行政院國家科學委員會專題研究計畫　期中進度報告

子計畫一:無線網路串流聲訊研究及聲視訊子系統整合

(1/2)

計畫類別： 整合型計畫
計畫編號： NSC92-2219-E-009-015-
執行期間： 92 年 08 月 01 日至 93 年 07 月 31 日
執行單位： 國立交通大學電子工程學系

計畫主持人： 杭學鳴

計畫參與人員： 楊政翰，陳繼大，曾建統，李仰哲，王盈閔，陳志楹，黃祥哲

報告類型： 完整報告

處理方式： 本計畫涉及專利或其他智慧財產權，1 年後可公開查詢

中　華　民　國 93 年 5 月 26 日

行政院國家科學委員會補助專題研究計畫 □成果報告　■期中進度報告

無線網路串流聲訊研究及聲視訊子系統整合 (1/2)
Wireless Streaming Audio Research and Audio/Video Subsystem Integration (1/2)

計畫類別：□ 個別型計畫　■ 整合型計畫

計畫編號： NSC 92-2219-E009-015

執行期間： 92 年 8 月 1 日至 93 年 7 月 31 日

計畫主持人：杭學鳴

計畫參與人員：楊政翰，陳繼大，曾建統，李仰哲，王盈閔，陳志楹，黃祥哲

成果報告類型(依經費核定清單規定繳交)：□精簡報告　■完整報告

本成果報告包括以下應繳交之附件：

□赴國外出差或研習心得報告一份

□赴大陸地區出差或研習心得報告一份

□出席國際學術會議心得報告及發表之論文各一份

□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢

□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立交通大學電子工程學系

中　華　民　國　93 年　5 月　15 日

行政院國家科學委員會專題研究計畫成果報告

無線網路串流聲訊研究及聲視訊子系統整合（1/2）

Wireless Streaming Audio Research and Audio/Video Subsystem Integration (1/2)

計畫編號: NSC 92-2219-E009-015
執行期限: 92 年 8 月 1 日至 93 年 7 月 31 日

主持人: 杭學鳴　國立交通大學電子工程學系教授
計畫參與人員：楊政翰，陳繼大，曾建統，李仰哲，王盈閔，陳志楹，黃祥哲

國立交通大學電子研究所

中文摘要

本子計畫之主要目標在研究與製作寬頻無線網路環境中的串流聲訊系統。本年度以 DSP 的實現為目標，其方向為音訊壓縮及錯誤控制編碼(Error Control Coding)的實現，並盡量增進其執行速度，以符合無線網路頻寬上的要求。音訊壓縮部分，本計畫採用 MPEG-4 AAC。由於霍夫曼解碼(Huffman decoding)及 IMDCT 這兩部分為速度上的瓶頸，為此，我們以各種快速演算法來加速這兩部分，並以 FPGA 實現其硬體，使得 IMDCT 部分加快了約二十倍。錯誤控制編碼部分，根據 IEEE 802.16a，將包含內外層編碼 –外層編碼為里德・所羅門編碼(Reed-Solomon Code)，而內層編碼為迴旋編碼(Convolutional Code)。為了增進其速度，我們找了數種 Finite Field 及 Viterbi 解碼器快速演算法，並利用 DSP 專門處理有限體指令，使得 DSP 在處理里德・所羅門編碼及迴旋編碼時，分別可達 18Mbps 及 6.2 Mbps 的速度。

關鍵詞：MPEG，音訊壓縮，AAC，IMDCT，霍夫曼解碼，里德・所羅門編碼，迴旋編碼

Abstract

The goal of this research project is to study, simulate and design effective streaming audio algorithms/systems transmitted in the wideband wireless environment. The objective of this year is to implement MPEG audio compression and 802.16a error control codes on DSP platforms. In addition to porting the c codes to the TI DSP processor, the challenge is to speed up the said audio compression and error control coding processes by modifying the algorithms and fine-tuning the programs. For audio compression, the MPEG-4 AAC codec is realized on the TI C6416 processor. Because the Huffman decoding and MDCT/IMDCT tools are the most computational intensive elements in an MPEG-4 AAC codec, several fast algorithms are adopted and modified to reduce the complexity of these two tools. To further increase the throughput, these two tools are also im-

