的會增加其硬體複雜度；第二種方式為藉由增加記憶體擷取頻寬來減少總 clock cycle 數。增加記憶體擷取頻寬的方式有兩種，一為使用可以同時存取多筆資料的單一 multiple-port 記憶體來達到，但此方式為讓硬體面積增大；另一種方式，則為將記憶體分割為多組 (r) single-port 記憶體來達到同時讀取 r 筆資料和同時寫回 r 筆資料的目的。為了使 memory based FFT/IFFT 處理器的硬體面積減少，使用多組 single-port 記憶體模式是最好的選擇，本節將探討此有效率增加記憶體擷取頻寬的方式。

若 butterfly 處理器為 radix- r 的 butterfly 處理器，如圖(4.18)所示，則記憶體需被分為 r 塊，使得 radix- r 的 butterfly 處理器可以同時讀 (寫) r 個值。為了避免 butterfly 處理器在同一記憶體區塊內同時讀 (寫) 兩筆資料以上的情況發生，我們必須對 FFT 的輸入序列做適當的記憶體區塊配置，使得每次 butterfly 的運算元都個別來自不同的記憶體區塊[19]。

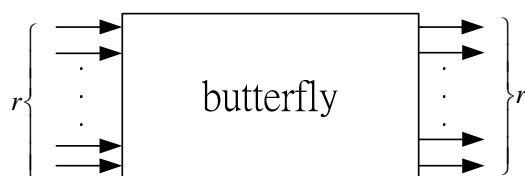


圖 4.18：Radix- r FFT butterfly 方塊圖

我們可根據下面所提到的方式來將輸入序列配置到不同的記憶體區塊內。

一個在長度為 N 之輸入序列中的資料，其原本儲存的記憶體位址為式(4.34)，欲將其配置至 r (radix- r FFT) 個記憶體區塊之一中，可由式(4.35)來決定其所屬之記憶體區塊和式(4.36)來決定其記憶體區塊的儲存位址。依此配置方式，可使 radix- r butterfly 運算器能在同一時間擷取到 r 筆資料來運算，藉以達到減少運算所需的 clock cycle 數。

$$data_original_address = (d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_r \quad (4.34)$$

$$memory_bank = (d_{n-1} + d_{n-2} + \dots + d_2 + d_1 + d_0) \bmod r \quad (4.35)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_r \quad (4.36)$$

$$n = \log_r N$$

下圖(4.19)為一個經過記憶體區塊配置的 16 點 radix-2 FFT 之例子。輸入序列採位元反向順序儲存，輸出序列採正常順序。

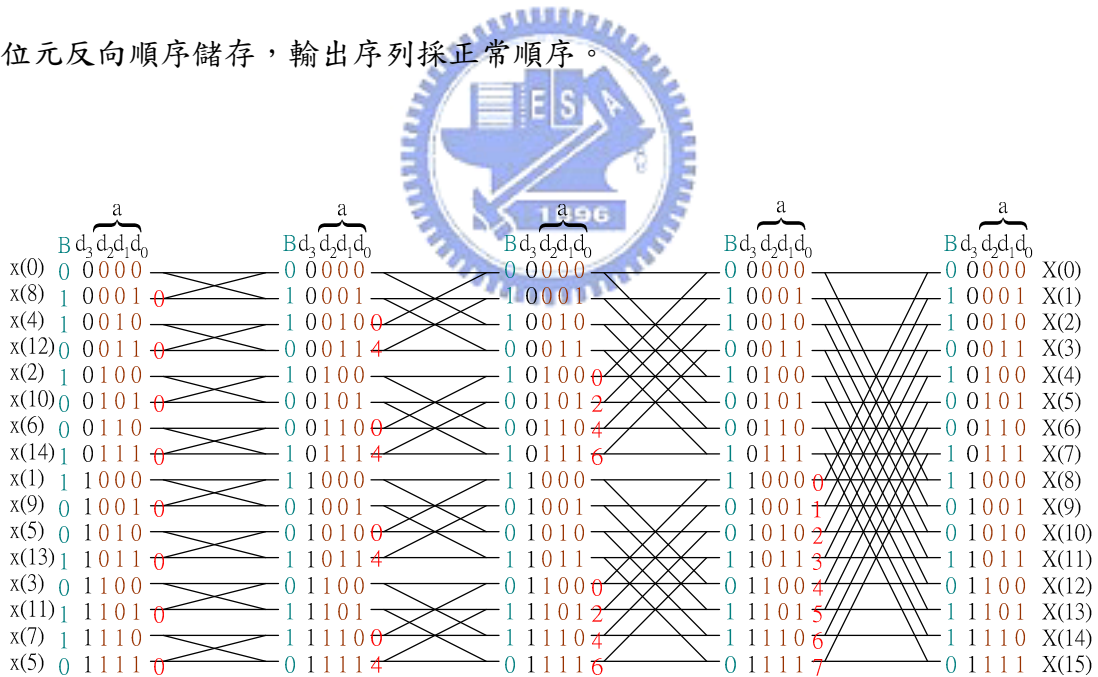


圖 4.19：16 點 radix-2 FFT 之區塊配置示意圖

根據上述，我們想把原本儲存在一個記憶體內的輸入序列，重新分配至兩個記憶體區塊，記憶體區塊分別為 Bank0 與 Bank1。根據下面兩個式子，可清楚的知道輸入序列要配置到哪個區塊和區塊中的哪個位址。

$$memory_bank = (d_{n-1} + d_{n-2} + \dots + d_2 + d_1 + d_0) \bmod 2 \quad (4.37)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_2 \quad (4.38)$$

B 代表分配到的區塊，0 為 Bank0、1 為 Bank1，a 代表區塊位址。

4.3.6 Block Floating Point Memory-Based 架構

就如同之前 pipeline 架構所述一樣，因為 fixed point 的字元長度有限，所以 fixed point FFT 處理器最重要的問題在於其計算後的精確度問題。同樣地，我們也可以使用 block scaling 的方式[12]來提高其 memory-based 架構運算後的訊號雜訊比。因為 memory-based 架構的計算方式是每階段每階段循序計算下去的方式，故可用 Block Floating Point (BFP)的方式來處理，此方法是先將每個階段所運算出的最大輸出值偵測出來，找到每階段的 scaling factor 後，再將所有下一階段的輸入值經過相同地 scaled 來當作 butterfly 處理器的輸入，以此方式來改善精確度。圖(4.20)為一個具有 BFP memory-based 架構的雙模 radix-2 FFT/IFFT 方塊圖。

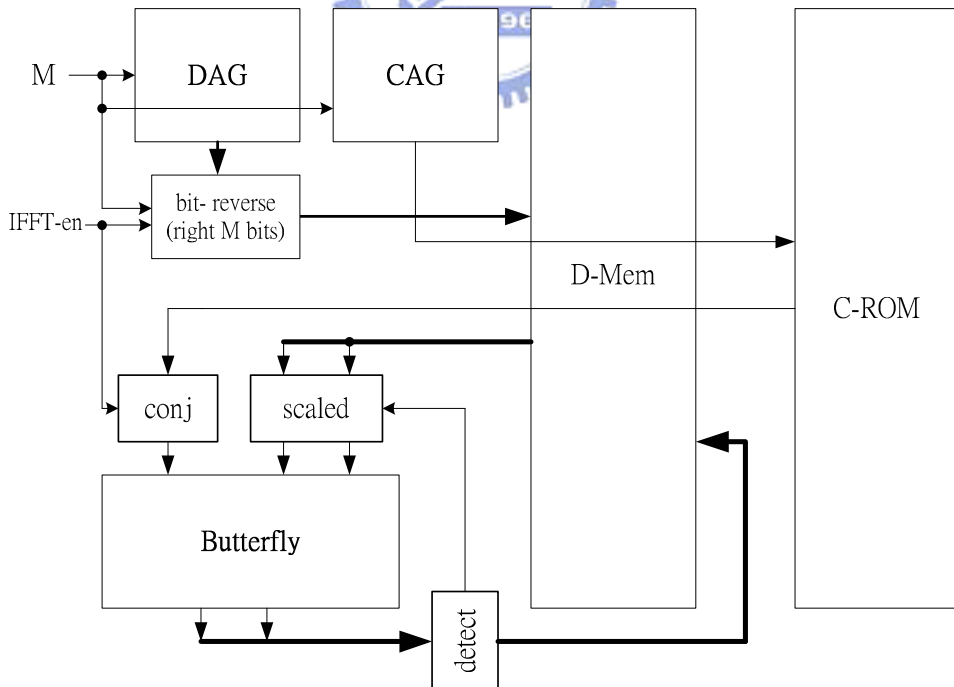


圖 4.20：BFP memory-based 架構的雙模 radix-2 FFT/IFFT 處理器方塊圖

Block scaling 的優點在於，可在不增加輸出字元長度的狀況下，讓輸出訊號雜訊比獲得改善。即使當欲轉換之輸入訊號值過小時，block scaling 也會在第一個階段就將其輸入訊號 scaled 至適當的準位，以增加其運算精確度。

4.4 Memory-Based Radix-4 雙模可變長度 FFT/IFFT 架構

一個要運算長度為 N 的 memory based FFT/IFFT 處理器來說，總共需要的記憶體讀寫次數為 $2N \times \log_2 N$ 次。我們可以使用 high-radix 的架構來減少記憶體讀寫次數，以增加運算效率(減少運算時所需總 clock cycle 數)。圖(4.21)為 64 點 radix-4 DIT FFT 運算的等效流程圖，其輸入序列的儲存順序為位元反向順序(digit-reverse)，輸出序列則為正常順序(in-order)。圖(4.22)為 64 點 radix-4 DIT IFFT 運算的等效流程圖，其輸入序列的儲存順序為正常順序(in-order)，輸出序列則為位元反向順序(digit-reverse)。同樣地，radix-4 butterfly 處理器的運算順序為，從每階段的第一個 butterfly 開始計算(由上而下)，算完此階段的 butterfly 後接著算下一階段(從左至右)，依此方式運算下去。且每個 butterfly 計算完後，會將結果採 in-place 方式存回原本所在的記憶體位址。每個階段也對應到不同形式的 shuffle permutation，如式(4.39)所示，藉由此 shuffle permutation 的重排矩陣運算可使輸入序列重新排列成每個階段中 butterfly 處理器所欲處理的序列順序。

第 i 個階段對應的 shuffle permutation 為：
$$I_{4^{M-i}} \otimes S_{4^i, 4} \quad i = 0, 1, \dots, M-1 \quad (4.39)$$

由圖(4.21)與圖(4.22)，我們知道 FFT 與 IFFT 在相同階段內所做的 shuffle permutation 都是一樣的。只是 IFFT 在產生輸入資料位址時，須比 FFT 多做一個 digit-reverse 的步驟。且可由相同的 CAG 來產生 butterfly 運算器所需的 twiddle factor 位址，IFFT 只需在將其所擷取出的 twiddle factor 取共軛即可。由於 FFT 與 IFFT 具有相同的 DAG、CAG、butterfly 運算器與記憶體需求，故可將其整合至同個平台中。同樣地，可用 Block Floating Point (BFP)的 Block scaling 方式來改善輸出訊號雜訊比。圖(4.23)為具有 BFP memory-based 架構的雙模 radix-4 FFT/IFFT 處理器方塊圖。

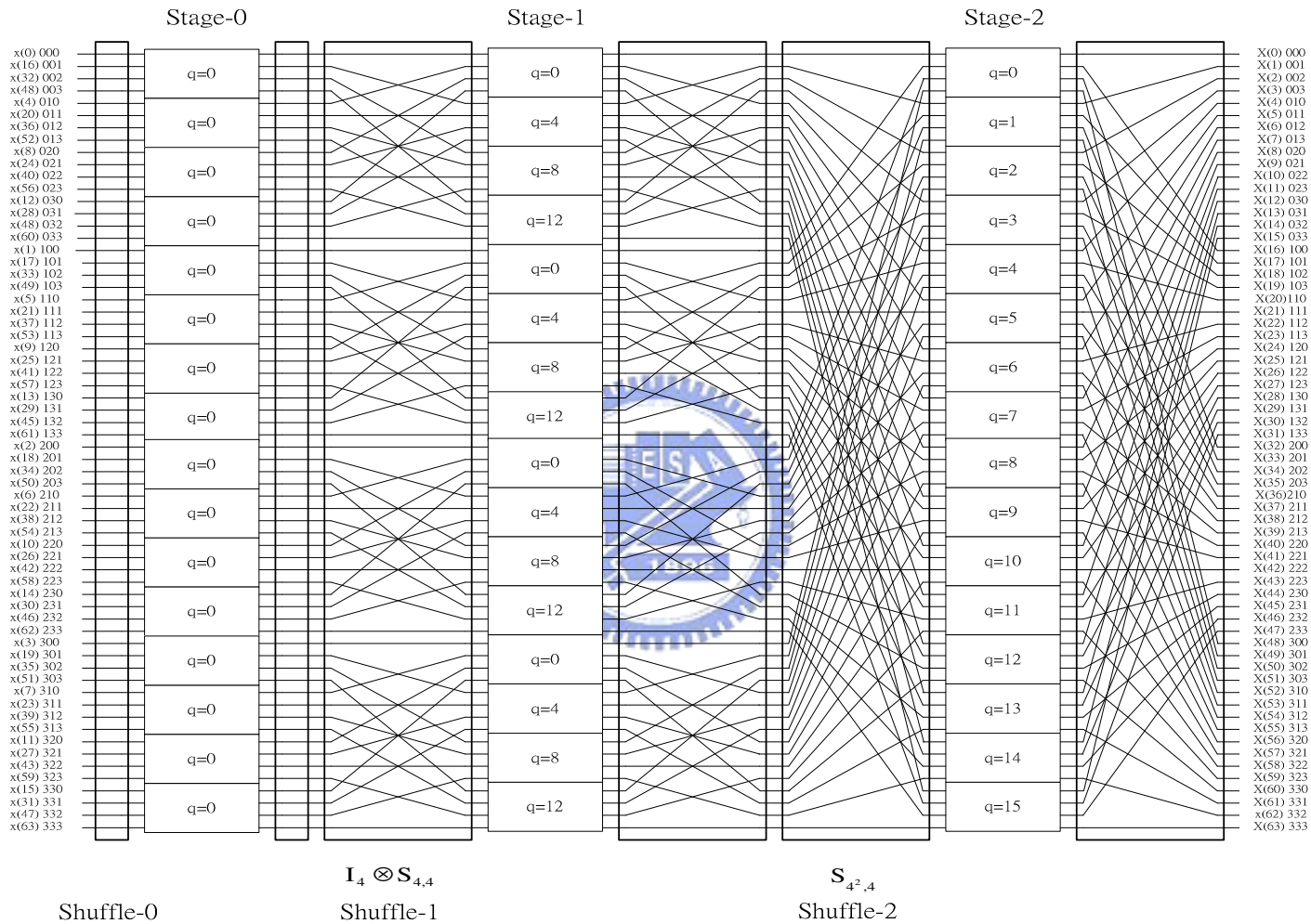


圖 4.21 : 64 點 radix-4 DIT FFT 運算等效流程圖

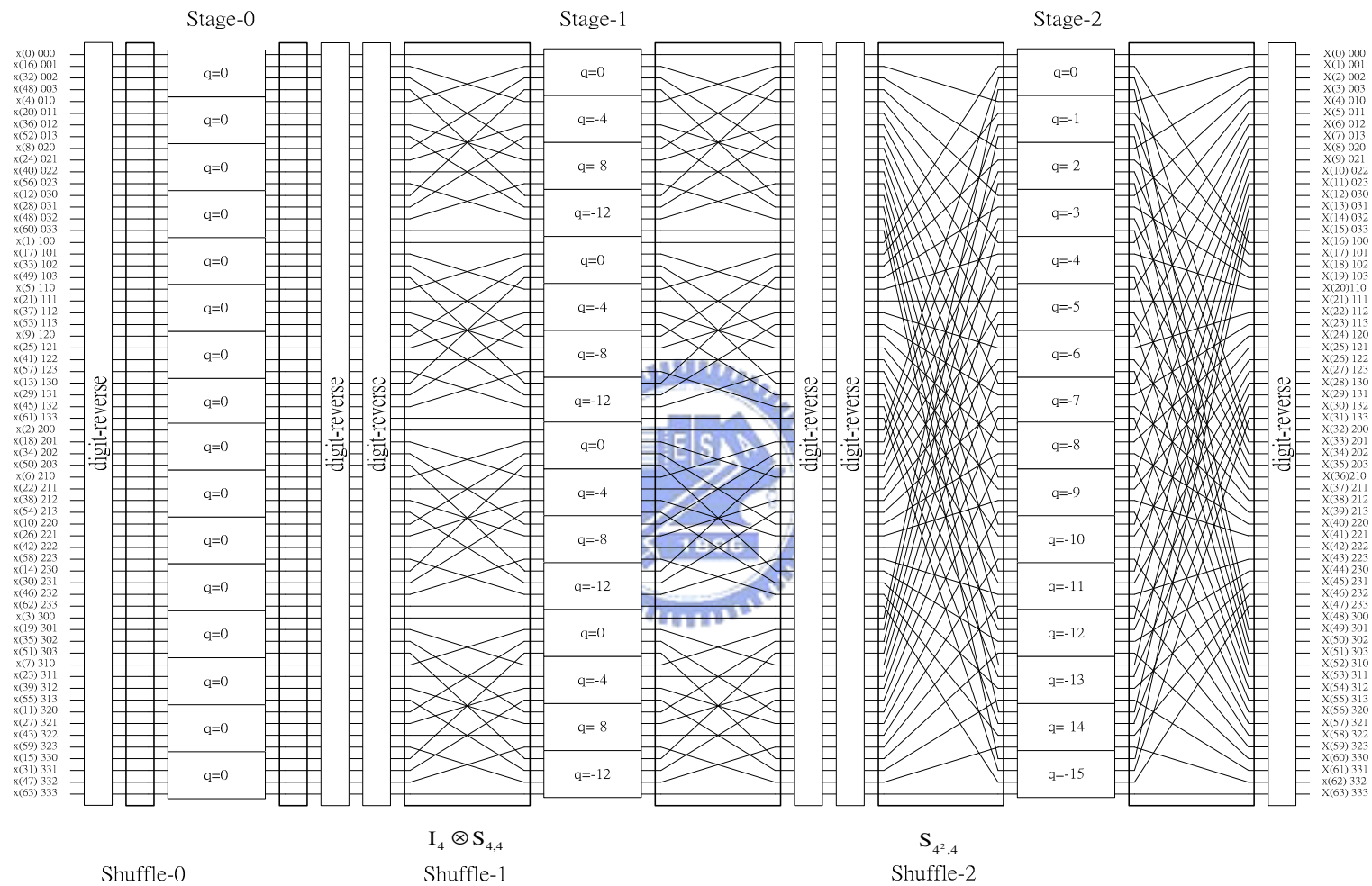


圖 4.22 : 64 點 radix-4 DIT IFFT 運算等效流程圖

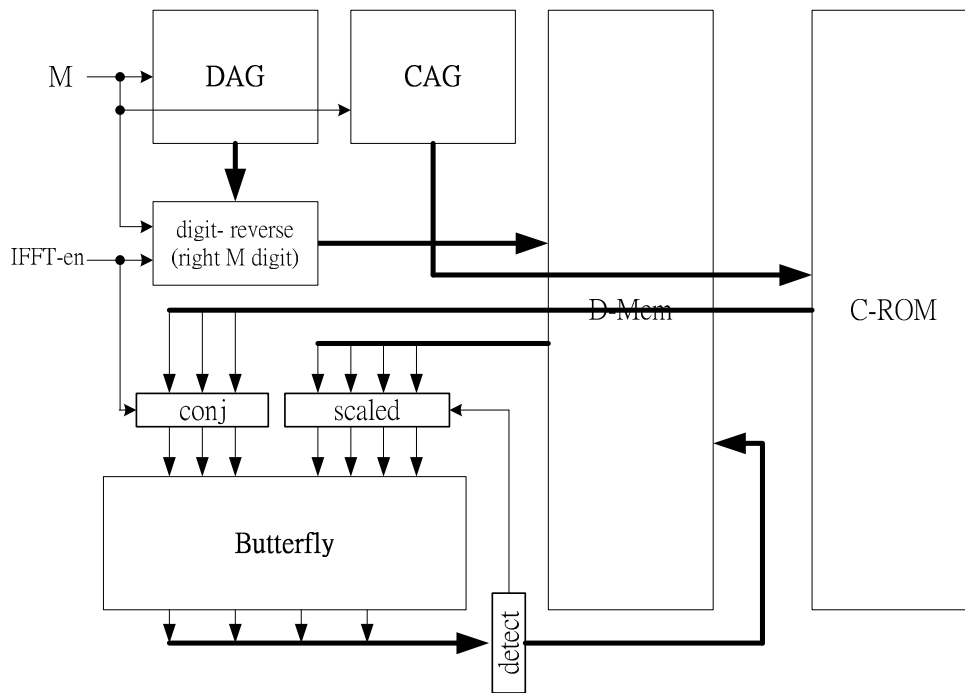


圖 4.23：BFP memory-based 架構的雙模 radix-4 FFT/IFFT 處理器方塊圖

此 BFP memory-based 架構的雙模 radix-4 FFT/IFFT 處理器，藉由控制訊號 IFFT-en 來做 FFT 與 IFFT 模式的切換，當控制訊號 IFFT-en 為 high 時，digit-reverse 與 conj(共軛)運作，做 IFFT 運算；反之，IFFT-en 為 low 時，digit-reverse 與 conj(共軛)停止運作，做 FFT 運算。當運算模式為 FFT 模式時，輸入序列是採位元反向順序(digit-reverse)的儲存方式存在記憶體中，運算完畢後的輸出，則是屬於正常順序(in-order)地被存放在記憶體中；若為 IFFT 模式時，輸入序列採正常順序(in-order)存放在記憶體中，其運算完畢後的輸出，則被位元反向順序(digit-reverse)地存放在記憶體中。控制訊號 M 是用來決定處理器運算的點數 N ，其中 $M = \log_4 N$ 。且藉由偵測出每個階段的 scaling factor 後，再將所有下一階段的輸入值經過相同地 scaled 來當作 butterfly 處理器的輸入。