plemented in FPGA. For example, a 20 times speed-up is demonstrated for executing the IMDCT operation. The error control codes defined by IEEE 802.16a consist of the RS code (Reed-Solomon Code) as the inner code and the convolutional code as the outer code. To increase the data processing rate of these two codes, several fast schemes for finite field operations and Viterbi algorithms have been studied and used. They are implemented on both DSP and FPGA. The results show that the data rates for the RS code and the convolutional code decoding are18 Mbps and 6.2 Mbps individually.

關鍵詞：AMR，Audio Compression，AAC，IMDCT，Huffman decoding，Reed-Solomon Code，

Convolutional Code

目錄 Table of Contents

A. Part 1: Audio Coding

A.1 背景

AAC 是 MPEG-2 中的音訊編碼標準，於 1997 制定完成。MPEG-2 AAC 聲訊編碼標準捨棄與 MPEG-1 聲訊編碼標準的相容性，加入了時域雜訊重整 Temporal Noise Shaping (TNS)及預測 Prediction 這兩個獨立的新模組，因此 AAC 能提供比 MP3 更好的壓縮率及聲訊品質[1] [2]。MPEG-4 AAC Version 2 是 ISO/IEC MPEG 於 1999 年制定完成之新一代聲訊編碼標準，架構如圖 A-1所示。MPEG-4 AAC 聲訊編碼是以 MPEG-2 AAC 為基礎，並加入了數個獨立的新模組，長期預測 Long Term Prediction (LTP)、感知雜訊替代 Perceptual Noise Substitution (PNS)、Transform-Domain Weighted Interleave Vector Quantization (Twin-VQ)等。這些新的模組將有助於更低位元率的聲訊壓縮。

本計畫將使用德州儀器 C6416 這顆 DSP 作為實現平台，此外並配合使用 Xilinx 的 Virtex-2 這顆 FPGA，對需大量運算的部分做加速。

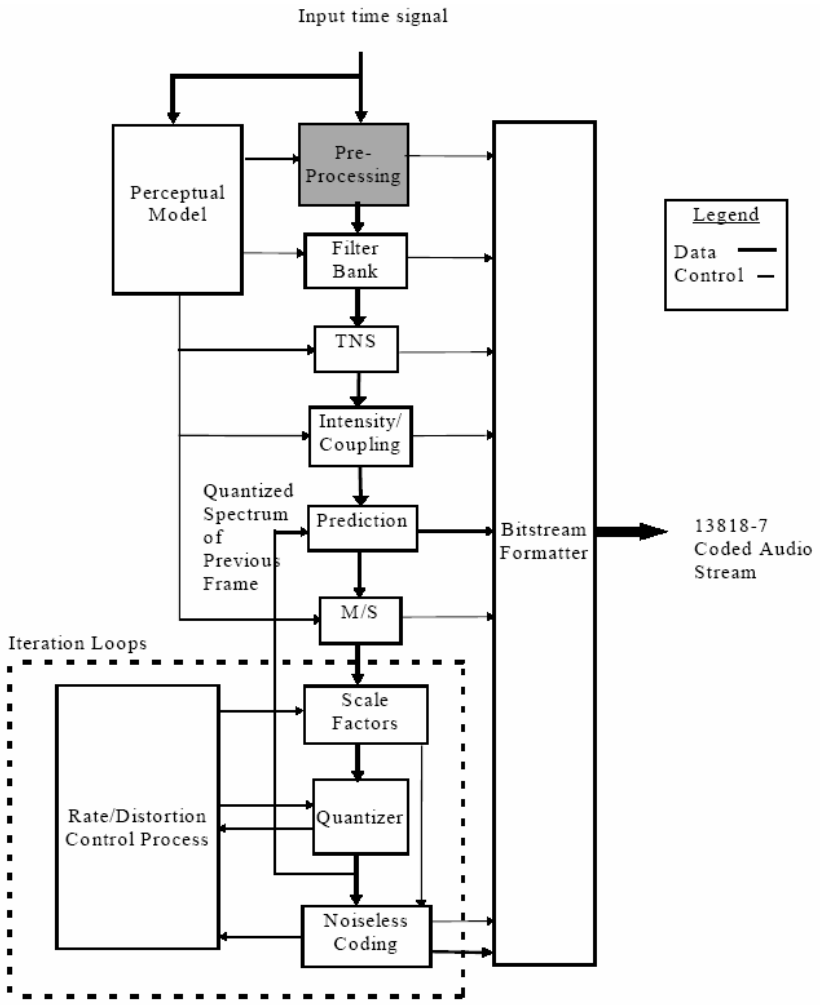


圖 A-1. MPEG-4 AAC 聲訊編碼整體架構

A.2 研究步驟

我們將 AAC 的編碼器和解碼器作了最佳化之後，放在 DSP 上執行。我們希望能夠進一步提高 AAC 解碼器的效能，因此統計 AAC 解碼程式在 DSP 上執行的時間，結果如圖 A-2 所示。

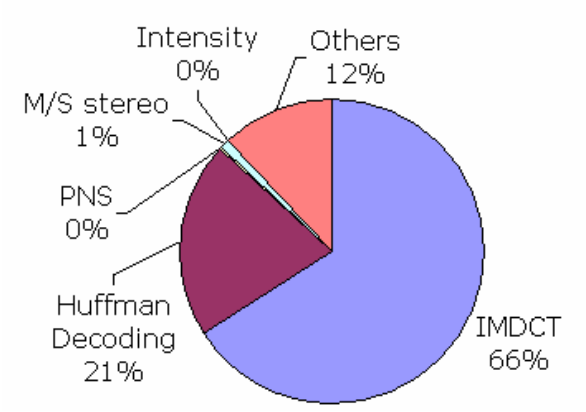


圖 A-2. 霍夫曼解碼方塊圖

我們根據圖 A-2的結果，發現反修正離散餘弦轉換 IMDCT 和霍夫曼解碼 Huffman Decoding 所需的時間分別佔了 66%以及 21%，不過由於處理器運算方式的限制，我們在 DSP 上所能作的改進有限，因此打算將這兩個部分移到 FPGA 來實現，期待 DSP/FPGA 的共同運作方式，能達到更高的效率。

A.2.1 霍夫曼解碼 Huffman Decoding：

首先我們先討論霍夫曼解碼的改進方式，AAC 解碼器對於霍夫曼解碼的流程如圖 A-3 所示。首先從位元流 bitstream 中，取出 1 到 19 位元長度不等的位元，以此來解出 0 到 120 的編碼索引 code index，然後根據側訊 side information 決定去那個霍夫曼編碼書 Huffman codebook 裡找出正確的碼字 codeword。

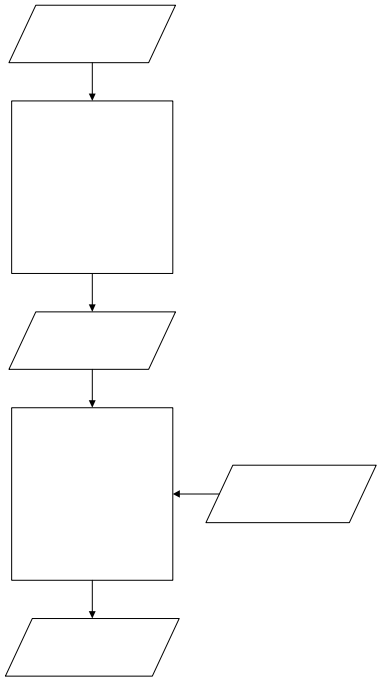


圖 A-3. 霍夫曼解碼方塊圖

A.2.1.1 變動輸入速率、固定輸出速率的實現方式:

這個動作對於處理器的架構來說，由於其無法確定要從位元流中取出多少的位元，所以只能一個時脈週期取出一個位元來比編碼索引書 Code index table 中是否有符合的位元串，這個動作變的浪費很多時間在抓一個位元，因此我們希望將其移到 FPGA 上來完成這個動作，達到每個時脈週期能固定輸出一個正確的碼字 codeword，這樣能大幅提昇其執行效率。因此我們作了以下的規劃，將編碼索引書 code index table 的部分放到 FPGA 上來實現，將固定長度的編碼索引 code index 傳回給 DSP 解出正確的碼字 codeword。

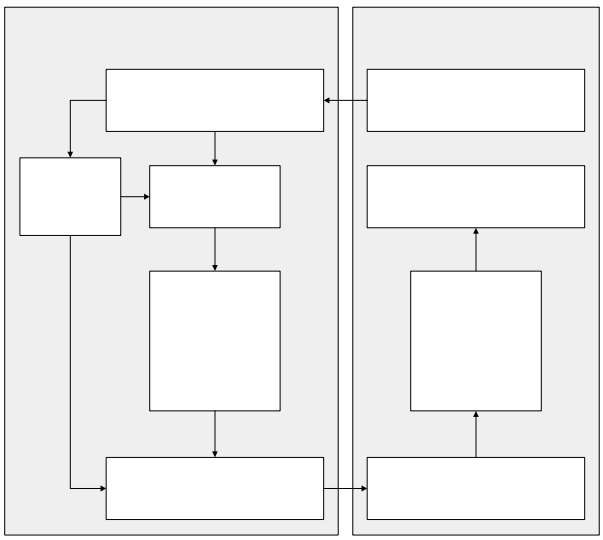


圖 A-4. 霍夫曼解碼的 DSP/FPGA 工作分配圖

A.2.1.2　變動輸入速率、變動輸出速率的實現方式:

根據[3],固定輸出速率的霍夫曼解碼會受限於編碼索引書 code index table 一次比對的數目,所以將解碼索引書 code index table 拆成多個小的區塊來作比對,這樣對較短的編碼索引 code index 可以較快的比對出來,但是對於較長的編碼索引則需要較多的時脈週期來完成,不過原則上較長的編碼索引出現的機會也較低,因此整體來說還是有比較大的機會能增加處理效率。