同點數的傅立葉轉換下，若欲在較少的 clock cycle 數下運算完畢，則可選擇 high-radix 的方式來達到，但相對地會增加其硬體的複雜度。

4.5 Memory-Based 多模可變長度 RFFT/HS-IFFT 架構

Real-valued FFT(RFFT)與 Hermitian Symmetric IFFT(HS-IFFT)分別針對輸入訊號為實數序列與 Hermitian Symmetric 序列提供了更有效率地運算方式[13][14][15]。以一個 N 點的 FFT(IFFT)來說，若輸入序列為實數序列(Hermitian Symmetric

序列)，則只需利用到點數為 $N/2$ 點的 Complex FFT(Complex IFFT)及一些後端(前端)處理動作即可得到與 N 點 CFFT(CIFFT)相同的結果。這將大大地減少了處理器所需運算的 clock cycle 數，將使原本從記憶體擷取資料了次數從 $2N \times \log_r N$ 次減少至 $N \times \log_r (N/2) + 2N$ 次(前端或後端處理都為兩個階段的處理方式)，加快了硬體的處理時間。此外，因為 RFFT 與 HS-IFFT 架構分別需要一個輸入序列位元反向順序(digit-reverse)、輸出序列正常順序(in-order)的 CFFT 和輸入序列正常順序(in-order)、輸出序列位元反向順序(digit-reverse)的 CIFFT 來處理其運算，故我們可以利用整合於同一平台的 memory based CFFT/CIFFT 處理器並在其前後端分別配置一個專為 HSIFFT 與 RFFT 運算的前端處理器(preprocessor)及後端處理器(post-processor)，如圖(4.24)所示，以達到將 RFFT/HS-IFFT 整合至同一平台且減少硬體面積的目的。

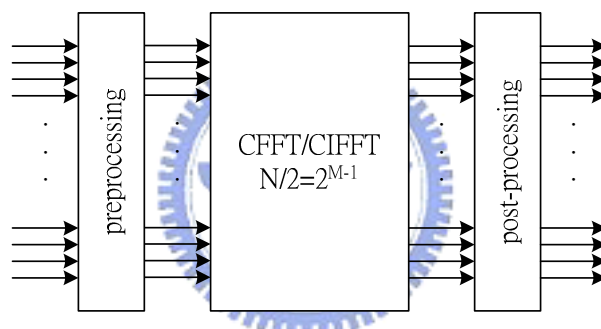


圖 4.24：RFFT/HS-IFFT 方塊圖

RFFT：

在 RFFT 演算法中，我們將長度為 N 的實數序列 $x[n]$ 分成奇偶兩序列 $g[n]$ 與 $h[n]$ ，使其組合成長度為 $N/2$ 的複數序列 $y[n] = x[2n] + jx[2n+1] = h[n] + jg[n]$ ，之後輸入 $N/2$ 點的 CFFT (Complex FFT)運算得到 $Y[k]$ ，如圖(4.25)所示，再藉由式(4.40)(4.41)(4.42)(4.43)(4.44)(4.45)(4.46)來還原 $X[k]$ 。

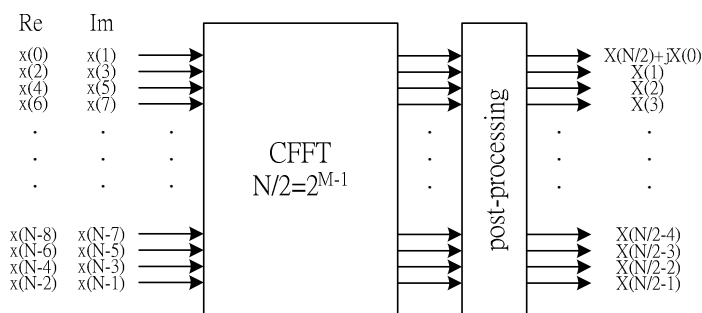


圖 4.25：RFFT 方塊圖

$$H[k] = (Y[k] + Y^*[N/2 - k])/2, \quad 0 \leq k \leq N/2 - 1 \quad (4.40)$$

$$jG[k] = (Y[k] - Y^*[N/2 - k])/2, \quad 0 \leq k \leq N/2 - 1 \quad (4.41)$$

$$X[k] = H[k] + W_N^{k+N/4} jG[k], \quad 0 \leq k \leq N/4 - 1 \quad (4.42)$$

$$X^*[N/2 - k] = H[k] - W_N^{k+N/4} jG[k], \quad 0 \leq k \leq N/4 - 1 \quad (4.43)$$

$$X[0] = \text{Re}\{Y[0]\} + \text{Im}\{Y[0]\} \quad (4.44) \quad X[N/2] = \text{Re}\{Y[0]\} - \text{Im}\{Y[0]\} \quad (4.45)$$

$$X[N/4] = Y^*[N/4] \quad (4.46)$$

圖(4.26)為其後端處理器的流程圖，共分為兩階段，第一階段在於產生 $H[k]$ 與 $G[k]$ ，第二階段則是藉由 $H[k]$ 與 $G[k]$ 得到最後的 $X[k]$ ，且我們將做個變數變換，把原本實數序列的長度 N 用 N_R 來取代，以避免跟之後所探討的複數序列長度混淆。

HS-IFFT :

在 HS-IFFT 演算法中，我們將長度為 N 的 Hermitian 對稱序列 $X[k]$ 取其 $0 \leq k \leq N/2$ 點數，藉由式(4.47)(4.48)(4.49)(4.50)得到 $Y[k]$ ，之後同樣地輸入 $N/2$ 點的 CIFFT (Complex IFFT) 得到 $y[n]$ ，如圖 (4.27) 所示，其中 $y[n] = x[2n] + jx[2n+1]$ ，其實部序列與虛部序列即為 $x[n]$ 的偶數跟奇數序列。

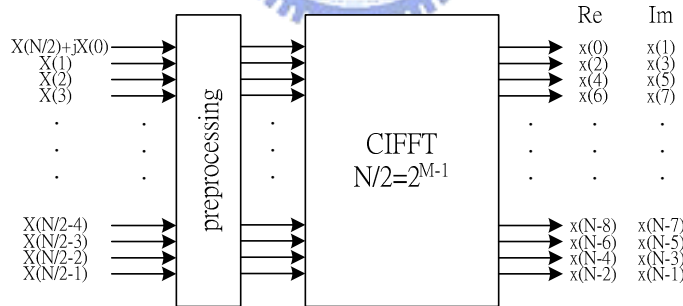


圖 4.27 : HS-IFFT 方塊圖

$$Y[k] = (X[k] + X^*[N/2 - k]) + (X[k] - X^*[N/2 - k]) W_N^{-(k+N/4)} \quad (4.47)$$

$$Y^*[N/2 - k] = (X[k] + X^*[N/2 - k]) - (X[k] - X^*[N/2 - k]) W_N^{-(k+N/4)} \quad (4.48)$$

$$Y[0] = [(1 - j)(X[0] + jX[N/2])] \quad (4.49)$$

$$Y[N/4] = 2X^*[N/4] \quad (4.50)$$

圖(4.28)為其前端處理器的流程圖，共分為兩階段。在此同樣地，把原本實數序列的長度 N 用 N_R 來取代。

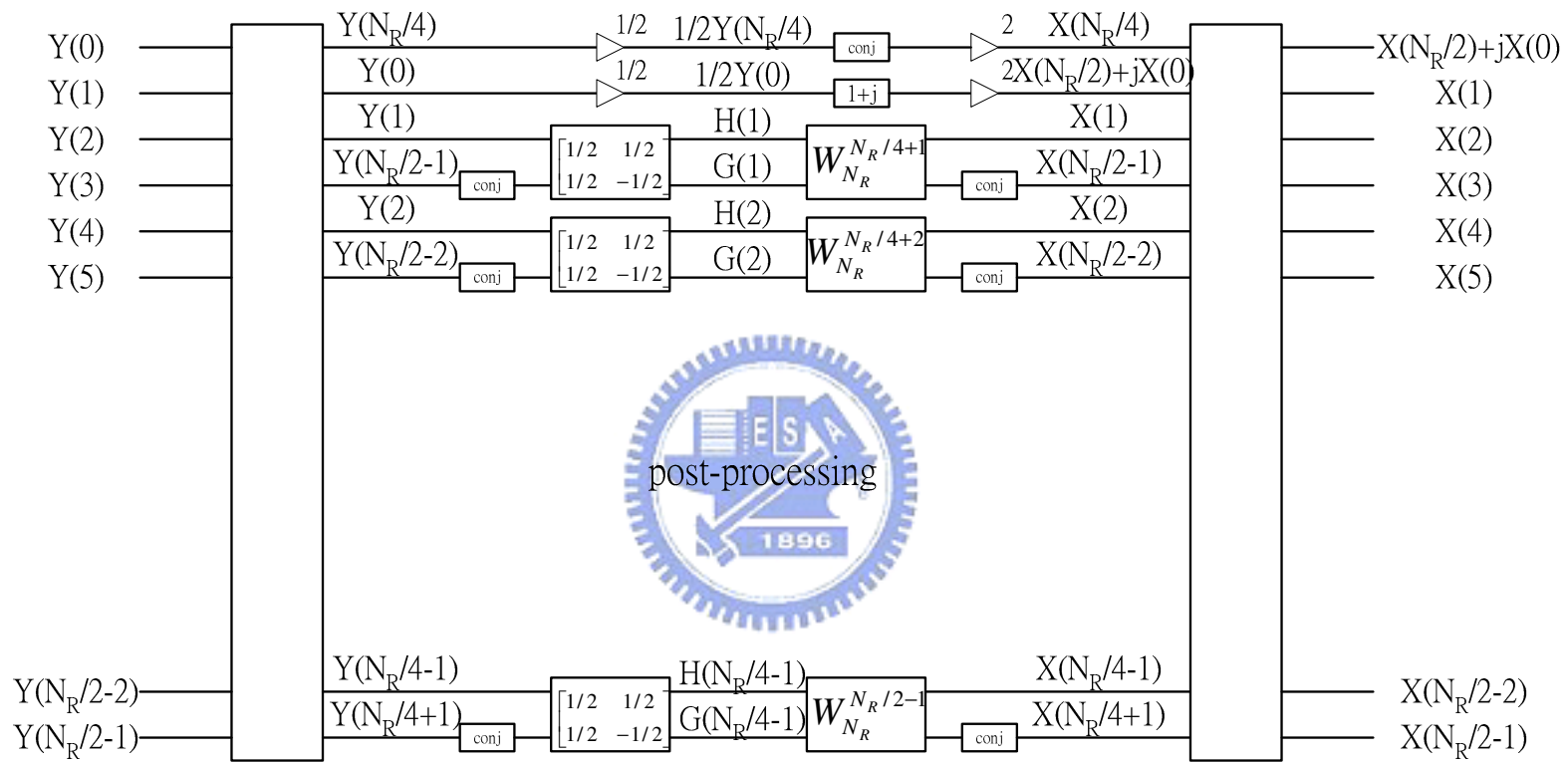


圖 4.26：後端處理器流程圖

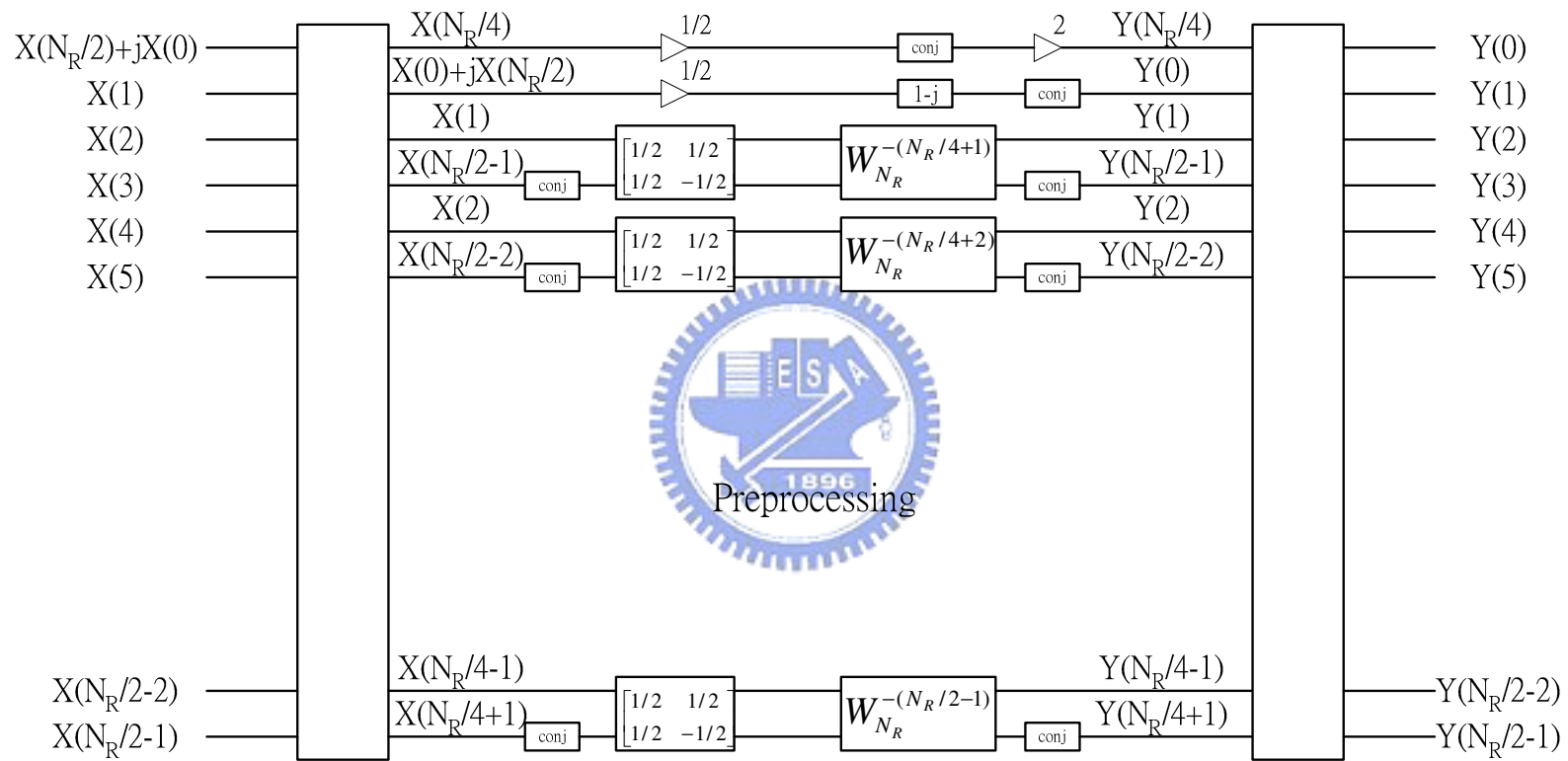


圖 4.28：前端處理器流程圖

RFFT/HS-IFFT 架構：

比較圖(4.26)與圖(4.28)，可以發現在相同階段中，資料和 twiddle factor 的擷取序列都是一樣的，差別只在於 HS-IFFT 擷取完 twiddle factor 後虛取共軛以及每階段所需的 butterfly 運算器架構的不同。故我們可以設計一個專為 RFFT/HS-IFFT 設計的 R_DAG、R_CAG 及一個多模的 butterfly 運算器來將其整合在同一平台上。圖(4.29)為整合後的 BFP memory-based 架構之多模 RFFT/HS-IFFT 處理器方塊圖。

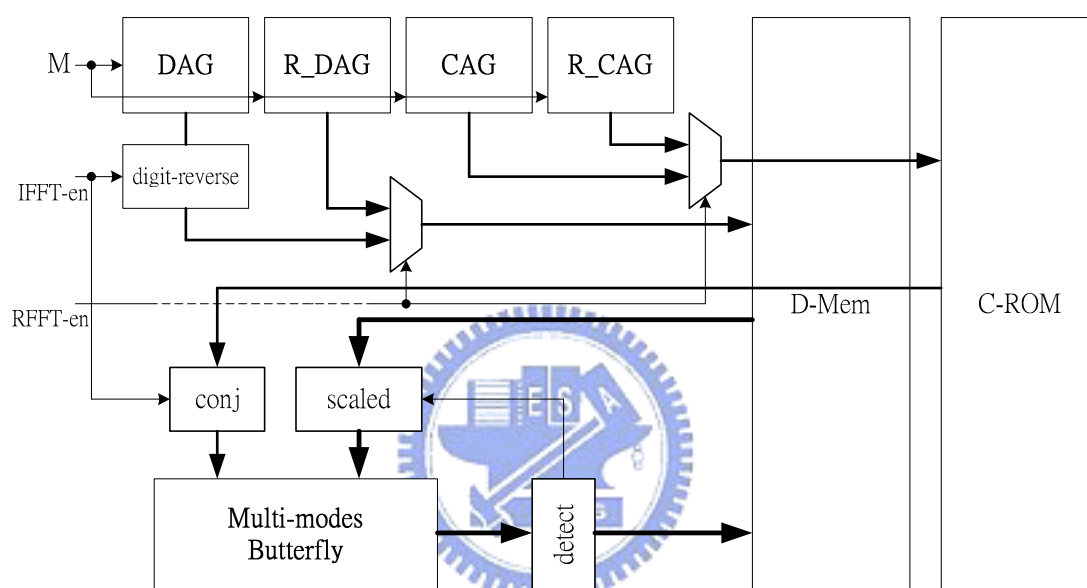


圖 4.29：BFP memory-based 架構之多模 RFFT/HS-IFFT 處理器方塊圖

藉由多模 butterfly 運算器的切換模式來適當地處理 preprocessing/post-processing 以及 CFFT/CIFFT 的每階段運算，且當執行 preprocessing/ post-processing 階段時，其資料及 twiddle factor 將根據 R_DAG 和 R_CAG 所產生的位址去記憶體內擷取。控制訊號 IFFT_en 主要是作為 CFFT 和 CIFFT 兩模式的切換，控制訊號 RFFT_en 為 RFFT/HS-IFFT 與 CFFT/CIFFT 模式的選擇，控制訊號 M 則為 CFFT/CIFFT 轉換點數 N 的選擇，其中 $M = \log_r N$ 。

4.6 結論

在 4.1 節，我們提到了專門處理實數 FFT 與 Hermitian Symmetric IFFT 的演算法，此兩種演算法可以將原本 radix-2 架構 N 點轉換序列的乘法計算複雜度由 $(N/2)\log_2 N$ 減少到 $(N/4+1)\log_2(N/2)$ 。4.2 節、4.3 節主要在討論 radix-2、radix-4 兩不同 radix 的 CFFT/CIFFT 架構，以提供之後 4.4 節 RFFT/HS-IFFT 架構的使用，且在不增加字元長度的狀況下，用 BFP 的方式來改善輸出訊號雜訊比。最後在 4.4 節中，藉由 butterfly 運算器的多模切換和資料、twiddle factor 位址產生器的選擇，RFFT/HS-IFFT 可以很容易地實現在 memory-based 架構上，這將更有效地處理實數序列的快速傅立葉轉換與 Hermitian 對稱序列的反向快速傅力葉轉換。



第五章 應用於實數時域 FFT/IFFT 處理器 之硬體實現與效能分析

5.1 簡介

本章將對第四章所提之處理器架構實現在硬體上，分析其效能且最後利用 FPGA 模擬板驗證其可行性。在 5.2 節、5.3 節將針對不同的 radix 的處理器來探討其硬體的設計，5.4 節將對所設計的處理器做效能分析來決定所需字元長度，最後在 5.5 節裡將用 FPGA 模擬板來驗證所設計處理器的硬體架構與效能。

5.2 Memory-Based Radix-2 多模可變長度 RFFT/HS-IFFT 處理器硬體設計

本節將詳細敘述以 memory-based 架構為基礎的 radix-2 多模可變長度 RFFT/HS-IFFT 處理器硬體設計方式。圖(5.1)為其硬體方塊圖，主要處理方式是根據 DAG(R_DAG)及 CAG(R_CAG)所產生出來的位址去記憶體擷取，再將記憶體所擷取到的兩個資料運算元和一個 twiddle factor 運算元餵給多模 radix-2 butterfly 處理器運算，之後將多模 radix-2 butterfly 的兩輸出值存回原擷取位址。

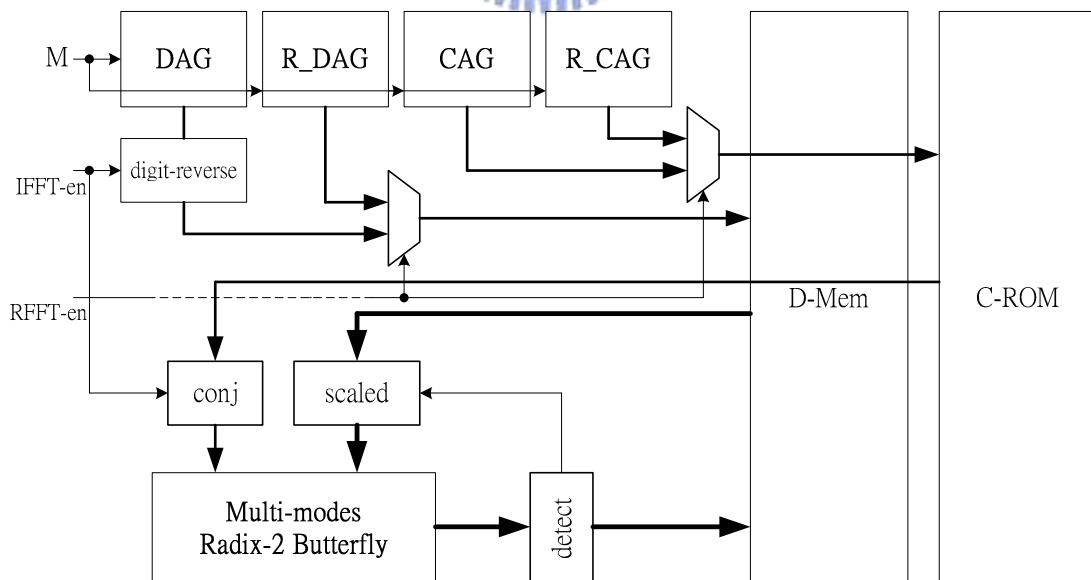


圖 5.1：Radix-2 多模可變長度 RFFT/HS-IFFT 處理器方塊圖

整個架構有三個控制訊號 M 、 $IFFT_en$ 、 $RFFT_en$ ，控制訊號 M ($M = M_R - 1$) 主要在於決定 CFFT/CIFFT 可變長度的點數 N ($N = 2^M$)， N_R ($N_R = 2^{M_R}$) 代表實

數模式或 Hermitain Symmetric 模式時的點數，RFFT_en 訊號與 IFFT_en 訊號是用來選擇所操縱的模式，其操縱模式如表(5.1)所示。

control signal \ mode	FFT	IFFT	RFFT	HS-IFFT
RFFT_en	low	low	high	high
IFFT_en	low	high	low	high

表 5.1：Radix-2 多模處理器操縱模式

5.2.1 DAG 和 R_DAG 硬體設計

DAG：

圖(5.2)為 DAG 的方塊圖，我們知道對於一個 radix-2 架構的處理器來說，其第 i 個 stage 的 shuffle permutation 如式(5.1)所示。且我們將根據此 shuffle permutation 數學式設計出其邏輯架構圖。

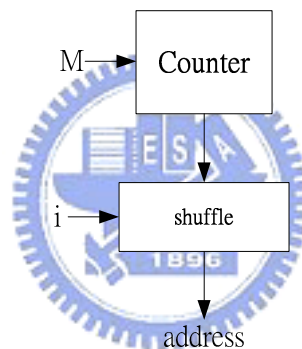


圖 5.2：資料位址產生器方塊圖

Shuffle Permutation 之邏輯架構設計：

第 i 個 stage 的 shuffle permutation 為： $I_{2^{M-1-i}} \otimes S_{2^i, 2} \quad i = 0, 1, \dots, M-1$ (5.1)

將計數器的輸出值 k ($k = 0, 1, \dots, N-1$) 經過適當的 shuffle permutation 後即可得輸入資料位址，如式(5.2)所示。

$$k : [a, b, c] \xrightarrow{\text{shuffle}} \text{address}[a, c, b] \quad (5.2)$$

若將 a 、 b 、 c 分別以二位元表示，則 a 、 b 、 c 位元數分別為 $M-1-i$ 、 i 、1 位元。由圖(5.3)與(5.4)我們可以很容易得到 shuffle permutation 的邏輯架構，其中方塊上方的值代表其位元數，而方塊內的值則為欲 shuffle permutation 的值。藉由產生一個與計數器輸出同位元的 mask 分別跟計數器輸出做相對位元間"AND"，之後再將其全部的結果做相對位元間"OR"即可得到 shuffle

permutation 後的輸入資料位址。圖(5.5)為 DAG 的邏輯架構圖，其中 M 為欲轉換點數 N 的選擇($N=2^M$)，i 為所在階段。因為 Radix-2 butterfly 處理器同時要有兩個資料運算元，故需要有兩個 DAG 來同時產生輸入資料位址。



圖 5.3 : $k:[a,b,c] \xrightarrow{\text{shuffle}} \text{address}[a,c,b]$ 之二位元表示圖

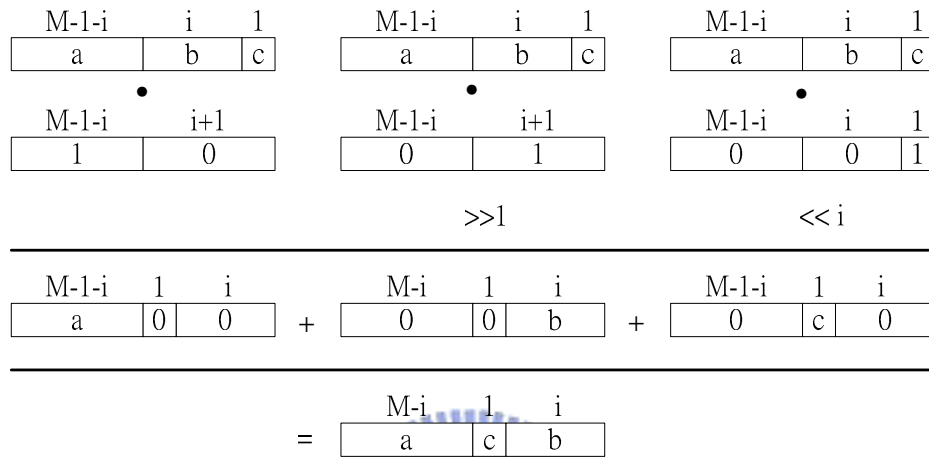


圖 5.4 : $k:[a,b,c] \xrightarrow{\text{shuffle}} \text{address}[a,c,b]$ 的示意圖

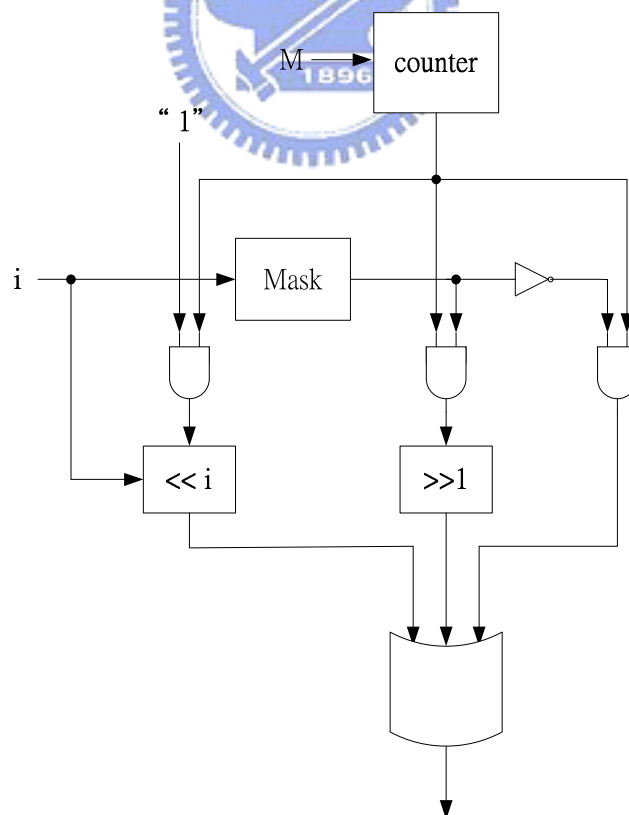


圖 5.5 : Radix-2 DAG 邏輯架構圖

R_DAG :

由圖(4.26)和圖(4.28)可知 post-processing 與 preprocessing 欲擷取位址的順序為 $\{N_R/4, 0\}$ 、 $\{1, N_R/2-1\}$ 、 $\{2, N_R/2-2\}$ 、 \dots 、 $\{i, N_R/2-i\}$ 、 \dots 、 $\{N_R/4-1, N_R/4+1\}$ ，此模式下所用到的 CFFT/CIFFT 的轉換點數 N 為原本大小的一半，故將其變數代換 $N_R/2=N$ 後可得 $\{N/2, 0\}$ 、 $\{1, N-1\}$ 、 $\{2, N-2\}$ 、 \dots 、 $\{i, N-i\}$ 、 \dots 、 $\{N/2-1, N/2+1\}$ ，下圖(5.6)為 R_DAG 的邏輯架構圖，因為 radix-2 butterfly 有兩個輸入資料運算元，故 R_DAG 得同時產生兩個輸入資料位址。其中 M 為 CFFT/CIFFT 轉換點數 N 的選擇 ($N=2^M$)。

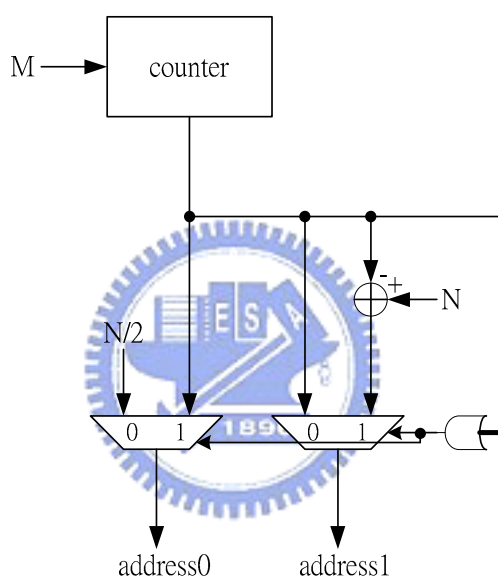


圖 5.6 : Radix-2 R_DAG 邏輯架構圖

Conflict Free Memory Addressing :

為了能讓多模 radix-2 butterfly 運算器能夠同時擷取到所需的資料運算元，我們必須使用兩個 memory banks，並將資料適當的儲存在此兩 memory banks 內，以達到同時擷取的目的。若為 CFFT/CIFFT 運算模式，則可根據下式來決定資料的配置方式。其 CFFT 的原始輸入儲存位置為位元反向儲存，所得之原始輸出位置在記憶體內為正常序列；CIFFT 則與之相反，原始輸入儲存位置為正常序列儲存，所得之原始輸出位置在記憶體內為位元反向儲存。

$$data_original_address = (d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_2 \quad (5.3)$$

$$memory_bank = (d_{n-1} + d_{n-2} + \dots + d_2 + d_1 + d_0) \bmod 2 \quad (5.4)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_2 \quad (5.5)$$

$$n = \log_2 N$$

圖(5.7)為記憶體區塊配置(Memory Bank Assignment, MBA)的方塊圖，其中 MB 代表所屬的 memory bank，MB_addr 則為所在 memory bank 的位址。

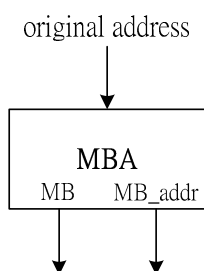


圖 5.7：Radix-2 記憶體區塊配置方塊圖

當運算模式為 RFFT/HS-IFFT 時，因為 post-processing/preprocessing 根據上面的配置方式是無法同時擷取到所要的資料運算元，例 32 點的 RFFT 在 post-processing 階段時想同時擷取原始記憶體位址為{2,14}的兩個位址，但根據式(5.4)的配置方式 2、14 都屬於同一記憶體區塊內(bank1)導致無法同時擷取，故我們需另外設計一套專為 post-processing/preprocessing 的資料配置方式，並適時地將此兩種配置方式做適當的互換以達到每階段都能同時擷取的目的。下式為 post-processing/preprocessing 階段時的記憶體配置方式，主要相異之處在於記憶體區塊的選擇方式，故 2 與 14 分別配置於 bank0 和 bank1 兩不同的區塊。

$$data_original_address = (d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_2 \quad (5.6)$$

$$memory_bank = d_{n-1} \text{ xor } d_0 \quad (5.7)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_2 \quad (5.8)$$

$$n = \log_2 N$$

之前有提到因為 post-processing/preprocessing 階段與 CFFT/CIFFT 階段所採用的記憶體區塊配置方式是不同的，故配置方式得適時地在此兩不同階段間做適當的互換。其互換方式如下所述，當為 RFFT 運算模式時，先以 CFFT/CIFFT 配置模

式配置，等到執行到 CFFT 的最後一階段時，在順勢更改成為 post-processing/preprocessing 的配置模式儲存回去，例 32 點 RFFT 其 CFFT 最後一階段的某一 butterfly 運算擷取的位址為{2,10}，其 2 進位表示式為{0010,1010}，擷取區塊及位址分別為{1,0}、{010,010}，運算完後儲存回去區塊及位址分別為{0,1}、{010,010}，且因為存回的區塊位址都是一樣的，故可以保持 in-place 模式。同理，當為 HS-IFFT 運算模式時，先以 post-processing/preprocessing 配置模式配置，等到執行到 CIFFT 的第一階段時，在順勢更改成為 CFFT/CIFFT 的配置模式儲存回去。下圖(5.8)為具有兩套記憶體區塊配置(MBA)的方塊圖，MB_R 為 post-processing/preprocessing 配置模式的 memory bank 選擇。

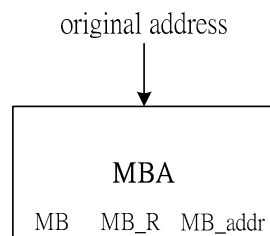


圖 5.8：Radix-2 記憶體區塊配置方塊圖

5.2.2 CAG 和 R_CAG 硬體設計

CAG：

圖(5.9)為 CAG 的方塊圖。其中 i 為所在階段， $M_R \max$ 為可變點數處理器所能處理的最大點 RFFT/HS-IFFT 數範圍，因為在 coefficient ROM 裡所儲存的 twiddle factor 必須包含最大點數的 twiddle factor。CAG 的邏輯架構很簡單，只需一個移位器即可達到。

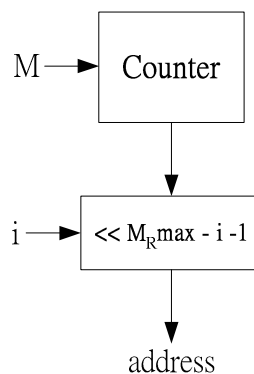


圖 5.9：Radix-2 係數位址產生器方塊圖

R_CAG :

同樣地，由圖(4.26)和圖(4.28)可知 post-processing 與 preprocessing 在第二階段欲擷取的 twiddle 的順序為 $1 \pm j$ 、 $W_{N_R}^{N_R/4+1}$ 、 $W_{N_R}^{N_R/4+2}$ 、 \dots 、 $W_{N_R}^{N_R/2-1}$ ，其等效於 $W_{N_R}^0 \pm jW_{N_R}^0$ 、 $-jW_{N_R}^1$ 、 $-jW_{N_R}^2$ 、 \dots 、 $-jW_{N_R}^{N_R/4-1}$ ，下圖(5.10)為 R_CAG 的邏輯架構圖。其中 M 為 CFFT/CIFFT 轉換點數 N 的選擇 ($N = 2^M$)， $M_R \max$ 為可變點數 RFFT/HS-IFFT 處理器所能處理的最大點數 $N_R \max$ 的範圍 ($N_R \max = 2^{M_R \max}$)。

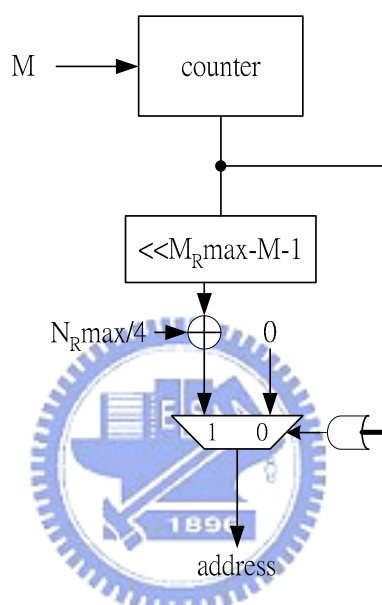


圖 5.10：Radix-2 R_CAG 邏輯架構圖

Reduce Coefficient Memory Size :

就如前一章節所說的，我們可以藉由 twiddle factor 相互間的相關性減少 twiddle factor 在唯讀記憶體儲存的個數。其方式如下，首先先根據 twiddle factor 的原始儲存位置經由式(5.9)來決定所屬之 block，決定了所屬的 block 之後，接下來根據式(5.10)(5.11)得到與 block I 所對應的位址，最後在根據每個 block 與 block I 之間的映射關係把所擷取出之 twiddle factor 做適當的映射即可得所要的 twiddle factor。

$$block = (b_{MSB}, b_{MSB-1} \& (b_{MSB-2} | b_{MSB-3} | \dots | b_1 | b_0)) = (s_0, s_1) \quad (5.9)$$

$$block \text{ I }、block \text{ III} : address = (b_{MSB-1}, b_{MSB-2}, \dots, b_2, b_1, b_0) \quad (5.10)$$

$$block \text{ II }、block \text{ IV} : address = \sim (b_{MSB-1}, b_{MSB-2}, \dots, b_2, b_1, b_0) + 1 \quad (5.11)$$

圖(5.11)為根據上述所設計出的 reduce coefficient memory size 架構圖，其 C-ROM 所儲存的 twiddle factor 個數由原本的 $N_R/2$ 降至 $N_R/8+1$ ，且為了有效地使用記憶體，C-ROM 只需儲存 $N_R/8$ 個 twiddle factor 即可，剩餘的一個 twiddle factor 可用多工器(mux)來選擇。

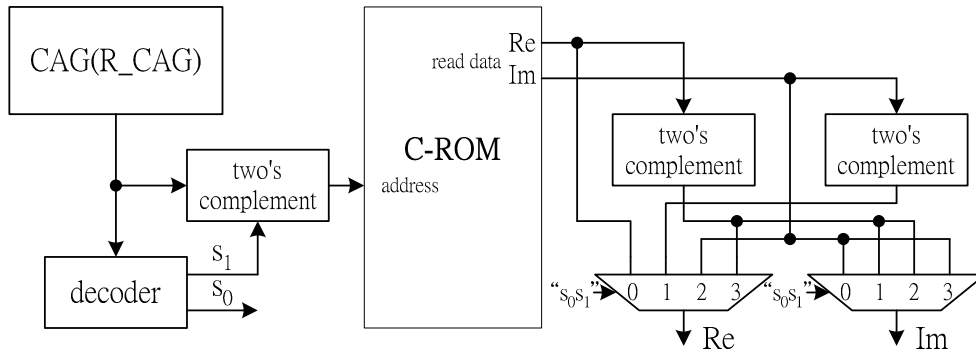


圖 5.11：Reduce coefficient memory size 架構圖

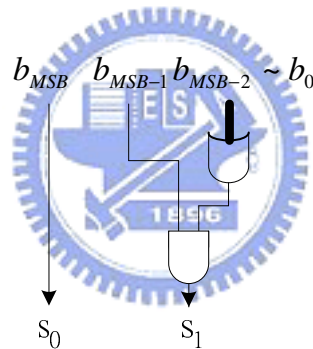


圖 5.12：Decoder 內部邏輯架構

5.2.3 多模 Radix-2 Butterfly 處理器硬體設計

為了不額外花費硬體，在 RFFT 或 HS-IFFT 的模式下，其 post-processing 和 preprocessing 的運算單元可與 CFFT/CIFFT 的 radix-2 butterfly 運算器整合在一起成為一個具有多種運算模式的多模 radix-2 butterfly 運算器，並藉由一些控制訊號決定其運算的資料路徑。此多模 radix-2 butterfly 運算器具有以下六種運算模式，mode 1 為 CFFT/CIFFT 的基本 radix-2 butterfly 運算器，mode 2、mode 3 為 RFFT/HS-IFFT 的第一階段所需運算器，mode 4、mode 6 為 RFFT 第二階段所需運算器，mode 5、mode 6 則為 HS-IFFT 第二階段所需運算器。圖(5.14)為多模 radix-2 butterfly 運算器架構圖。

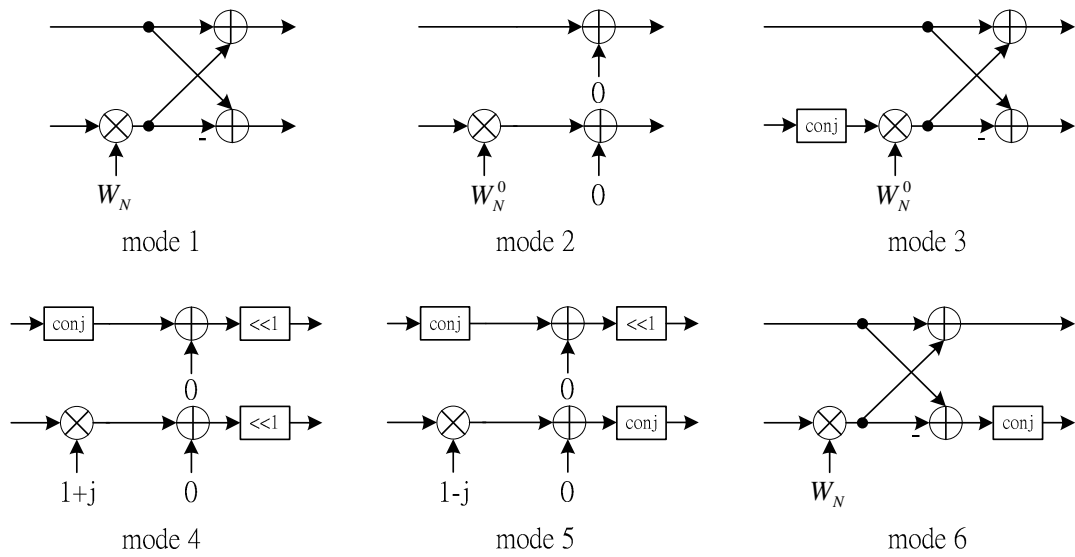


圖 5.13：多模 radix-2 butterfly 運算器的六種運算模式

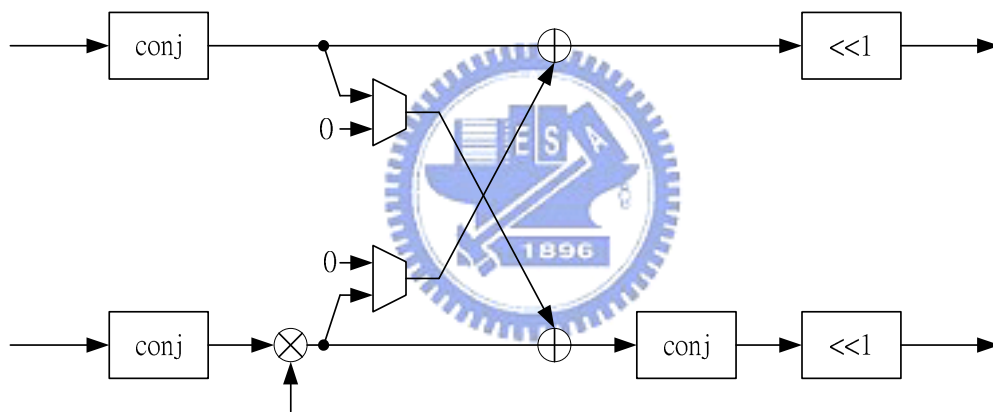


圖 5.14：多模 radix-2 butterfly 運算器架構

此外，因為本論文所採 memory-based 架構所用的資料記憶體為 single port 架構，讀寫不可能同時發生，故每筆資料從記憶體內擷取出後再經運算寫回去記憶體的區間為兩個 clock cycles，如圖(5.15)所示，D 為從記憶體內擷取至暫存器的讀取資料，D' 為運算完後等待寫回記憶體的暫存器儲存資料。Butterfly 運算器有兩個 cycles 期間可供運算，我們可以善用這兩個 cycles 將一個複數運算所需的實數乘法器由四個降為兩個，如圖(5.16)所示。