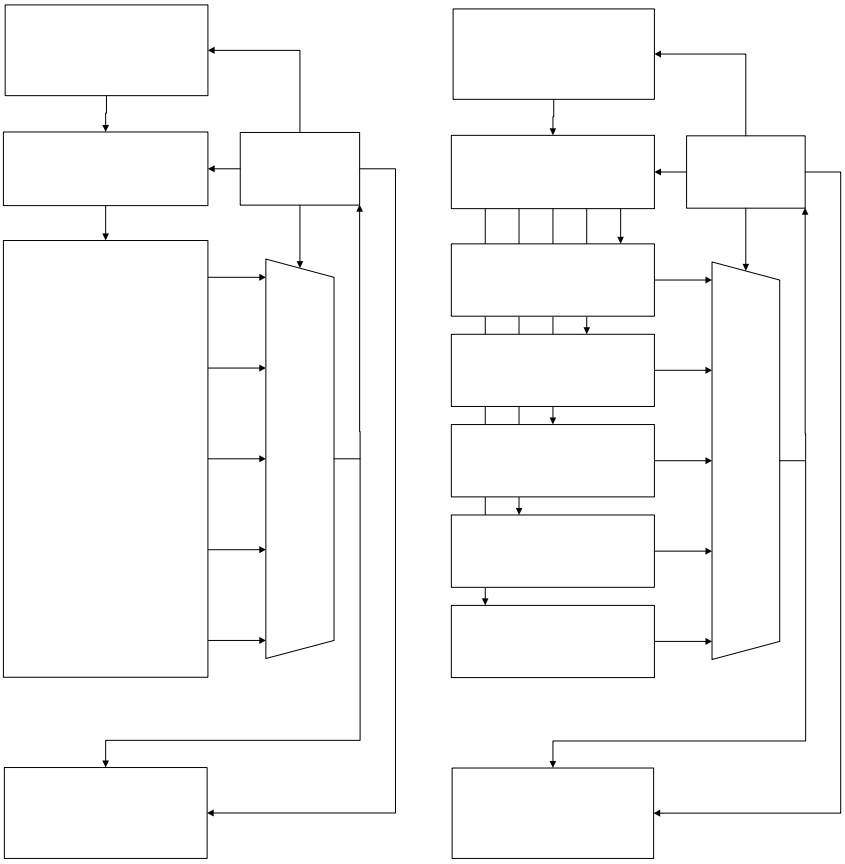


圖 A-5. 左邊為固定輸出速率的架構圖，右邊為變動輸出速率的架構圖

A.2.2 反修正餘弦轉換 IMDCT：

接下來討論反修正離散餘弦轉換 IMDCT，我們先看看 AAC 原來的要求，他要能對長窗 Long window 作 2048 點的 IMDCT，或是對短窗 short window 作 256 點的 IMDCT，我們根據[4]，IMDCT 的以用 N/4 點的快速傅立葉轉換 Fast Fourier Transform 來完成，其轉換關係如下。

Input Buffer

Barrel Shifter

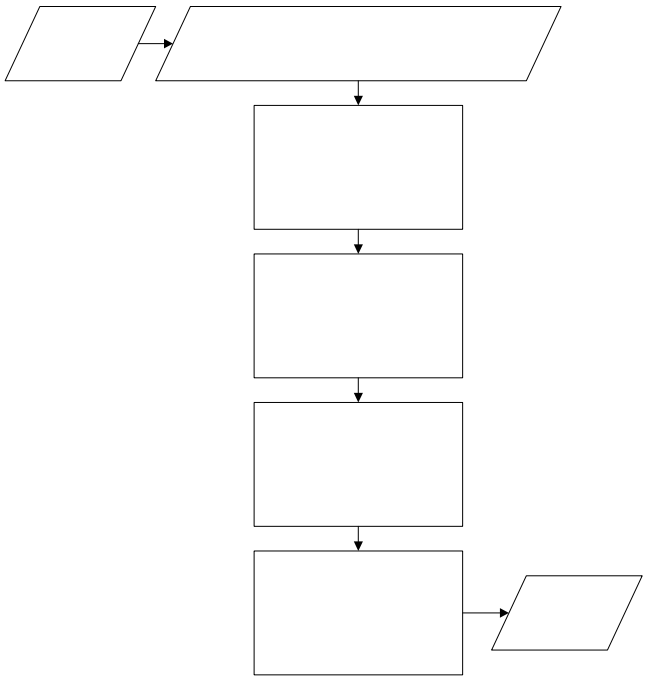


圖 A-6. IMDCT

除了在 DSP 上針對 DSP 的處理方式作最佳化之外，接下來我們打算把這個部分移到 FPGA 去作更大幅度的效能改進，我們根據上述的理論，希望能用 FFT 的架構來提高整體實現效能。我們參考[7]，表 A-1列出了常見的 FFT 實現架構，由於我們需要的是 512 點以及 64 點的反快速傅立葉轉換 IFFT，因此使用 Radix-2^3 用來做為我們在 FPGA 實現上的架構，Radix-2^3 架構和 Radix-2 比起來大約可以減少 2/3 的乘法器，其訊號圖又較 Split-radix 有規則性，更加適合實際電路上的實現。

	Number of Complex Multipliers	Number of Complex Adders	Number of Complex Registers
Radix-2 SDF	$\log_2 N-2$	$2\log_2 N$	N-1
Radix-4 SDF	$1/2\log_2 N-1$	$4\log_2 N$	N-1
Radix-8 SDF	$1/3\log_2 N-1$	$(8+2t/3)\log_2 N$	N-1
Radix-2^2 SDF	$1/2\log_2 N-1$	$2\log_2 N$	N-1
Radix-2^3 SDF	$1/3\log_2 N-1$	$(2+t/3)\log_2 N$	N-1
Radix-2 MDC	$\log_2 N-2$	$2\log_2 N$	1.5 N-2
Radix-4 MDC	$3/2\log_2 N-3$	$4\log_2 N$	2.5 N-4
Radix-8 MDC	$7/3\log_2 N-7$	$(8+2t/3)\log_2 N$	4.5 N-8
Radix-2^2 MDC	$\log_2 N-2$	$2\log_2 N$	1.5 N-2
Radix-2^3 MDC	$2/3\log_2 N-2$	$(2+t/3)\log_2 N$	1.5 N-2