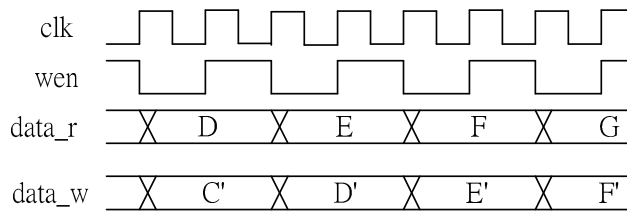


圖 5.15：Single port 記憶體 timing diagram

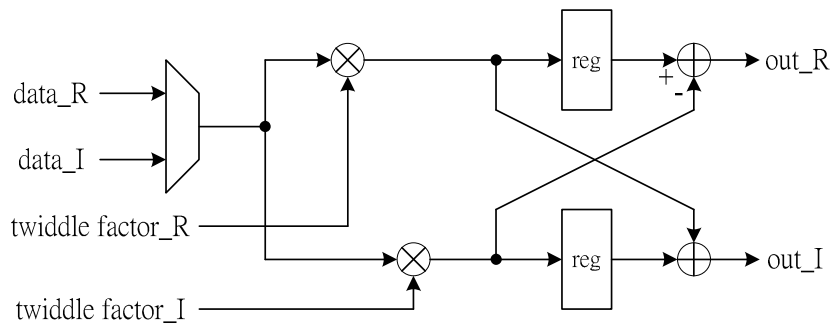


圖 5.16：以兩個實數乘法器實現在兩個 clock cycles 內運算的複數運算架構圖

此多模 radix-2 butterfly 運算器，可以有效的將所有的運算單元整合在一起，且因為提供了兩個 cycles 運算期間的緣故，故只需用兩個實數乘法器就可以完成複數運算，節省了不少運算單元的面積。

此外，因為 radix-2 butterfly 運算器為有限字元長度的硬體架構，故對於乘法運算及加法運算都必須考慮到精確度問題。一個二進二出的 radix-2 butterfly 運算器，其基本運算方程式如式(5.12)(5.13)所示。

$$output0 = input0 + input1 \times W_N^\theta \quad (5.12)$$

$$output1 = input0 - input1 \times W_N^\theta \quad (5.13)$$

其中 W_N 為複數座標的旋轉因子，且因為輸入字元長度有限，令 K 為有限字元長度所能表示的最大值，可得 $\text{Re}\{input0\} \leq K$ 、 $\text{Im}\{input0\} \leq K$ 、 $\text{Re}\{input1\} \leq K$ 、 $\text{Im}\{input1\} \leq K$ 。如圖(5.17)所示，對於任何輸入訊號 $input0$ 、 $input1$ 在複數座標上所對應的值都落在小正方形內，而最外圍的線所框起來的範圍則為最後

$output0$ 與 $output1$ 在複數座標上對應值的範圍。由圖可知，所獲得的輸出訊號其實部(虛部)最大值為原本輸入訊號實部(虛部)最大值的 $1 + \sqrt{2}$ 倍，也就是說最後

輸出字元長度要比輸入字元長度多兩位元以避免溢位發生，但由於本處理器所使用的記憶體字元長度固定，故 radix-2 butterfly 運算器運算前與運算後的字元長度

需保持一樣。根據軟體模擬，我們統計出輸出訊號落在陰影部分的機率為 0.0064，此機率非常小，幾乎所有的輸出值都落在大正方形內(大方塊內只需增加一位元的字元長度即可)，故為了使輸出和輸入位元長度一致，以及增加精確度的考量，我們將原本需多兩位元的輸出值，先後對其捨棄一位元的 MSB 並判斷是否需設為飽和值，以及捨棄一位元的 LSB 來符合要求。

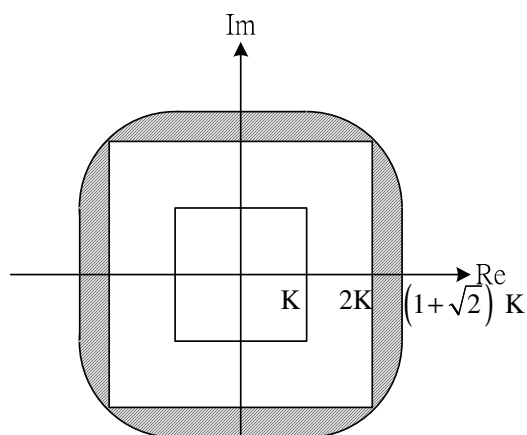


圖 5.17：Radix-2 butterfly 輸入範圍和輸出範圍的相關圖

5.2.4 Radix-2 多模可變長度 RFFT/HS-IFFT 硬體架構

圖(5.18)為 radix-2 多模可變長度 RFFT/HS-IFFT 硬體架構圖。DAG0、DAG1 以及 CAG 分別提供 CFFT/CIFFT 模式的兩個資料運算元位址和 twiddle factor 運算元位址，DAG_R、CAG_R 則提供 RFFT/HS-IFFT 模式在 post-processing/preprocessing 階段時的兩個資料運算元位址和 twiddle factor 運算元位址。MBA 主要在處理 DAG 所產生的位址被分配到哪個記憶體區塊位址內，MB0、MB1、C-ROM 分別代表儲存資料運算元及 twiddle factor 運算元的記憶體，detecte 及 scaled 則是做 BFP 處理來改善精確度，且整個架構有三個 register files 用來加速硬體的速度。

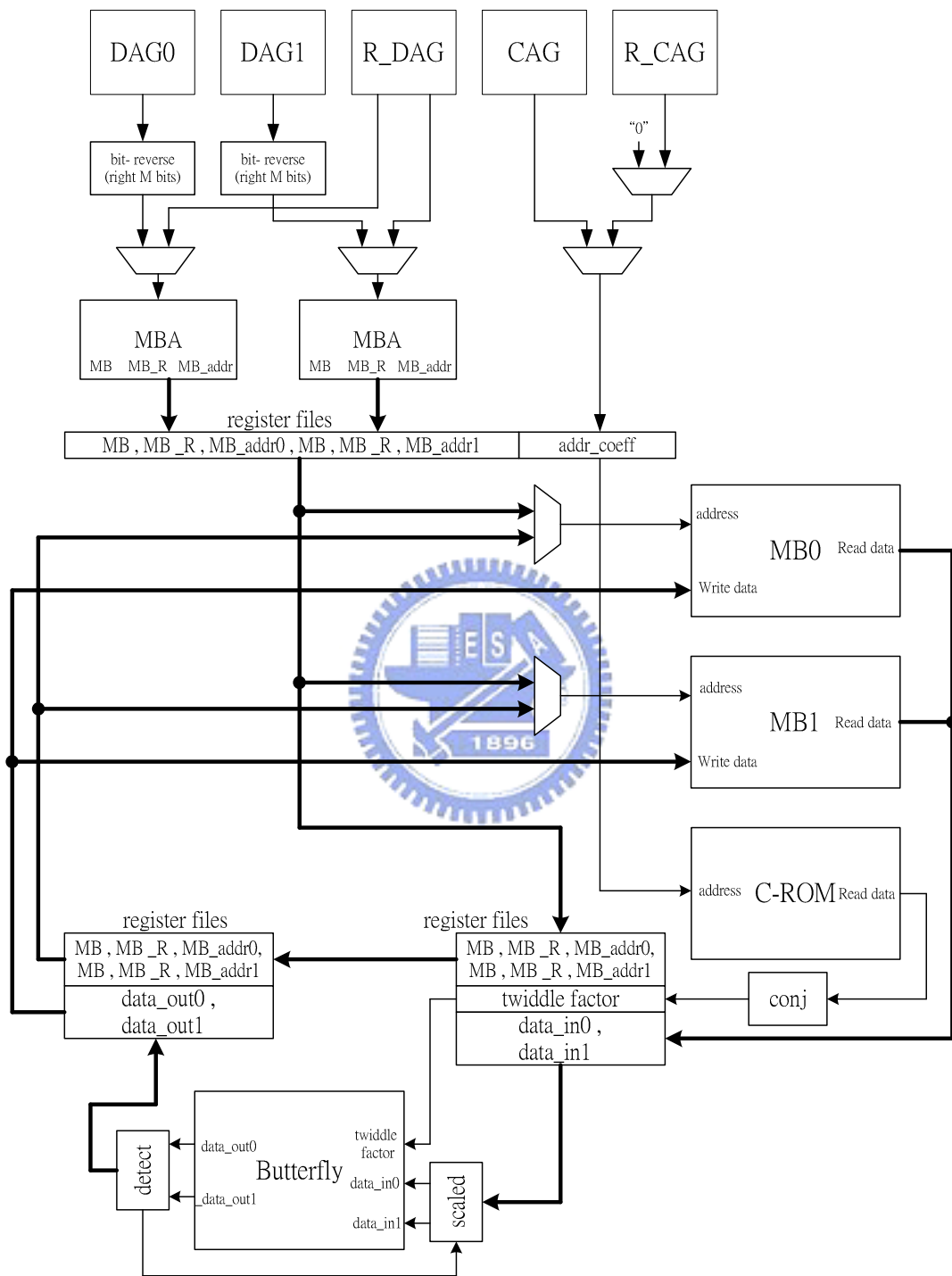


圖 5.18 : Radix-2 多模可變長度 RFFT/HS-IFFT 硬體架構圖

5.3 Memory-Based Radix-4 多模可變長度 RFFT/HS-IFFT 處理器硬體設計

下圖(5.19)為 memory-based radix-4 多模可變長度 RFFT/HS-IFFT 架構之方塊圖。主要處理方式與 radix-2 架構大同小異，都是根據 DAG(R_DAG)及 CAG(R_CAG)所產生出來的位址去記憶體擷取，再將記憶體所擷取到的四個資料運算元和三個 twiddle factor 運算元餵給多模 radix-4 butterfly 處理器運算，之後將多模 radix-4 butterfly 的兩輸出值存回原擷取位址。

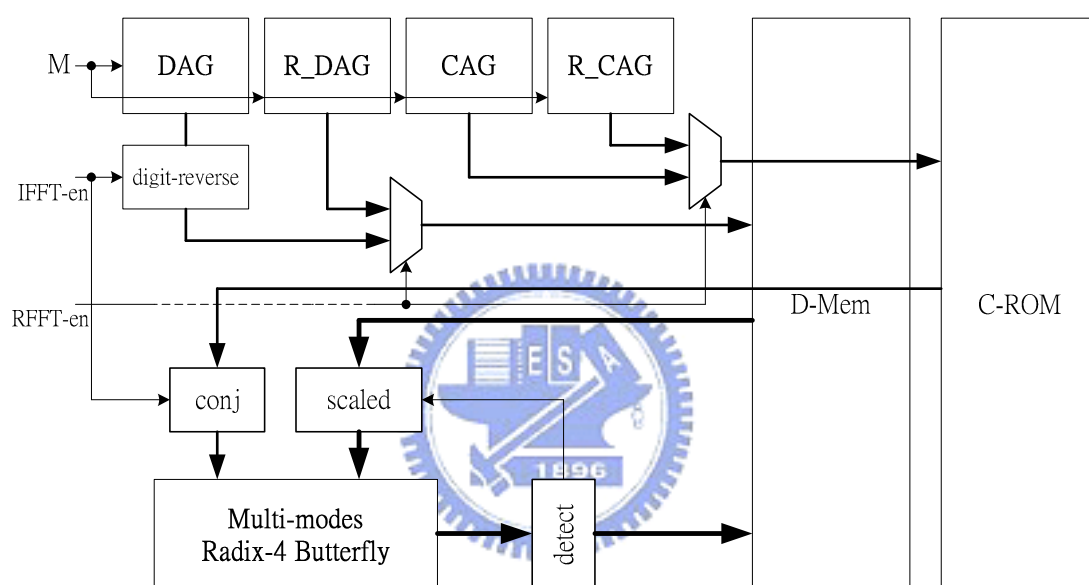


圖 5.19：Radix-4 多模可變長度 RFFT/HS-IFFT 處理器方塊圖

整個架構有三個控制訊號 M 、IFFT_en、RFFT_en，控制訊號 M 主要在於決定 CFFT/CIFFT 可變長度的點數 N ($N = 4^M$)， N_r ($N_r = 2N = 2^{M_r}$) 代表實數模式或 Hermitain Symmetric 模式時的點數，RFFT_en 訊號與 IFFT_en 訊號是用來選擇所操縱的模式，其操縱模式如表(5.2)所示。

mode control signal	FFT	IFFT	RFFT	HS-IFFT
RFFT_en	low	low	high	high
IFFT_en	low	high	low	high

表 5.2：Radix-4 多模處理器操縱模式

5.3.1 DAG 和 R_DAG 硬體設計

DAG :

對於一個 radix-4 架構的處理器來說，其第 i 個 stage 的 shuffle permutation 如式 (5.14) 所示。圖(5.20)為根據此 shuffle permutation 數學式所設計出的邏輯架構圖，因為 radix-4 架構同時要有四個資料運算元，故需要四組 DAG 來產生資料位址。

$$\text{第 } i \text{ 個 stage 的 shuffle permutation 為： } I_{4^{M-1-i}} \otimes S_{4^i, 4} \quad i = 0, 1, \dots, M-1 \quad (5.14)$$

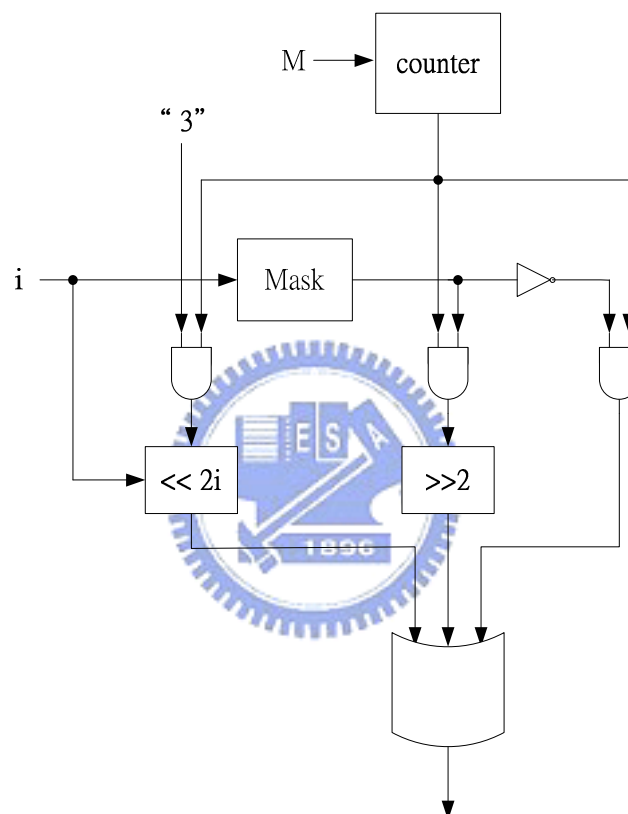


圖 5.20 : Radix-4 DAG 邏輯架構圖

R_DAG :

因為要同時擷取四個資料運算元，且考慮到記憶體區塊配置互換的問題，故需運算順序需更改為 $\{(N/2, 0), (N/4, 3N/4)\}, \{1, N-1), (N/4+1, 3N/4-1)\}, \dots, \{(N/4-1, 3N/4+1), (N/2-1, N/2+1)\}$ ，以此順序運算則四個運算元可有相同的記憶體區塊位址，藉以解決記憶體區塊配置互換問題。圖(5.21)為 R_DAG 的邏輯架構圖，其中 M 為 CFFT/CIFFT 轉換點數 N 的選擇 ($N = 4^M$)。

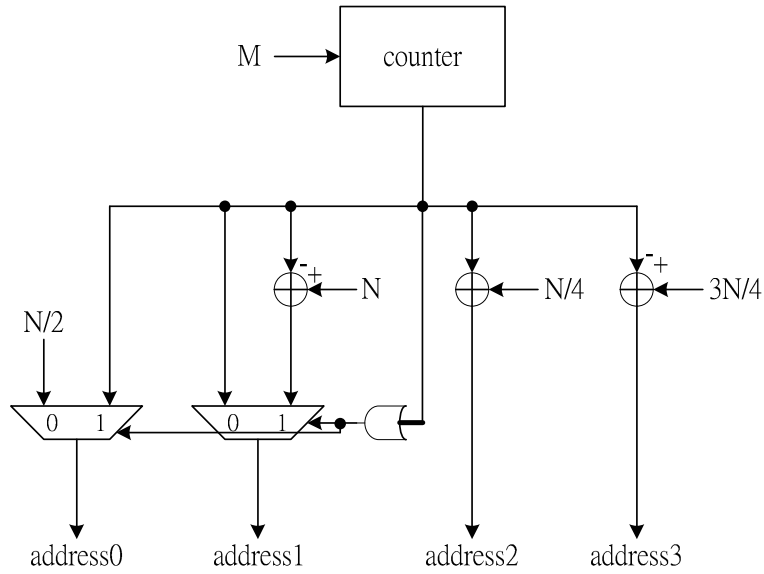


圖 5.21：Radix-4 R_DAG 邏輯架構圖

Conflict Free Memory Addressing :

同樣地，為了能讓多模 radix-4 butterfly 運算器能夠同時擷取到所需的資料運算元我們必須使用四個 memory banks，並將資料適當的儲存在此四個 memory banks 內，以達到同時擷取的目的。下式為 CFFT/CIFFT 運算模式的配置方式，其 CFFT 的原始輸入儲存位置為位元反向儲存，所得之原始輸出位置在記憶體內為正常序列；CIFFT 則與之相反，原始輸入儲存位置為正常序列儲存，所得之原始輸出位置在記憶體內為位元反向儲存。

$$data_original_address = (d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_4 \quad (5.15)$$

$$memory_bank = (d_{n-1} + d_{n-2} + \dots + d_2 + d_1 + d_0) \bmod 4 \quad (5.16)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_4 \quad (5.17)$$

$$n = \log_4 N$$

與 radix-2 架構相同，radix-4 架構在 RFFT/HS-IFFT 的 post-processing/preprocessing 階段也會遇到記憶體區塊配置的衝突，故需有另一套配置方式，此配置方式如下所示，其記憶體區塊配置的互換時機與 radix-2 架構相同。圖(5.22)為記憶體區塊配置(MBA)的方塊圖。

$$data_original_address = (d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_4 \quad (5.18)$$

$$data_original_address = (b_{2n-1}, b_{2n-2}, \dots, b_2, b_1, b_0)_2 \quad (5.19)$$

$$memory_bank = (b_{2n-1} \text{ xor } b_0, b_{2n-2} \text{ xor } b_1)_2 \quad (5.20)$$

$$memory_bank_address = (d_{n-2}, \dots, d_2, d_1, d_0)_4 \quad (5.21)$$

$$n = \log_4 N$$

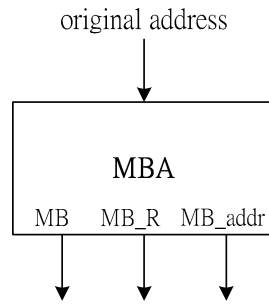


圖 5.22：Radix-4 記憶體區塊配置方塊圖

5.3.2 CAG 和 R_CAG 硬體設計

CAG：

因為 radix-4 butterfly 運算器需要三個 twiddle factor 運算元，故 CAG 需產生三組不同的 twiddle factor 位址，且必須有三組 C-ROM 來儲存以達到同時擷取的可能。圖(5.23)為 CAG 的邏輯架構圖，所需三個 twiddle factor 運算元分別為 W_N^k 、 W_N^{2k} 、 W_N^{3k} ，coeff0_addr、coeff1_addr、coeff2_addr 分別為所在的 C-ROM 位址，其中 M 為 CFFT/CIFFT 轉換點數 N 的選擇 ($N = 4^M$)， i 為所在階段。