表 A-1. 不同 FFT 架構所需的運算量之比較

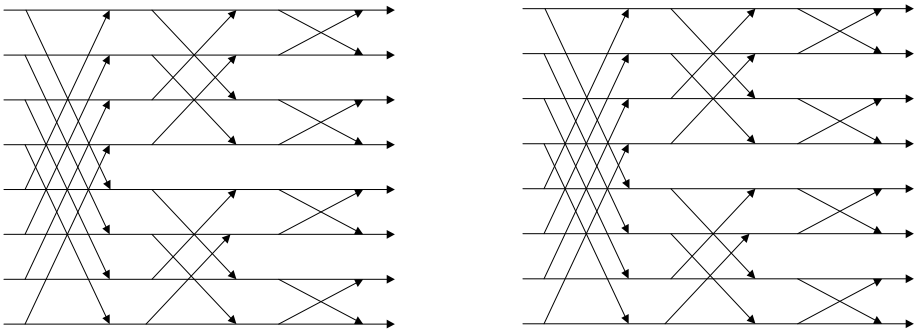


圖 A-7. 左邊為 8 點 Radix-2 的訊號圖，右邊 8 點為 Radix-2^3 的訊號圖

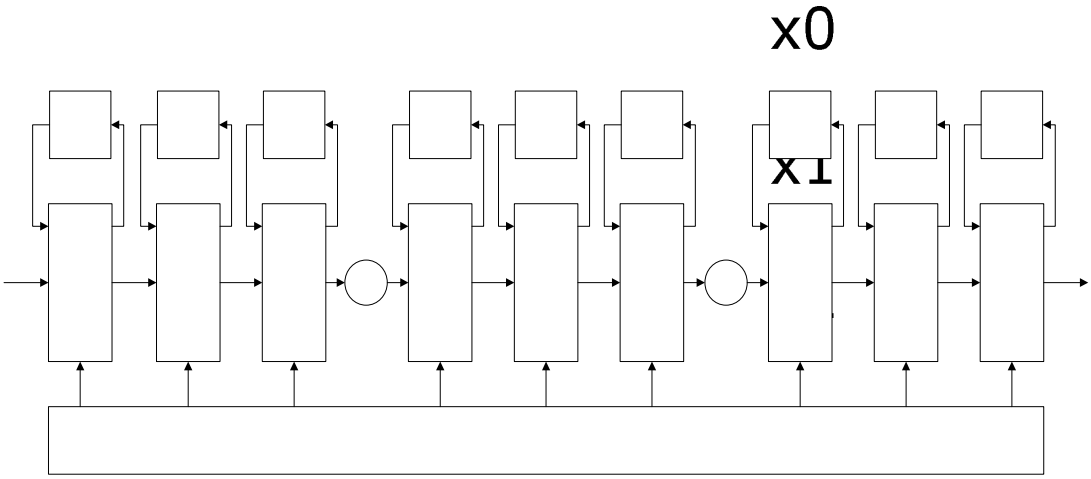


圖 A-8. 512 點的 Radix-2^3 管線化架構圖

A.3 實驗與結果

A.3.1 霍夫曼解碼 Huffman Decoding：

這個部分我們在 FPGA 上實現的幾個版本。

(a)有一個大的編碼索引書，變動輸入速率、固定輸出速率的架構。

(b)有許多小的編碼索引書，變動輸入速率、變動輸出速率的架構。

圖 A-9及圖 A-10分別表示固定輸出溯路架構的輸出波形以及變動輸出速率架構的輸出波形，可以觀察到變動輸出速率架構所需要的運算時間不是固定的。

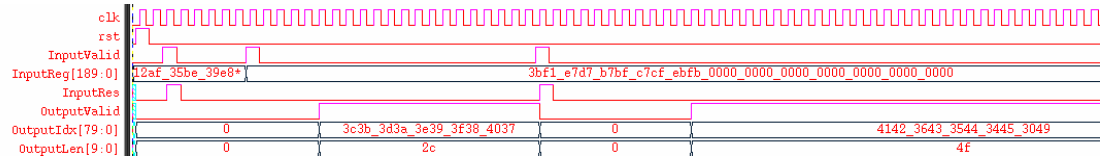


圖 A-9. 固定輸出速率架構的波形圖

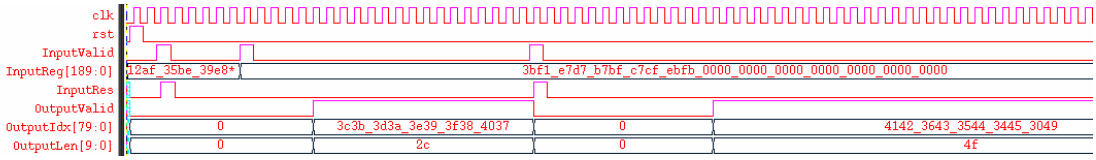


圖 A-10. 變動輸出速率架構的波形圖

表 A-2表示不同架構在 FPGA 上的資源使用率，可以看出變動輸出速率的架構因為控制單元設計的較為複雜，所以需要較多的資源來實現其電路。不過根據電路設計的基本觀點，變動輸出速率的執行頻率應該要比較快，不過這裡顯示的結果卻較慢，應該是 FPGA 設計的因素，或是控制單元的複雜度提高影響到執行頻率。

	IOBs	SLICEs	Freq
(a)	285	830	129.47
(b)	285	989	83.06

表 A-2. FPGA 資源使用結果

A.3.2 反修正餘弦轉換 IMDCT：

這個部分我們在 DSP 上作最佳化的幾個版本。

(a)初步在 DSP 上實現的結果。