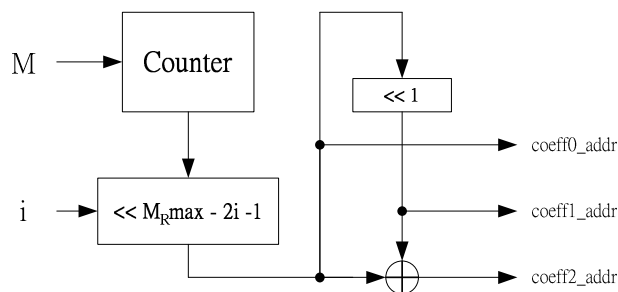


圖 5.23：Radix-4 CAG 邏輯架構圖

R_CAG :

在 RFFT/HS-IFFT 的 post-processing/preprocessing 中，資料運算元位址擷取的順序為 $\{(N/2,0),(N/4,3N/4)\}$ 、 $\{1,N-1),(N/4+1,3N/4-1)\}$ 、 \dots 、 $\{(N/4-1,3N/4+1),(N/2-1,N/2+1)\}$ ，故所需的 twiddle factor 運算元順序為 $\{1 \pm j, W_{N_R}^{N_R/4+N_R/8}\}$ 、 $\{W_{N_R}^{N_R/4+1}, W_{N_R}^{N_R/4+N_R/8+1}\}$ 、 \dots 、 $\{W_{N_R}^{N_R/4+N_R/8-1}, W_{N_R}^{N_R/4+N_R/4-1}\}$ ，等效於 $\{W_{N_R}^0 \pm jW_{N_R}^0, -jW_{N_R}^{N_R/8}\}$ 、 $\{-jW_{N_R}^1, -jW_{N_R}^{N_R/8+1}\}$ 、 \dots 、 $\{-jW_{N_R}^{N_R/8-1}, -jW_{N_R}^{N_R/4-1}\}$ 。下圖(5.24)為 R_CAG 的邏輯架構圖，其中 M 為 CFFT/CIFFT 轉換點數 N 的選擇 ($N = 4^M$)， $M_R \max$ 為可變點數 RFFT/HS-IFFT 處理器所能處理的最大點數 ($N_R \max$ 的範圍 ($N_R \max = 2^{M_R \max}$))。

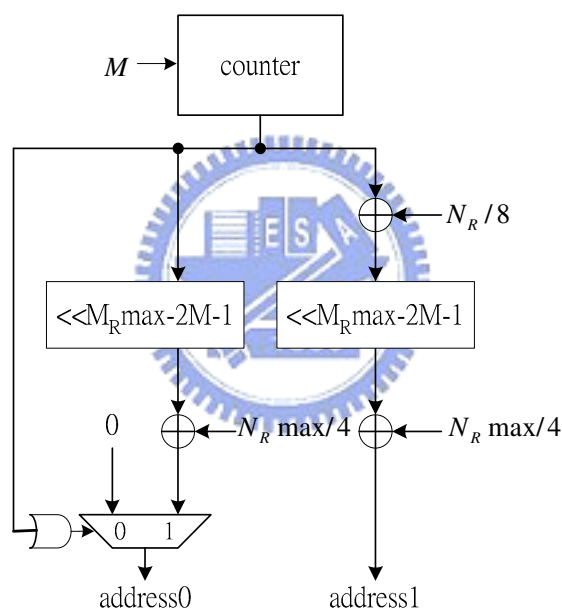


圖 5.24 : Radix-4 R_CAG 邏輯架構圖

5.3.3 多模 Radix-4 Butterfly 運算器硬體設計

圖(5.25)為基本模式的 radix-4 butterfly 運算器，此多模運算器還具有以下五種運算模式，如圖(5.26)所示。同理，此多模運算器的複數乘法運算可用兩個實數乘法器來實現。

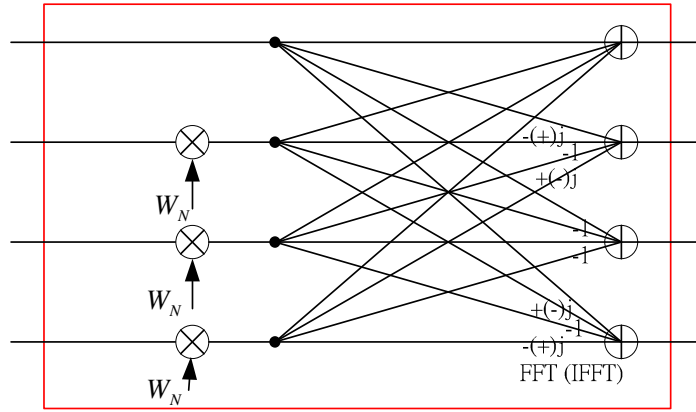
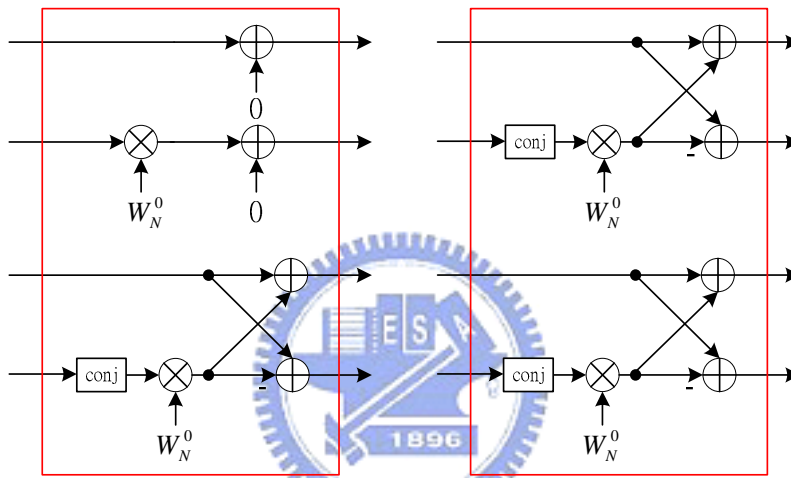
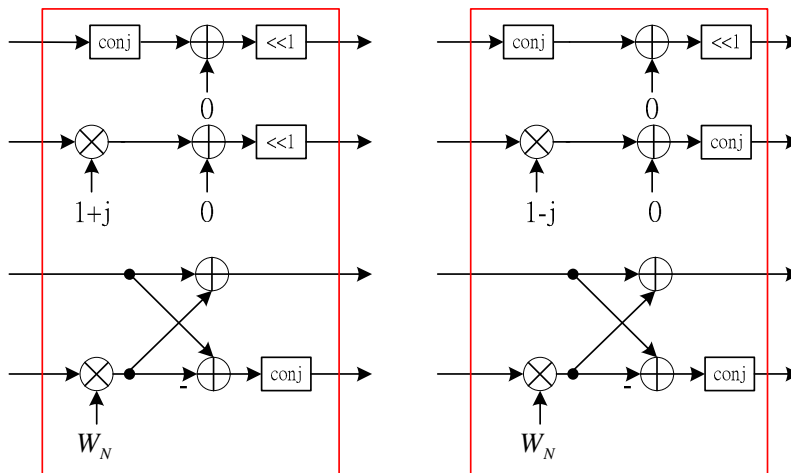


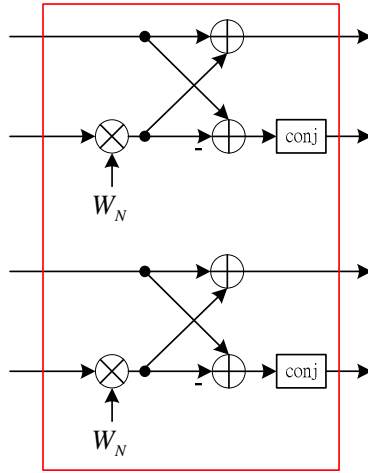
圖 5.25 : Basic radix-4 butterfly 運算器



post(pre)processing 第一階段運算模式 1 post(pre)processing 第一階段運算模式 2



post-processing 第二階段運算模式 1 preprocessing 第二階段運算模式 1



post(pre)processing 第二階段運算模式 2

圖 5.26：多模 radix-4 butterfly 運算器之其它運算模式

此外，因為 radix-4 butterfly 運算器也為有限字元長度的硬體架構，故對於乘法運算及加法運算也須考慮到精確度問題。一個四進四出的 radix-4 butterfly 運算器，其基本運算方程式如式(5.22)(5.23)(5.24)(5.25)所示。

$$output0 = input0 + input1 \times W_N^\theta + input2 \times W_N^{2\theta} + input3 \times W_N^{3\theta} \quad (5.22)$$

$$output1 = input0 + j \times input1 \times W_N^\theta - input2 \times W_N^{2\theta} - j \times input3 \times W_N^{3\theta} \quad (5.23)$$

$$output2 = input0 - input1 \times W_N^\theta + input2 \times W_N^{2\theta} - input3 \times W_N^{3\theta} \quad (5.24)$$

$$output3 = input0 - j \times input1 \times W_N^\theta - input2 \times W_N^{2\theta} + j \times input3 \times W_N^{3\theta} \quad (5.25)$$

其中 W_N 為複數座標的旋轉因子，且因為輸入字元長度有限，令 K 為有限字元長度所能表示的最大值，可得 $\text{Re}\{input0\} \leq K$ 、 $\text{Im}\{input0\} \leq K$ 、 $\text{Re}\{input1\} \leq K$ 、 $\text{Im}\{input1\} \leq K$ 、 $\text{Re}\{input2\} \leq K$ 、 $\text{Im}\{input2\} \leq K$ 、 $\text{Re}\{input3\} \leq K$ 、 $\text{Im}\{input3\} \leq K$ 。如圖(5.27)所示，對於任何輸入訊號 $input0$ 、 $input1$ 、 $input2$ 、 $input3$ 在複數座標上所對應的值都落在小正方形內，而最外圍的線所框起來的範圍則為最後 $output0$ 、 $output1$ 、 $output2$ 與 $output3$ 在複數座標上對應值的範圍。由圖可知，所獲得的輸出訊號其實部(虛部)最大值為原本輸入訊號實部(虛部)最大值的 $1+3\sqrt{2}$ 倍，也就是說最後輸出字元長度要比輸入字元長度多三位元以避免溢位發生，但由於本處理器所使用的記憶體字元長度固定，故 radix-4 butterfly 運算器運算前與運算後的字元長度需保持一樣。同樣地根據軟體模擬，我們統計出輸出訊號落在陰影部分的機率為 0.000097，幾乎所有的輸出值都落在大正方形

內(大方塊內只需增加二位元的字元長度即可)，故為了使輸出和輸入位元長度一致，以及增加精確度的考量，我們將原本需多三位元的輸出值，先後對其捨棄一位元的 MSB 並判斷是否需設為飽和值，以及捨棄二位元的 LSB 來符合要求。

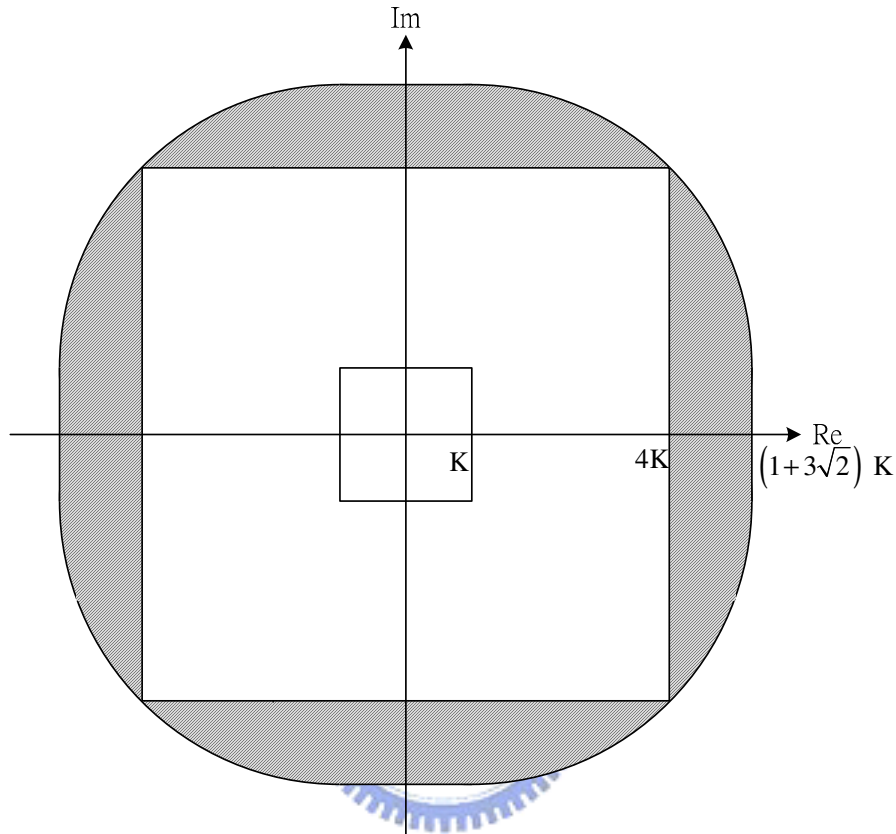


圖 5.27：Radix-4 butterfly 輸入範圍和輸出範圍的相關圖

5.3.4 Radix-4 多模可變長度 RFFT/HS-IFFT 硬體架構

圖(5.28)為 radix-4 多模可變長度 RFFT/HS-IFFT 的硬體架構圖。與 radix-2 架構大同小異，由 DAG 和 CAG 來提供 CFFT/CIFFT 模式的四個資料運算元位址和三個 twiddle factor 運算元位址，DAG_R、CAG_R 則提供 RFFT/HS-IFFT 模式在 post-processing/preprocessing 階段時的四個資料運算元位址和兩個 twiddle factor 運算元位址。MBA 將 DAG(DAG_R)所產生的四個資料運算元位址做記憶體區塊的配置。MB0、MB1、MB2、MB3、C-ROM 分別代表儲存資料運算元及 twiddle factor 運算元的記憶體，detecte 及 scaled 則是做 BFP 處理來改善精確度，整個架構有三個 register files 用來加速硬體的速率。

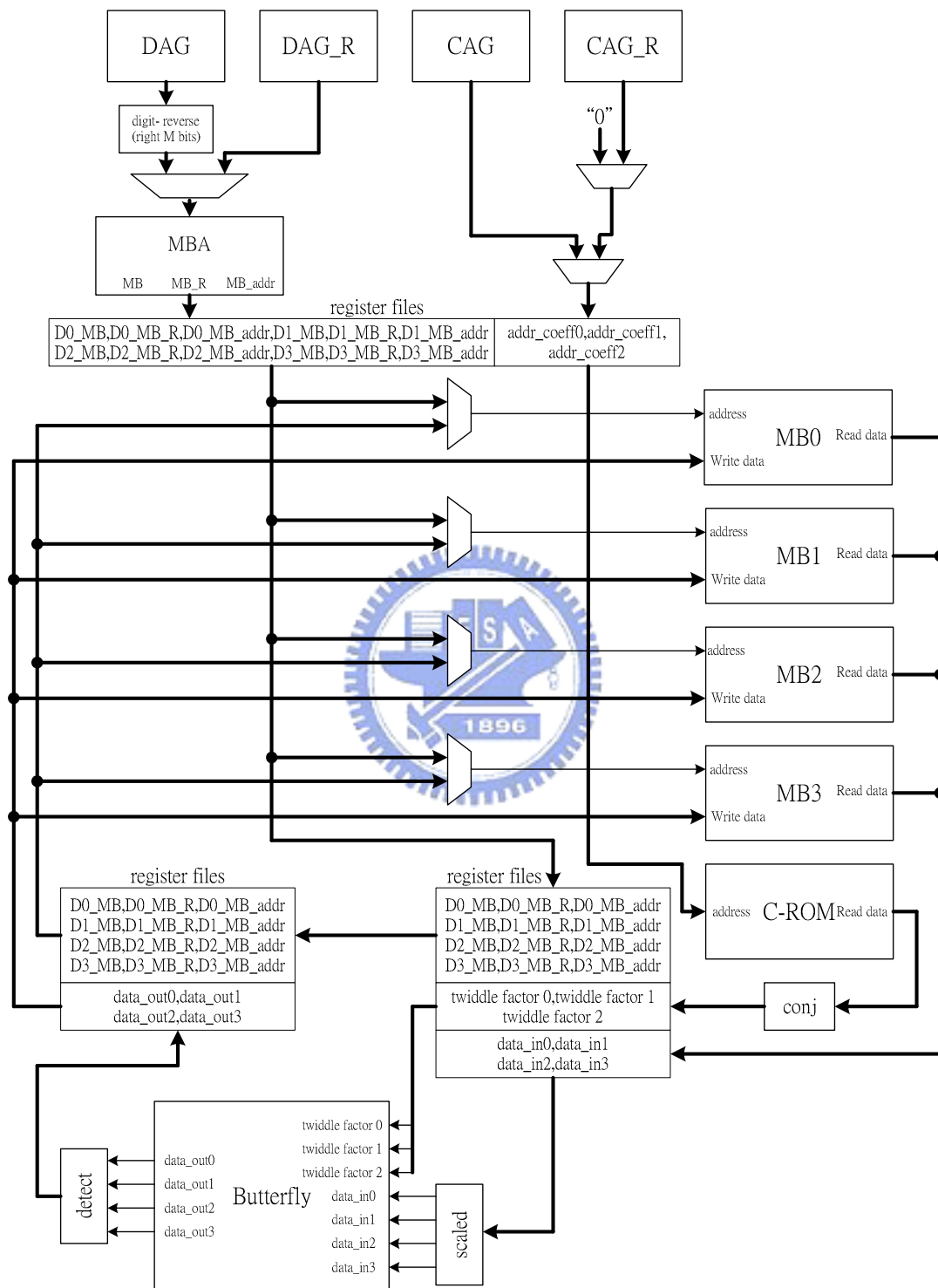


圖 5.28 : Radix-4 多模可變長度 RFFT/HS-IFFT 的硬體架構圖

5.4 效能分析及硬體需求

本節將討論 memory-based 架構 RFFT/HS-IFFT 處理器的輸出訊號量化雜訊比 (signal to quantization noise ratio, SQNR)，並決定 memory-based 硬體架構的字元長度。此外，我們也將探討使用 BFP 的 block scaling 方式對 SQNR 的改善效果。最後以 memory-based 可變長度處理器的硬體需求作個結束。

圖(5.29)為計算 SQNR 的示意圖， $x_q[n]$ 為輸入序列、 $X_q[k]$ 為經過理想 FFT 所運算出的結果、 $\hat{X}_q[k]$ 為 fixed-point FFT 所運算出的結果、 $\Delta X_q[k]$ 則為理想 FFT 與 fixed-point FFT 結果之間的雜訊值，最後在由 $X_q[k]$ 與 $\Delta X_q[k]$ 來求得 SQNR，式(5.26)為 SQNR 的計算方式。

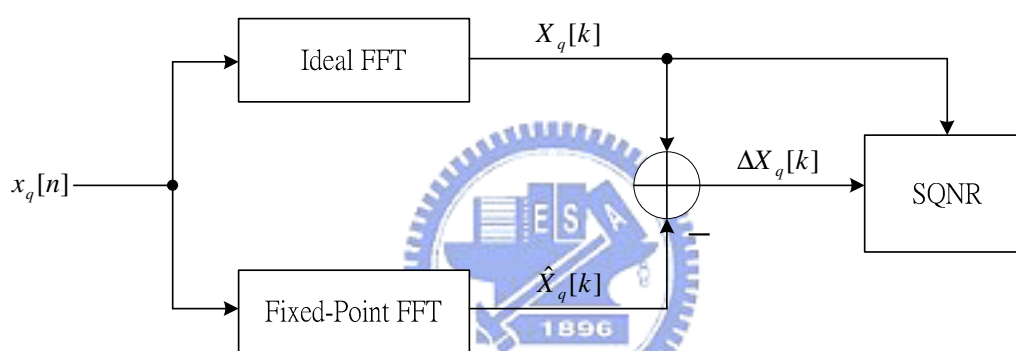


圖 5.29：計算 SQNR 的示意圖

$$SQNR = 10 \log \frac{\sum_{k=0}^{N-1} X_q^2[k]}{\sum_{k=0}^{N-1} (X_q[k] - \hat{X}_q[k])^2} \quad (5.26)$$

5.4.1 Radix-2 架構之 SQNR 分析

Input power 與 noise power 間的關係(輸入訊號為高斯分佈)

因為 memory-based 架構處理器其運算單元是採 fixed-point 模式運算，故在沒有做 block scaling 的狀況下，不管輸入訊號功率的大小其輸出端雜訊功率所能影響的值都是一樣的，如圖(5.30)(5.32)(5.34)所示。圖(5.30)(5.32)(5.34)分別為 radix-2 架構的 FFT/IFFT、RFFT、HS-IFFT 輸入訊號功率與其它功率相關圖，其中資料字元長度及 twiddle factor 長度為 $W = 24$ 、CFFT/CIFFT 處理器轉換點數大小為

4096(2^M , $M = 12$)。由圖(5.30)(5.32)(5.34)可知道，在沒有 block scaling 狀況下，輸出端的雜訊功率是不會受到輸入訊號功率的大小所影響。但在有 block scaling 狀況下，因其為動態的 scaling 方式，所以我們可以發現當輸入訊號功率大時，block scaling 可做的 scaling 範圍不大，當輸入訊號功率漸漸變小時，block scaling 可 scaling 的範圍也漸漸變大，故雜訊功率會受輸入訊號功率大小所做的不同動態 scaling 範圍而有所影響，且當輸入功率小於某值之後其雜訊功率將會是定值。比較有 block scaling 與無 block scaling 的雜訊功率後，可以發現有明顯的改善，尤其在輸入訊號功率較小時可改善將近 40dB 的值。另外，輸出訊號功率曲線是指理想 FFT 所運算出來的輸出結果功率，其功率大小會因輸入訊號功率變大而變大。輸出功率曲線與雜訊功率曲線相減可得 SQNR 曲線，如圖(5.31)(5.33)(5.35)所示，此 SQNR 曲線會隨著輸入功率的提高而得到較高的 SQNR 值。

字元長度與 noise power 間的關係(輸入訊號為高斯分佈)

圖(5.36)為 FFT/IFFT 不同字元長度下，轉換點數為 4096(2^M , $M = 12$)的輸入訊號功率與其它功率相關圖。由圖可知，不管字元長度及輸入功率大小為何，無 block scaling 的雜訊功率所能影響的值都是一樣的，而 block scaling 的雜訊功率則會因字元長度及輸入功率所導致的動態 scaling 而影響，RFFT、HS-IFFT 亦同。且因為字元長度侷限的關係，不同字元長度其所能提供輸出訊號功率有限，故圖(5.36)中所標示的虛線代表著各不同字元長度所能提供的輸出功率最大範圍。

字元長度的選擇(輸入訊號為高斯分佈)

而圖(5.37)(5.38)(5.39)分別為 FFT/IFFT、RFFT、HS-IFFT 在不同字元長度下，CFFT/CIFFT 處理器轉換點數為 4096(2^M , $M = 12$)且規定輸入訊號功率與字元長度的關係為 $input_power / 2^{word_length} = 0.15$ ，主要在於比較不同字元長度所能獲得的 SQNR 大小，且 block scaling 可得到較高的 SQNR。

所以為了有較高的 SQNR，我們選擇 block scaling 的方式來實現硬體，並且考慮到硬體的成木，我們要求 SQNR 至少得高於 60dB。由圖(5.37)(5.38)(5.39)可知，字元長度為 16 位元即可達到 SQNR=60dB 的要求，故我們將以字元長度為 16 位元來設計處理器硬體。

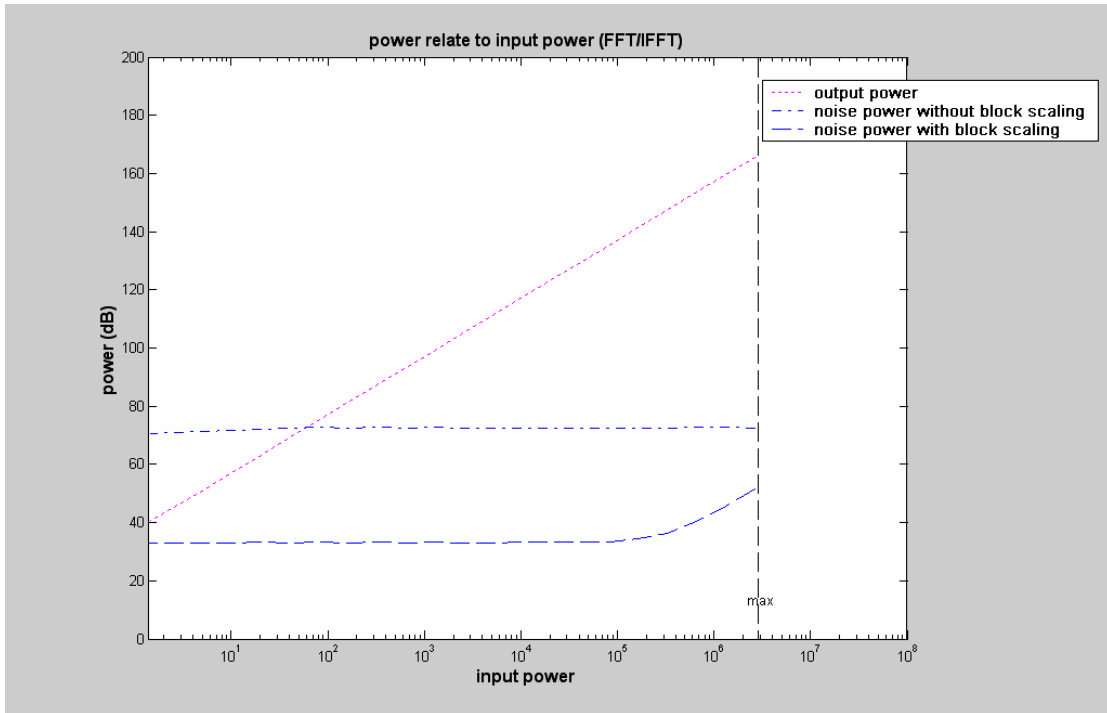


圖 5.30：FFT/IFFT 輸入訊號功率與其它功率相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

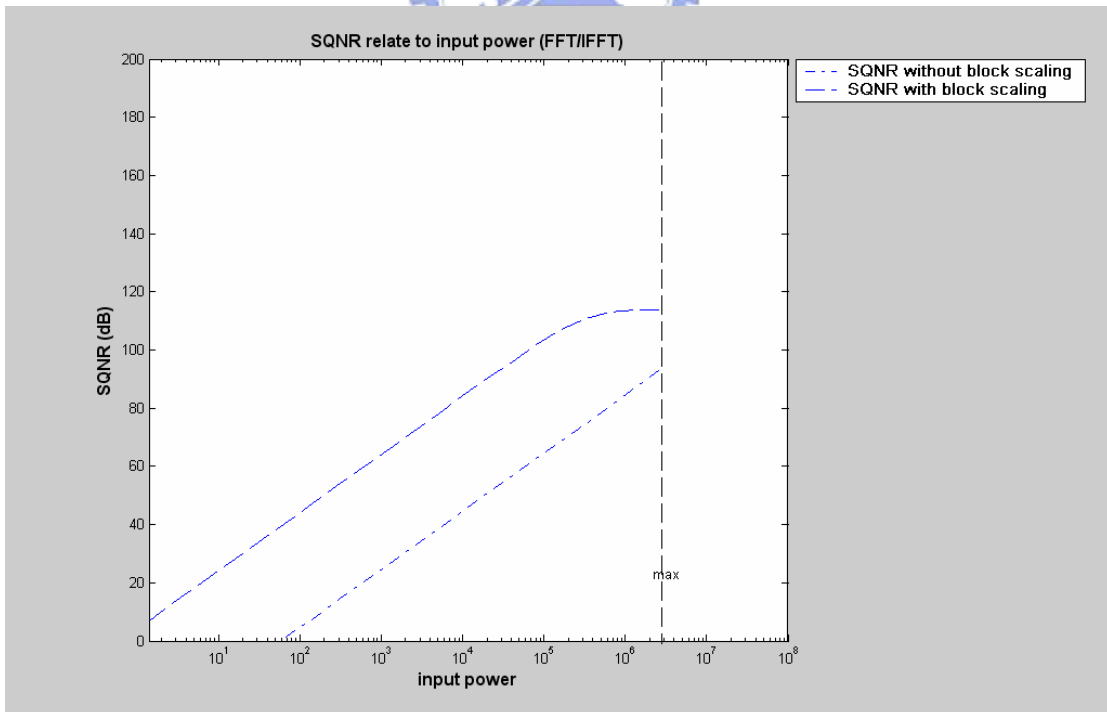


圖 5.31：FFT/IFFT 輸入訊號功率與 SQNR 相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

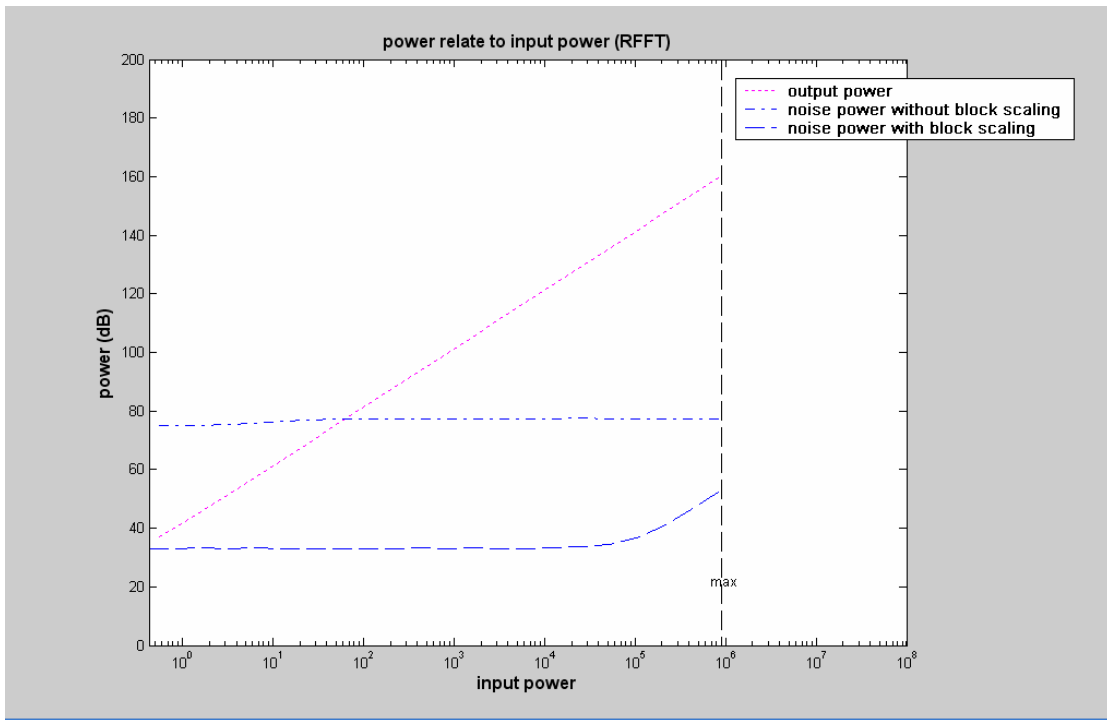


圖 5.32：RFFT 輸入訊號功率與其它功率相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

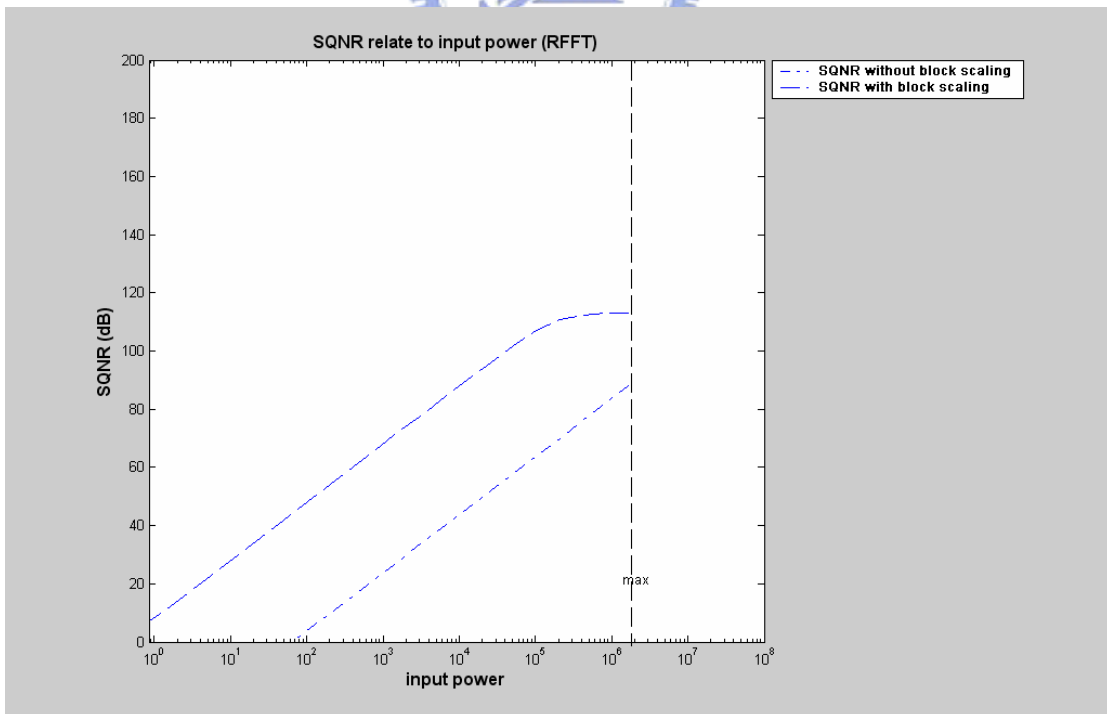


圖 5.33：RFFT 輸入訊號功率與 SQNR 相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

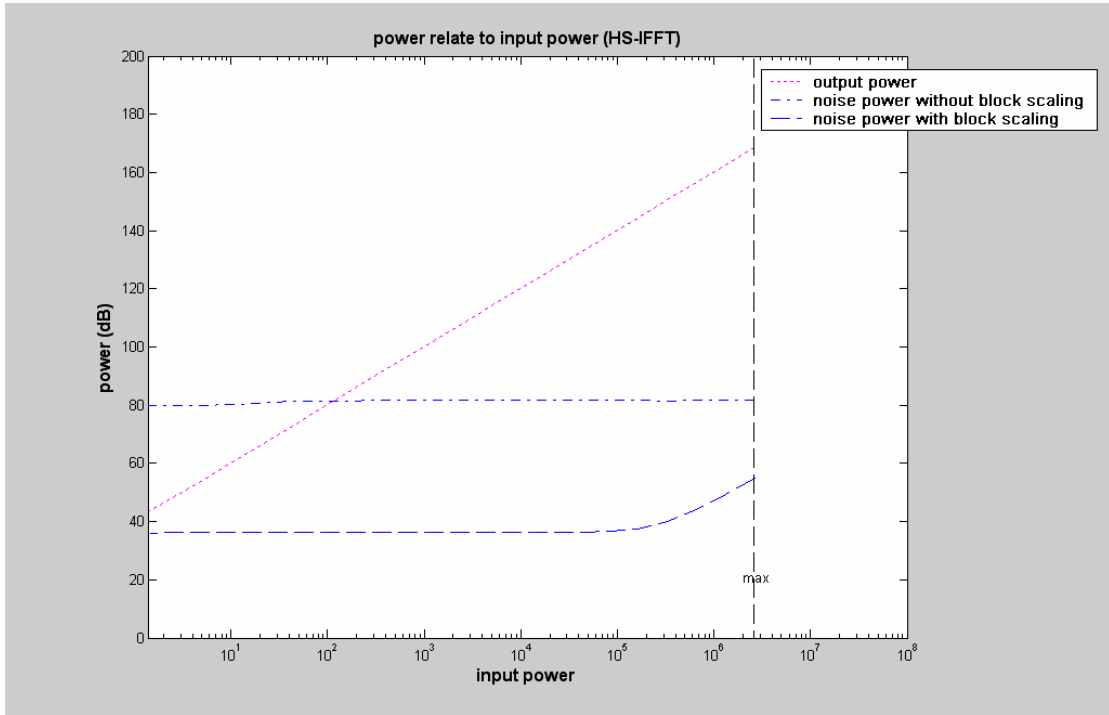


圖 5.34：HS-IFFT 輸入訊號功率與其它功率相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

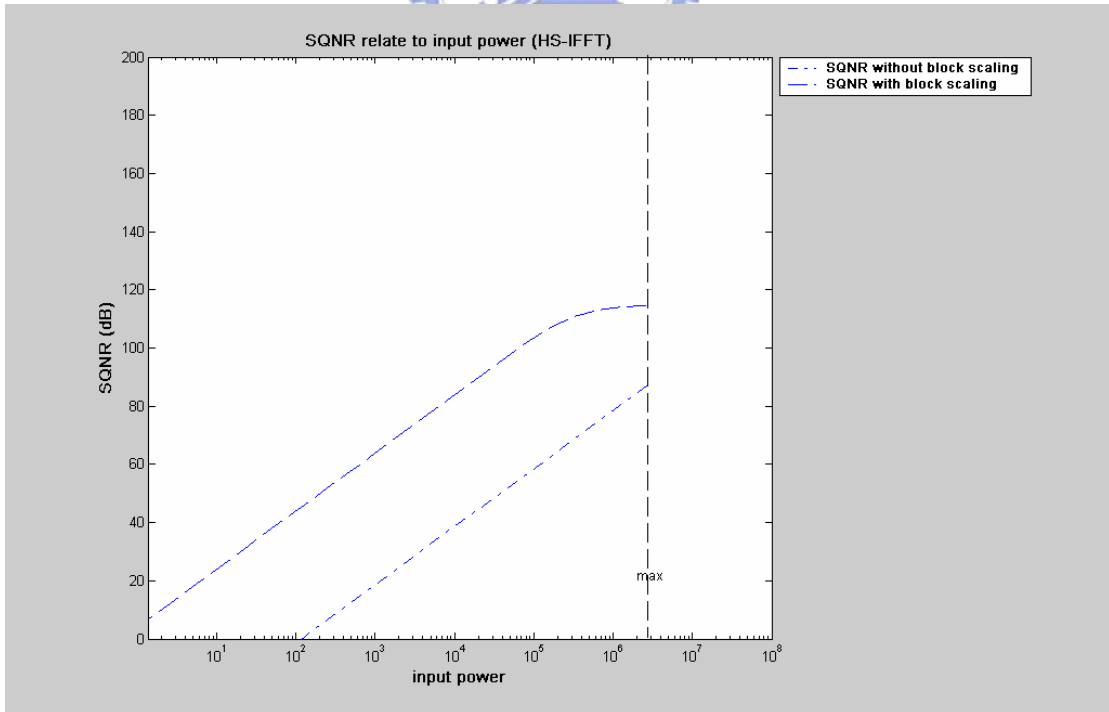


圖 5.35：HS-IFFT 輸入訊號功率與 SQNR 相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

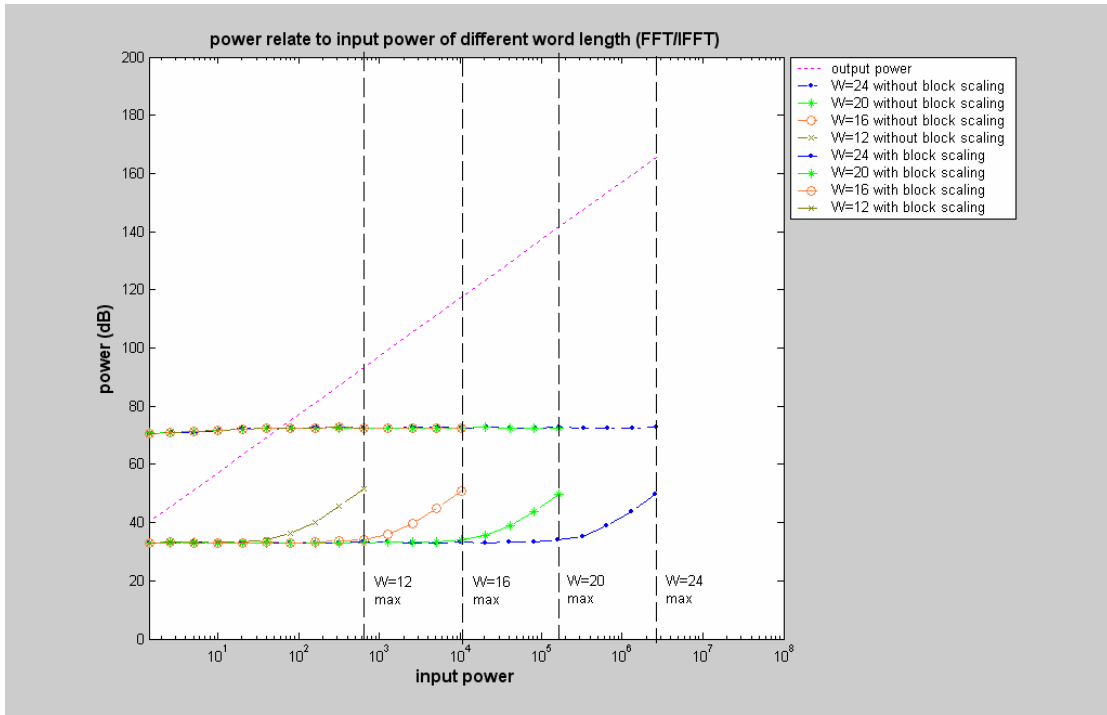


圖 5.36：不同字元長度下，FFT/IFFT 輸入訊號功率與其它功率相關圖
(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

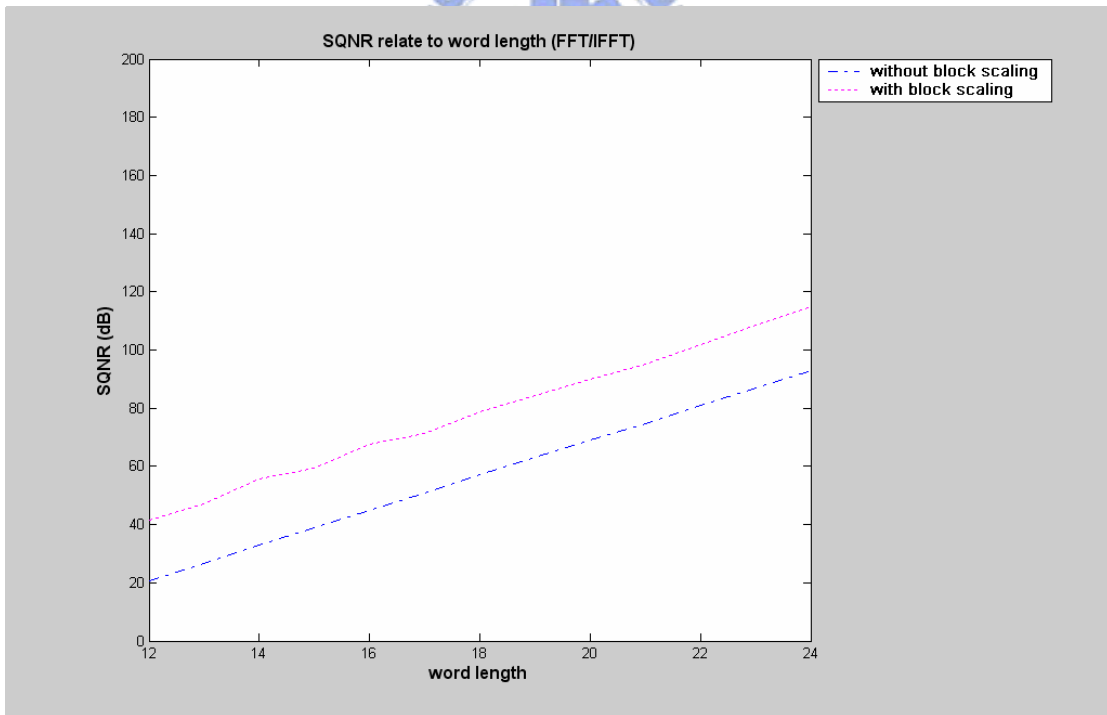


圖 5.37：FFT/IFFT 不同字元長度的 SQNR 圖
(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為 $input_power / 2^{word_length} = 0.15$ ，輸入訊號為高斯分布)