(b)將資料送入 FFT 的前處理和後處理的部分針對 DSP 處理方式作作最佳化。

(c)將 FFT 部分針對 DSP 處理方式作最佳化。

表 A-3和表 A-4是實現結果的比較。

	Code Size	Clock Cycle
(a)	20112	1894040
(b)	21496	831295
(c)	4096	91025

表 A-3. DSP 程式大小和執行時間的比較

ODG	32 kbps	64 kbps	96 kbps	SNR(dB)
(a)	-3.71	-1.61	-0.38	278.5798
(b)	-3.71	-1.61	-0.38	73.53463
(c)	-3.74	-2.01	-0.75	29.99346

表 A-4. 客觀差異等級在不同壓縮比率的比較，最右邊是反離散餘旋轉換的 SNR

這裡可以看到客觀差異等級 Objective difference grade 在合理的範圍內，DSP 執行速度大約加快了約二十倍，人耳聽覺幾乎無法察覺其差異。

A.4 結論

在 AAC 編碼器和解碼器的實現方面，我們希望針對 AAC 解碼器作更大幅度的效能改進，因此引入了 FPGA 的實現平來，來解決 DSP 在處理上遇到的瓶頸部分。

在霍夫曼解碼的部分，我們對於 DSP 的處理方式移植到 FPGA 後，作了大幅的改進，將原來固定輸入速率、變動輸出速率的架構，改變為變動輸入速率、固定輸出速率的架構，另外我們考慮一次比對多組的樣本容易造成電路運作上的瓶頸，因此在作了更進一步的修正，降低同時比對的樣本數，亦即把原來的霍夫曼編碼書分散為幾個小的部分，以期能較快速的找出較常出現的碼字。

在反修正離散餘弦轉換的部分，我們採用四分之一點快速傅立葉演算法來取代，除了在 DSP 上作針對 DSP 處理方式的最佳化之外，還將快速傅立葉轉換的部分移到 FPGA 的平台上來實現，使用 Radix-2^3 的快速傅立葉轉換架構來增進其的實現效率。