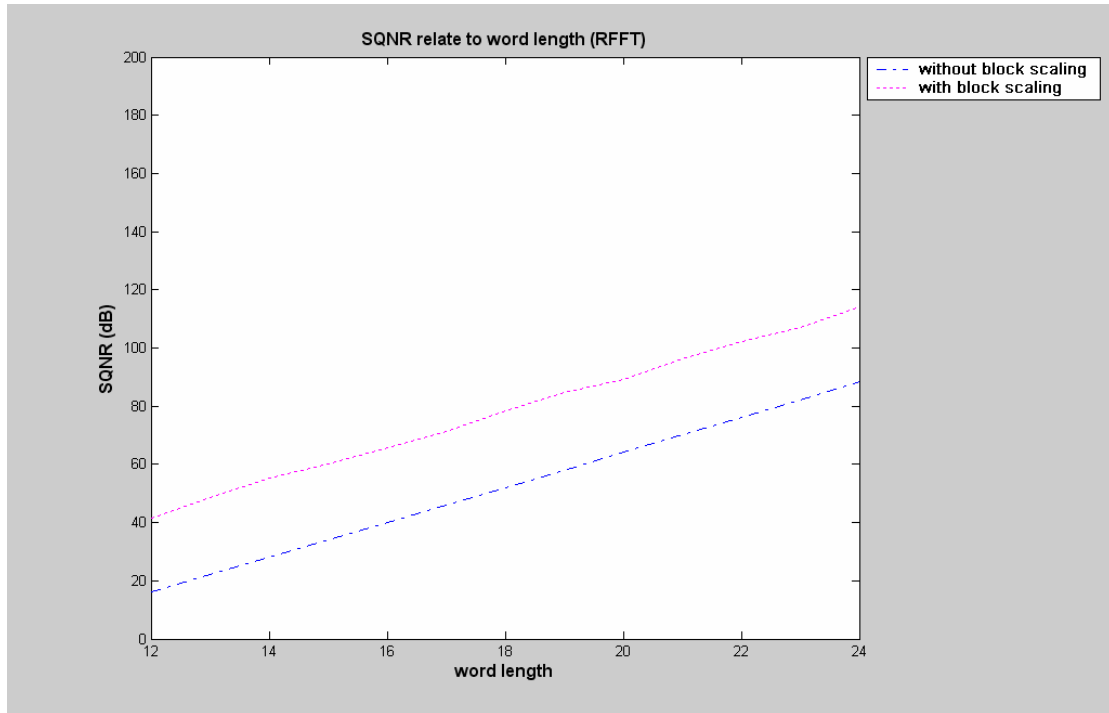


圖 5.38：RFFT 不同字元長度的 SQNR 圖

(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為

$$input_power / 2^{word_length} = 0.15, \text{ 輸入訊號為高斯分布})$$

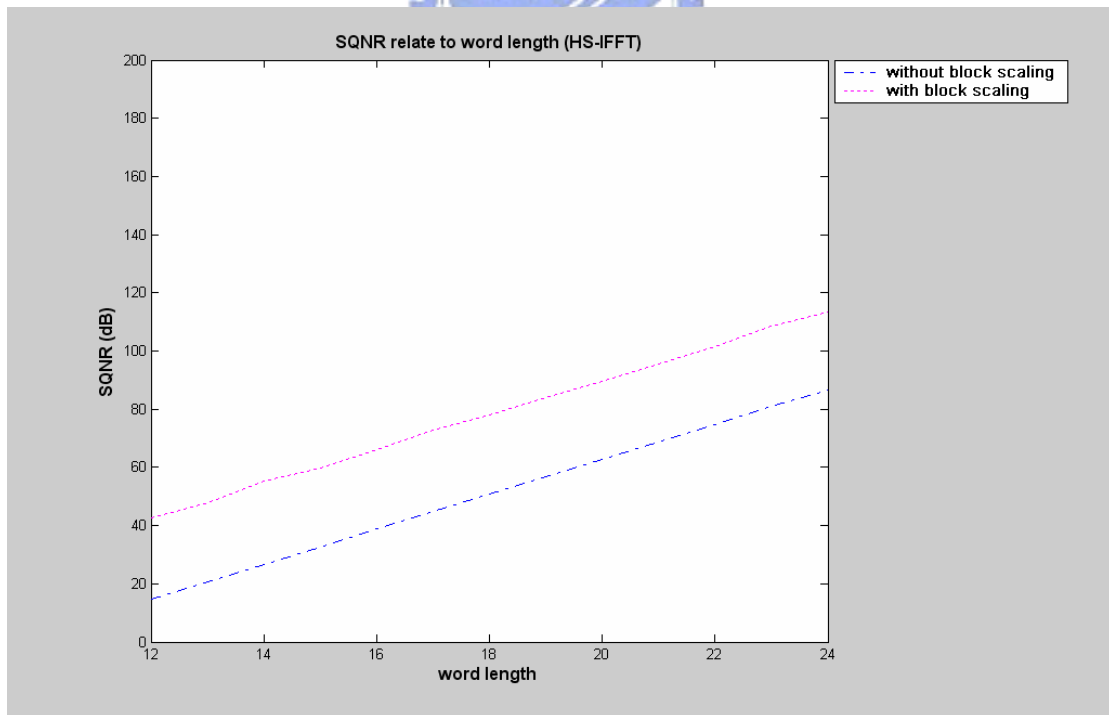


圖 5.39：HS-IFFT 不同字元長度的 SQNR 圖

(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為

$$input_power / 2^{word_length} = 0.15, \text{ 輸入訊號為高斯分布})$$

5.4.2 Radix-4 架構之 SQNR 分析

input power 與 noise power 間的關係(輸入訊號為高斯分佈)

同樣地在 radix-4 架構裡，我們也將比照 radix-2 架構的 SQNR 分析。圖(5.40)為 radix-4 架構的輸入訊號功率與其它功率相關圖，其中資料字元長度及 twiddle factor 長度為 $W=24$ 、CFFT/CIFFT 處理器轉換點數大小為 $4096(2^M, M=12)$ 。由圖可知無 block scaling 的雜訊功率與輸入訊號功率無關，但有 block scaling 的雜訊功率會因動態 scaling 的關係而受輸入訊號功率大小的影響。圖(5.41)為輸出功率曲線與雜訊功率曲線相減所得到的 SQNR 曲線圖，且輸入功率越大所得到的 SQNR 越高。

字元長度與 noise power 間的關係(輸入訊號為高斯分佈)

圖(5.42)為不同字元長度下，CFFT/CIFFT 處理器轉換點數為 $4096(2^M, M=12)$ 的輸入訊號功率與其它功率相關圖。

字元長度的選擇(輸入訊號為均勻分佈)

圖(5.43)(5.44)(5.45)則分別為 FFT/IFFT、RFFT、HS-IFFT 在不同字元長度下，CFFT/IFFT 處理器轉換點數為 $4096(2^M, M=12)$ 且規定輸入訊號功率與字元長度的關係為 $input_power / 2^{word_length} = 0.15$ 的定值，所得到的字元長度與 SQNR 相關圖。

由圖(5.43)(5.44)(5.45)可知，達到 SQNR=60dB 以上的字元長度至少為 16 位元且為 block scaling 的方式，故 radix-4 處理器的字元長度為 16 位元。

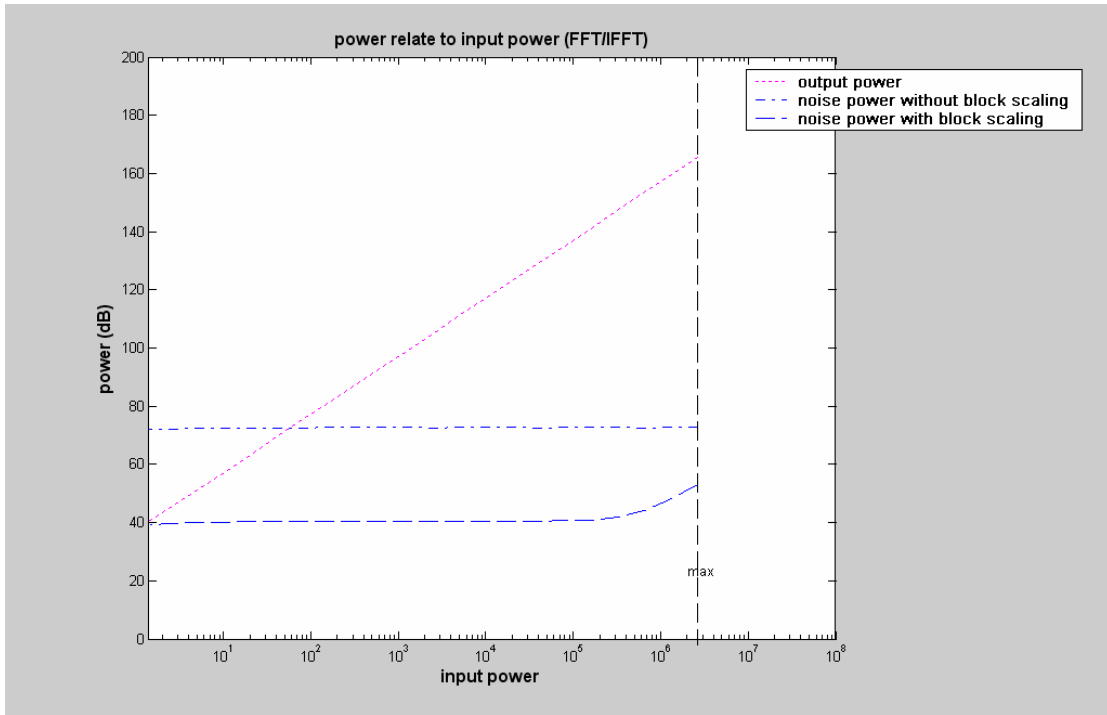


圖 5.40：FFT/IFFT 輸入訊號功率與其它功率相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

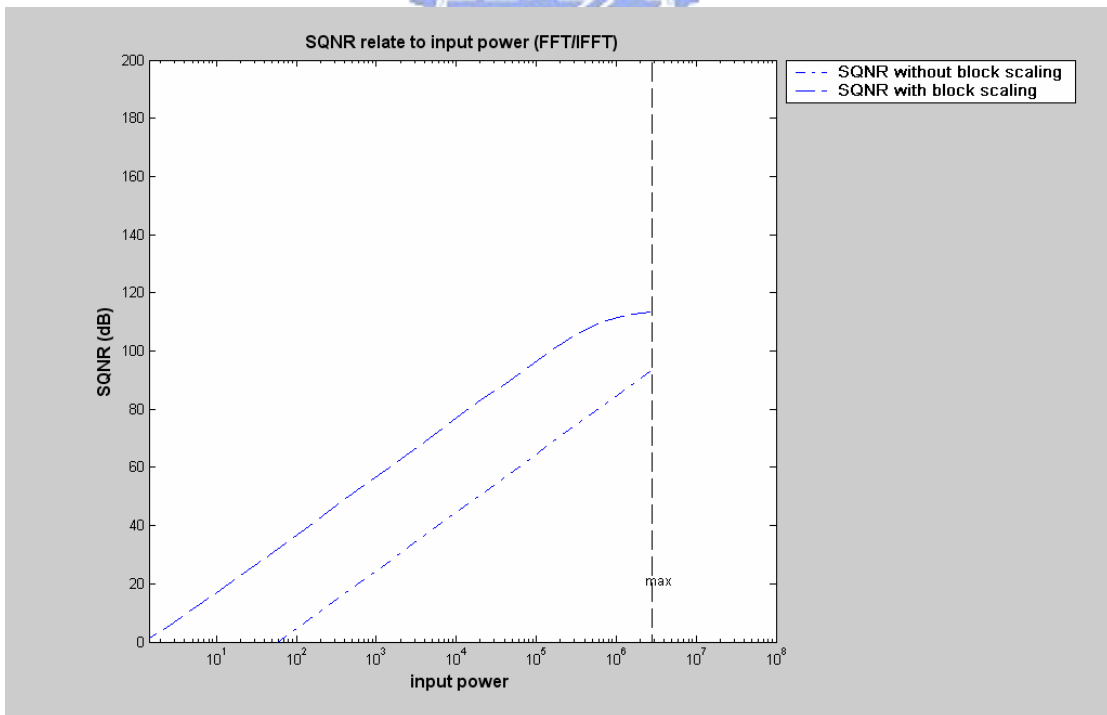


圖 5.41：FFT/IFFT 輸入訊號功率與 SQNR 相關圖

(字元長度 $W = 24$ 位元，CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

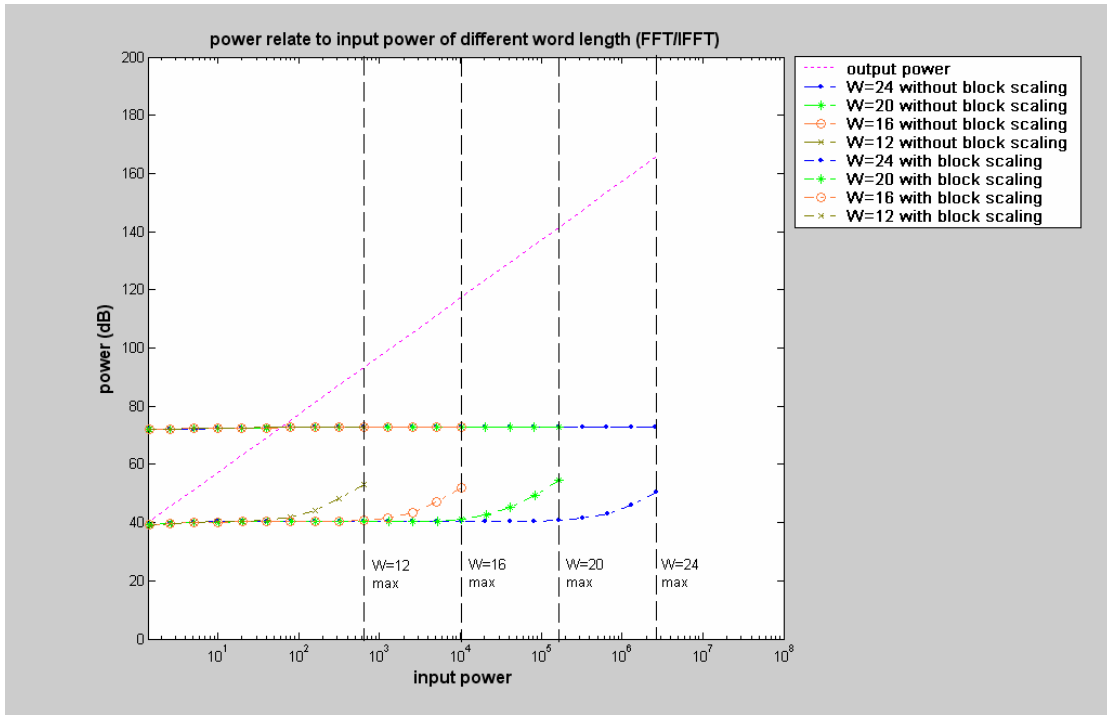


圖 5.42：不同字元長度下，FFT/IFFT 輸入訊號功率與其它功率相關圖
(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號為高斯分布)

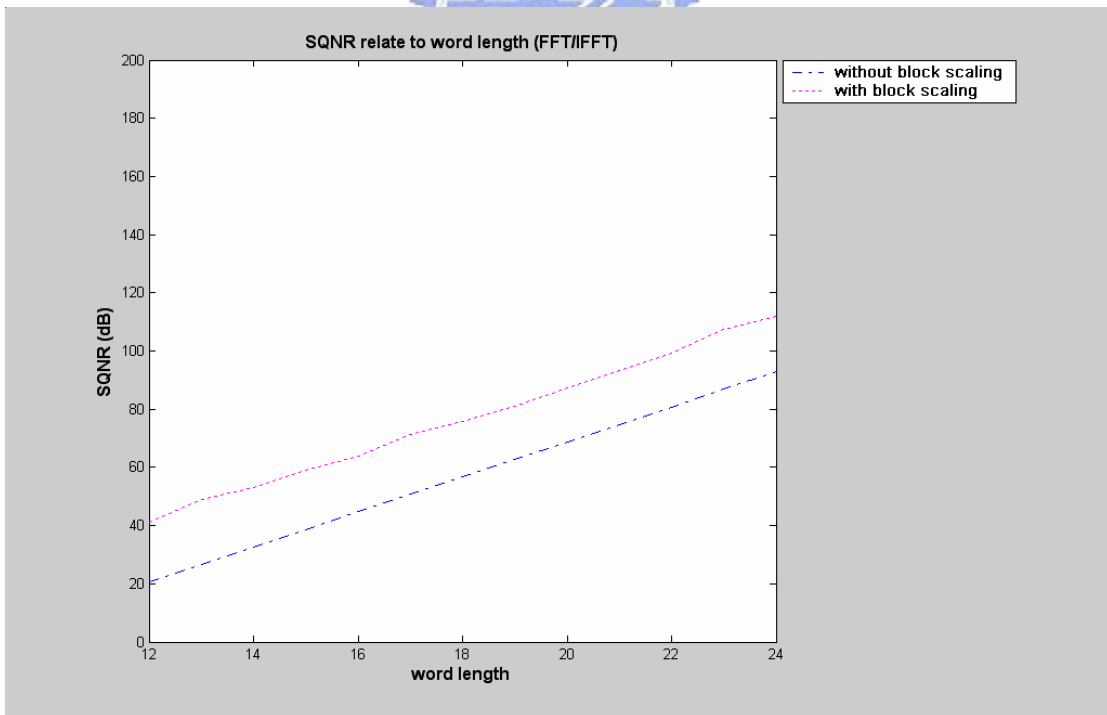


圖 5.43：FFT/IFFT 不同字元長度的 SQNR 圖
(CFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為
 $input_power / 2^{word_length} = 0.15$ ，輸入訊號為高斯分布)

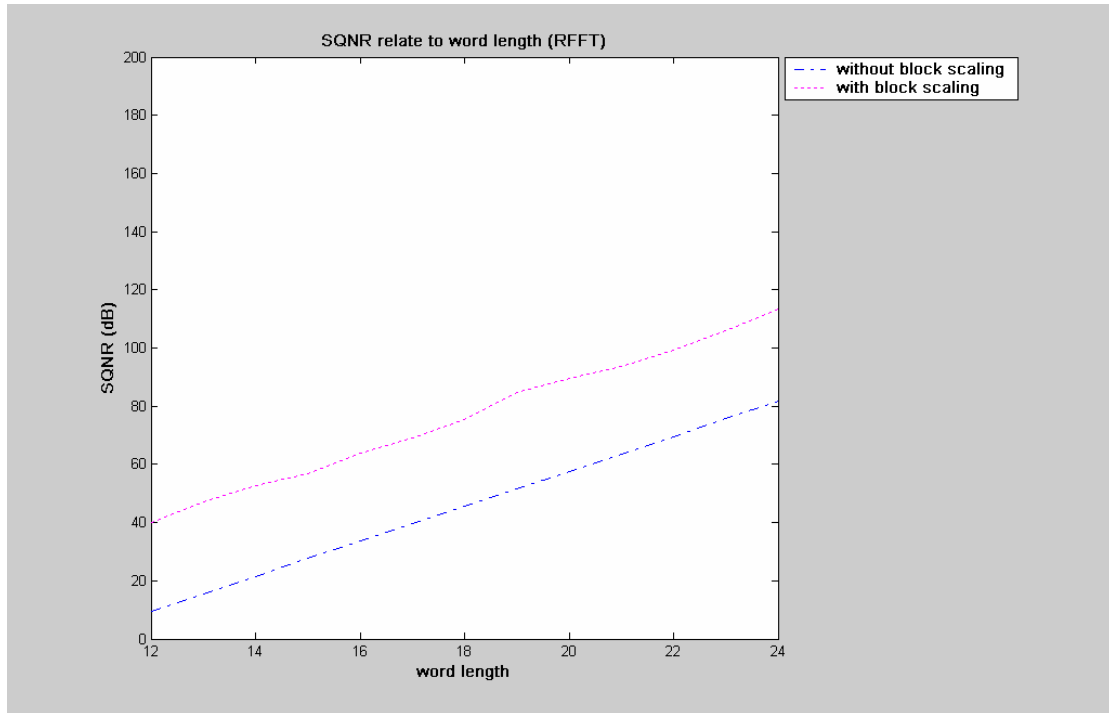


圖 5.44：RFFT 不同字元長度的 SQNR 圖

(CFFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為

$$input_power / 2^{word_length} = 0.15, \text{ 輸入訊號為高斯分布}$$

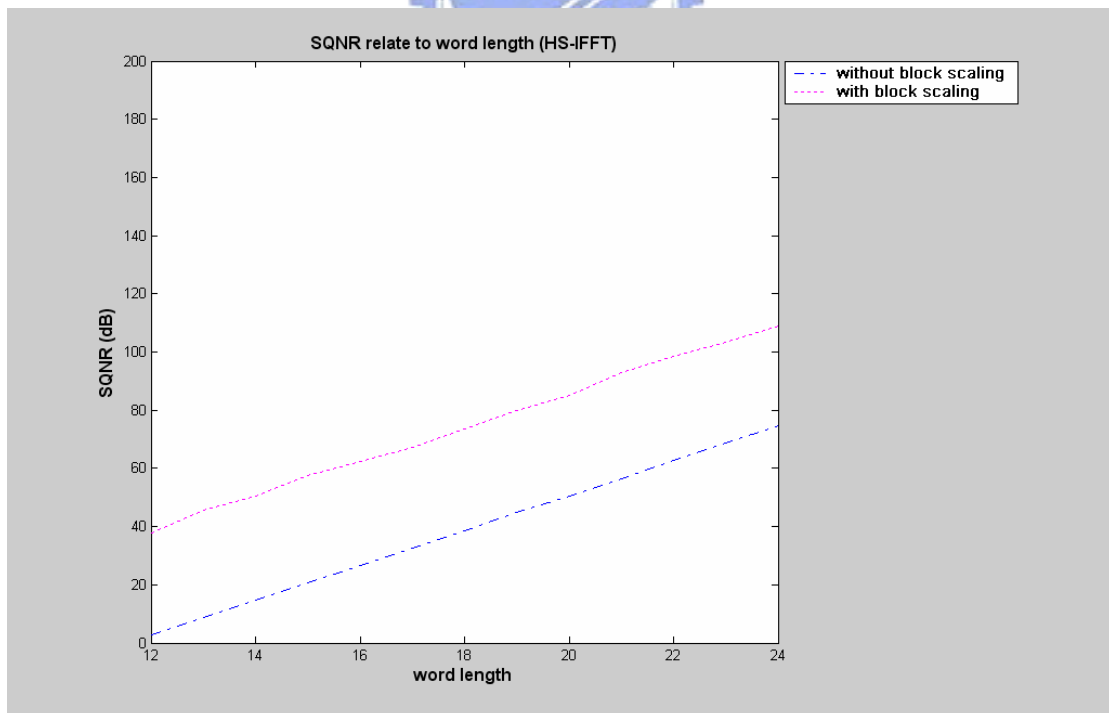


圖 5.45：HS-IFFT 不同字元長度的 SQNR 圖


(CFFFT/CIFFT 處理器轉換點數為 4096，輸入訊號功率與字元長度的關係為

$$input_power / 2^{word_length} = 0.15, \text{ 輸入訊號為高斯分布}$$

5.4.3 記憶體大小需求與運算所需 Clock Cycles 數

因為 memory-based 架構多模可變長度處理器的運算單元數目是固定的，也就是說不會隨著轉換點數的大小而有所增減，所以當字元長度給定之後，其運算單元的硬體大小也就同時決定了。但記憶體大小卻跟可變長度的最大範圍有關，因為記憶體至少得提供最大範圍所需大小才行，且也與所選擇的字元長度有關。表(5.3)為字元長度 16 位元的最大可變長度範圍所需記憶體大小，又因為 radix-r butterfly 運算單元欲同時讀(寫)所需運算元，故得將記憶體分成 r 個區塊來達到同時讀(寫)的目的。

之前有提到過，因為記憶體為 singal port 模式讀寫不可能同時發生，故 radix-r butterfly 處理器將有兩個 clock cycles 的時間來計算一次的 radix-r butterfly 運算，依據轉換點數 N 及多模處理器模式的選擇，我們可知全部轉換所需 radix-r butterfly 的運算次數，進而得知全部轉換所需的 clock cycles 數，表(5.4)為各不同轉換點數與運算模式所需的 clock cycles 數。表(5.5)為各不同通訊系統所需 FFT 處理器轉換大小和資料取樣速率。



Radix-2 FFT/IFFT $2^{M-1} \times 32$		Radix-2 RFFT/HS-IFFT $2^{M-2} \times 32$		Radix-4 FFT/IFFT $4^{M-1} \times 32$		Radix-4 RFFT/HS-IFFT $4^{M-1} / 2 \times 32$	
points(2^M)	Mem-size (bits)	points(2^M)	Mem-size (bits)	points(4^M)	Mem-size (bits)	points(4^M)	Mem-size (bits)
8192		8192	2Kx32 x2	8192		8192	1Kx32 x4
4096	2Kx32 x2	4096	1Kx32 x2	4096	1Kx32 x4	4096	
2048	1Kx32 x2	2048	512x32 x2	2048		2048	256x32 x4
1024	512x32 x2	1024	256x32 x2	1024	256x32 x4	1024	
512	256x32 x2	512	128x32 x2	512		512	64x32 x4
256	128x32 x2	256	64x32 x2	256	64x32 x4	256	
128	64x32 x2	128	32x32 x2	128		128	16x32 x4
64	32x32 x2	64	16x32 x2	64	16x32 x4	64	

表 5.3：字元長度 16 位元的最大可變長度範圍所需記憶體大小

Radix-2 FFT/IFFT $M \cdot 2^M$		Radix-2 RFFT/HS-IFFT $((M+1)/2) \cdot 2^M$		Radix-4 FFT/IFFT $M \cdot 4^M / 2$		Radix-4 RFFT/HS-IFFT $(M/2+1) \cdot 4^M$	
points(2^M)	cycles	points(2^M)	cycles	points(4^M)	cycles	points(4^M)	cycles
8192		8192	57344	8192		8192	16384
4096	49152	4096	26624	4096	12288	4096	
2048	22528	2048	12288	2048		2048	3854
1024	10240	1024	5632	1024	2560	1024	
512	4608	512	2560	512		512	768
256	2048	256	1152	256	512	256	
128	896	128	512	128		128	160
64	386	64	224	64	96	64	

表 5.4：各不同轉換點數與運算模式所需的 clock cycles 數

Communication System	FFT Size	Sampling Rate	Radix-2 processor operation frequency	Radix-4 processor operation frequency
802.11a	64	20 MHz	120.7 MHz	30 MHz
DAB	2048	2 MHz	22 MHz	
	1024	2 MHz	20 MHz	5 MHz
	512	2 MHz	18 MHz	
	256	2 MHz	16 MHz	4 MHz
DVB-T	8192	8 MHz	104 MHz	
	2048	8 MHz	96 MHz	
ADSL	512	2.2 MHz	11 MHz	3.3 MHz
VDSL	8192	34.5 MHz	241.5 MHz	69 MHz
	4096	17.3 MHz	112.45 MHz	51.9 MHz
	2048	8.6 MHz	51.6 MHz	16.2 MHz
	1024	4.3 MHz	23.65 MHz	10.75 MHz
	512	2.2 MHz	11 MHz	3.3 MHz

表 5.5：各通訊系統所需 FFT 處理器轉換大小和資料取樣速率

式(5.27)為處理器對各不同系統所對應的操作頻率：

$$\text{operation frequency} = \frac{\text{sampling rate} \times \text{clock cycles}}{\text{points}} \quad (5.27)$$

以 VDSL 為例，其轉換為 8192 點的實數序列轉換，經由上式運算所得的 radix-4 處理器需操作在 69MHz。

5.5 硬體實作與量測結果

決定完字元長度與最大可變長度範圍之後(字元長度為 16 位元，最大可變範圍的實數轉換為 8192 點)，我們就可以根據所設計好的架構動手撰寫 verilog 硬體描述語言，且因為所根據的架構一樣，故由 verilog 硬體描述語言所得之輸出結果必與 C 語言模擬結果一致。

本節的硬體實作將用 ASIC flow 與 FPGA flow 兩方式來驗證。其中關於 ASIC flow 方面，因為我們只想知道其硬體合成完後的邏輯閘數目(gate counts)，故我們只執行到 Design Compiler 這合成步驟及 gate simulation，關於後面的 Place & Route 等後端的處理就不執行了。至於 FPGA flow，我們將利用 QuartusII 軟體模擬所寫好的 RTL code，再下載至 FPGA 模擬板上做硬體驗證。

ASIC flow

因為記憶體大小會隨著可變長度的最大範圍而有所不同，故我們在撰寫 verilog 硬體描述語言時，會希望把記憶體獨立出來撰寫，然後 top module 再將其包含進去，這樣我們可以很明確的知道不含記憶體在內的邏輯閘數目(gate counts)，如下圖(5.46)所示，其中 core.v 為不包含記憶體的 RTL code，memory.v 則為所有記憶體的 RTL code。下表(5.6)為用 design complie 在 $0.25\mu\text{m}$ 製程環境下 clock constraint 設為 100MHz 合成後所得到的 core.v 邏輯閘數目(gate counts)。

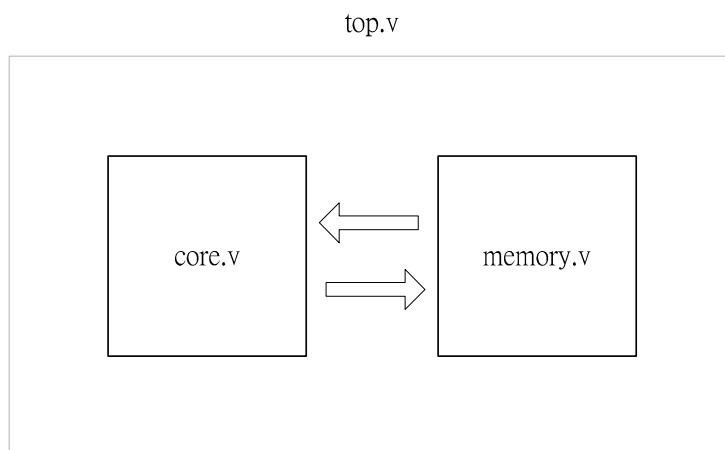


圖 5.46：RTL code 階層圖

Processor core.v	Radix-2 RFFT/HS-IFFT	Radix-4 RFFT/HS-IFFT
Gate Counts	27,362	79,123

表 5.6：多模可變長度 RFFT/HS-IFFT 處理器不含記憶體邏輯閘數目
(字元長度為 16 位元，最大可變範圍的實數轉換為 8192 點，
0.25 μ m 製程，clock constraint 為 100MHz)

FPGA flow

將寫好的 veilog 硬體描述語言，經由 QuartusII 軟體模擬編譯後，再下載至 FPGA 模擬板上並做硬體驗證，以下為驗證環境及驗證方式。表(5.7)為下載至 FGPA 模擬板上的 radix-2 與 raidix-4 多模可變長度 RFFT/HS-IFFT 處理器的最大操作頻率，基本上在 FPGA 上所得到的最大操作頻率會比在 ASIC 上差，因其 FPGA 內部的佈局並不是最佳狀態，故會有比較嚴重的 wire delay 發生。所以一般來說在 ASIC 上所得到的操作頻率會是在 FPGA 上的兩倍以上。

驗證環境：

1. design software：Quartus II ver.2.2。
2. FPGA 模擬版：DSP development board, Stratix edition，
Stratix EP1S25F780 device。如圖(5.47)所示。

驗證方式：

1. 由 data generator 產生 clock signal 與輸入訊號，餵入 FPGA development board，並以 logic analyzer 測得輸出訊號，如圖 5.48(a) 所示。
2. 將輸入訊號存於 FPGA development board 的 ROM 內，並藉由板內的石英震盪器與 P L L 產生所需的 clock signal，最後同樣地由 logic analyzer 測得輸出訊號，如圖 5.48(b)所示。

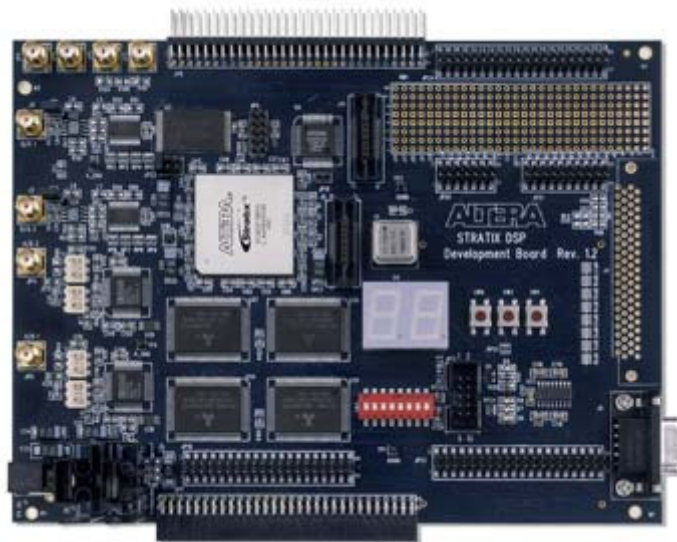


圖 5.47：DSP development board, Stratix edition

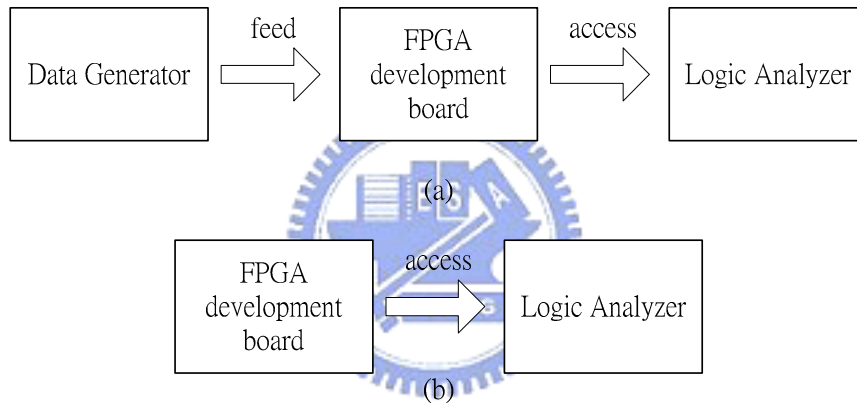


圖 5.48：驗證方式

Processor	Radix-2 RFFT/HS-IFFT	Radix-4 RFFT/HS-IFFT
Maximum Frequency	35MHz	32MHz

表 5.7：多模可變長度 RFFT/HS-IFFT 處理器的最大操作頻率
(字元長度為 16 位元，最大可變範圍的實數轉換為 8192 點)

5.6 結論

本章實現了之前所提到的以 memory-based 為基礎架構之多模可變長度 RFFT/HS-IFFT 處理器，並根據所需要的 SQNR 決定字元長度，且最後以 Design Compiler 合成估計出不包含記憶體之邏輯閘數目(gate count)，及下載至 FPGA 模擬板上驗證。從表(5.4)與表(5.5)中，可以知道對於不同通訊系統處理器相對應的操作頻率，例如轉換點數為 8192 點實數序列的 VDSL，其 radix-4 處理器得操作在 69MHz，雖然在 FPGA 模擬板上所能得到的 radix-4 處理器最大操作頻率只有 32MHz，但我們知道在 FPGA 上所得到的最大操作頻率會比在 ASIC 上差，因為 FPGA 內部的佈局並不是最佳狀態，故會有比較嚴重的 wire delay 發生。所以一般來說在 ASIC 上所得到的操作頻率會是在 FPGA 上的兩倍以上，故 69MHz 的操作頻率 radix-4 處理器是可以達到。若 radix-2、radix-4 處理器的 8192 點實數轉換要達到 SQNR 為 60dB 以上，則字元和 twiddle factor 長度至少需 16 位元，且經由 Design Compiler 在 0.25 μ m 製程、clock constraint 為 100MHz 環境下合成後，所得到不含記憶體在內的邏輯閘數目分別為 27.4K 與 79.1K。



第六章 結論與未來展望

6.1 結論

在本論文中，提出了一個可適用於中、低 throughput rate OFDM 系統(如 ADSL、VDSL)的 memory-based 多模可變長度 RFFT/HS-IFFT 處理器，此處理器不只可用於計算一般複數序列的 FFT 和 IFFT，還可針對實數序列 FFT 和 Hermitian Symmetric IFFT 做更有效率的處理，也就是說當轉換序列為實數(Hermitian Symmetric)序列時，我們只需利用一半長度的複數 FFT 加上後(前)端處理即可完成。且藉由 BFP 的 block scaling 方式來改善處理器的精確度。相較於 pipeline 處理器架構而言，memory-based 處理器架構因其 butterfly 運算單元不會隨著轉換點數的多寡而增減，故在大點數轉換時可達到非常低的硬體需求，且可在不增加 butterfly 運算單元的狀況下達到可變長度的設計。此外，因為所採用的記憶體為 single port 架構，讀寫無法同時發生，故 butterfly 運算器可利用兩個 clock cycles 的運算時間，將原本需要四個實數乘法器的複數乘法運算以兩個實數乘法器來實現。並且為了降低儲存 twiddle factors 的唯讀記憶體大小，我們可利用 twiddle factors 相互間的映射關係來減少所需儲存的數目，藉由此相互間的相關性可將 twiddle factor 儲存個數減至為 $N/8+1$ 。

最後，我們利用 FPGA 模擬板來驗證本論文所設計的 memory-based 架構多模可變長度 RFFT/HS-IFFT 處理器，並提供了 radix-2 和 radix-4 這兩種不同 radix 的實現方式。且要求此兩種不同 radix 處理器所能處理的最大可變長度為 8192 點的實數序列 FFT，在最大長度轉換下所得到的 SQNR 需大於 60dB，經由分析所需字元及 twiddle factor 長度至少要 16 位元。在相同點數相同模式轉換下，radix-4 處理器所需運算的 clock cycles 數約為 radix-2 處理器的四分之一，故可利用 high radix 的方式來設計出符合時間需求的處理器。但 high radix 方式的處理器所需硬體花費相對的也較高，我們可從 radix-2 與 radix-4 處理器經由 Design Compiler 在 $0.25\mu\text{m}$ 製程環境下合成之後，所得到的邏輯閘數目得知，其中 radix-2 與 radix-4 處理器經合成後，所得不含記憶體在內的邏輯閘數目分別為 27.4K 與 79.1K。

6.2 未來展望

本論文所提及到的多模可變長度 RFFT/HS-IFFT 處理器，因具有多模式切換 (FFT/IFFT/RFFT/HS-IFFT) 與動態可變長度等優點。除了可以應用於大多數中、低 throughput rate 的大點數轉換 OFDM 系統中，亦可應用於先進的通訊系統中，如結合了 OFDM 與 FDMA (frequency division multiple access) 兩大系統優點的 OFDMA 通訊系統，我們可利用動態調整轉換長度的優點並配合適當的處理方式，來針對特定傳輸載波做更有效的處理。此外，對於硬體面積及 SQNR 來說，此處理器都有不錯的表現。



參考資料

- [1] Salzberg, B. R., “Performance of an efficient parallel data transmission system”, *IEEE Trans. Commun.*, Vol. COM-15, pp. 805-811, Dec. 1967.
- [2] Weinstein, S. B., and P. M. Ebert, “Data Transmission by Frequency Division Multiplexing Using the Discrete Fourier Transform”, *IEEE Trans. Commun.*, Vol. COM-19, pp. 628-634, Oct. 1971.
- [3] Yeong-Terng Lin, “Design and Implementation of a Variable-Length FFT Processor for OFDM systems”, NTU, Master Thesis, June 2001.
- [4] A. V. Oppenheim R. W. Schaffer, “Discrete-Time Signal Processing”, Prentice-Hall Inc, 1999.
- [5] Shousheng He and Mats Torkelson, “A New Approach to Pipeline FFT Processor, Parallel Processing Symposium”, The 10th International, pp. 766-770, 1996.
- [6] Mark A. Richards, “On the Efficient Implementation of the Split-Radix FFT”.
- [7] Chao Kai Zhang, “Investigation and Design of FFT Core for OFDM Communication Systems”, NCTU, Master Thesis, June 2002.
- [8] Shousheng He and Mats Torkelson, “Designing Pipeline FFT Processor for OFDM (de)Modulation”, URSI International Symposium on Signals, Systems and Electronics, pp. 257-262, 1998.
- [9] Shousheng He and Mats Torkelson, “Design and Implementation of a 1024-point FFT Processor”, in Proc. IEEE custom Integrated Circuit Conference, 99, 131-134, 1998.
- [10] Yutai Ma, “An Effective Memory Addressing Scheme for FFT Processors”, *IEEE Transactions on Signal Processing*, Vol. 47 Issue:3, pp. 907-911, March 1999.

- [11] Yutai Ma and Lars Wanhammar, "A Hardware Efficient Control of Memory Addressing for High-Performance FFT Processors", IEEE Transactions on Signal Processing, Vol. 48 Issue:3, pp. 917-921, March 2000.
- [12] Thomas Lenart and Viktor Owall, "A Pipelined FFT Processor using Data Scaling with Reduced Memory Requirements", CCCD, Department of Electrosceince, Lund University, Box 118, SE-221 00 Lund, Sweden.
- [13] Henrik V. Sorensen, Douglas L. Jones, C. Sidney Burrus, and Michael T. Heideman, "On Computing the Discrete Hartley Transform", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-33, No. 4, October 1985.
- [14] Henrik V. Sorensen, Douglas L. Jones, C. Sidney Burrus, and Michael T. Heideman, "Real-Valued Fast Fourier Transform Algorithms", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-35, No. 6, June 1987.
- [15] B. Raja Sekar and K. M. M. Prabhu, "Radix-2 Decimation-in-Frequency Algorithm for the Computation of the Real-Valued FFT", IEEE Transactions on Signal Processing, Vol. 47, No. 4, April 1999.
- [16] Marc Davio, "Kronecker Products and Shuffle Algebra", IEEE Transactions on Computers, Vol. C-30, No. 2, February 1981.
- [17] Harold S. Stone, "Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers, February 1971.
- [18] M. Hasan and T. Arslam, "FFT Coefficient Memory Reduction Technique for OFDM Application".
- [19] L. G. Jonhson, "Conflict Free Memory Addressing for Dedicated FFT Hardware", IEEE Transactions on Circuit and System-II: Analog and Digital Signal Processing, Vol. 39 No. 5, pp. 312-316, May 1992.