整體來說，我們利用 DSP/FPGA 不同平台之間的互補性，將 AAC 解碼器的運算效能發揮的更強大。

B. Part 2: Error Control Coding

B.1 背景

本期計畫第一年中已經針對 AAC Audio Coding 編碼法與可調式(scalable)切片式算數編碼(Bit-Sliced Arithmetic Coding，BSAC)進行深入的分析研究，在計畫的第二年我們則要著手將 IEEE 802.16a 標準中，OFDMA 分支規格所規定的正向錯誤改正編碼(Forward Error Correction Coding、簡稱 FEC)，真實實現(Implementation)以及最佳化(Optimization)於以德州儀器(Texas Instrument,簡稱 TI)所生產之數位訊號處理晶片(Digital Signal Processor、簡稱為 DSP)為核心的 DSP 平台上，本計畫所採用的 DSP 平台為 Innovative Integration(II)公司於 2003 年最新推出之產品，代號為 Quixote，內含 TI 的 TMS320C6416 DSP 晶片以及 Xilinx 公司所推出之 FPGA(中文？)晶片，以閘數區分，分別有兩百萬及六百萬兩種版本。

由於電子訊號在透過傳輸媒介(實體網路或無線網路)發送時，或多或少都會遭到周圍環境雜訊的干擾、而產生失真的現象，反映在接收端就是收到錯誤的訊號。為了減少雜訊對訊號判斷的影響程度，可以使用 FEC 的機制，FEC 運作的原理是將原始的訊號流(Bitstream)經過特定的處理(例如：各種不同的編碼)後，轉換成另一串較長的訊號流，這些多出來的訊號長度稱為冗位(Redundancy)，其中包含原始訊號的一些資訊，之後我們可以利用這些冗位對應特定的編碼格式，將經過環境雜訊干擾之後的原始訊號流、有限度地(冗位越多，錯誤更正能力就越強，但也更浪費頻寬)還原回來。這也是為何在大多數的通訊系統中都必定有 FEC 結構的存在。

IEEE 802.16a 標準[1]中所訂定之 FEC 規格為一種串連型態的 FEC，總共由兩層錯誤控制編碼(Error Control Code)所組成，其中之外層編碼(Outer Code)為 (n=255,k=239,t=8)的里德‧所羅門編碼(Reed-Solomon Code)(其 Codeword 皆為 FF(2^8)中之元素)、內層編碼(Inner Code)為限制長度 K=7，編碼率 1/2 的迴旋編碼(Convolutional Code)。除基本的串連式 FEC 結構之外，規格中另外規定在 FEC 結構之前端與後端分別要連接一隨機產生器(Randomizer)與一交錯器(Interleaver)以提升叢集錯誤更正的能力。此外、為了對應傳輸端不同的調變模式，規格中制訂了六種不同的編碼結構，分別對應 QPSK、16QAM、64QAM 三種調變模式，而這六種編碼都是由上述基本的 FEC 加入縮短(Shortening)與穿孔(Puncturing)機制後所產生的，並不需要另外設計不同的 FEC 結構。本篇報告將著重於 FEC 結構於 TI DSP 上的最佳化過程與方法，以及最後的成效。

B.2 研究步驟

B.2.1 里德‧所羅門編碼器之最佳化研究：

參考 TI 出版之 Programmer's Guide[2]，TI 所建議的 DSP 程式設計及最佳化流程可歸納為以下數個步驟：

1. 設計原始 C 語言程式，驗證程式之功能正確性。
2. 使用 TI DSP 專用之程式編譯器 Code Composer Studio (CCS)進行編譯後，執行內附之 Profiler 功能後，分析所產生之檔案 Profile，可以觀察出程式中各個部分功能運行時所消耗的資源(或是運算時間)。
3. 優先由 Profile 中佔去大部分運算時間的函式開始進行最佳化的工作，一般最直觀的最

佳化方法為修改原始程式的演算法部分，藉由直接簡化運算的複雜性或是移除多餘不必要的計算來達到 DSP 硬體的加速，通常經由演算法的改進可以大幅度地提升執行的速度。另外也可藉由 CCS 編譯器回報(Compiler's Feedback)部分，觀察到我們所設計的程式在 DSP 硬體中的資源使用情形以及 TI DSP 特別支援的 Software Pipelining(SP)功能的運作情況，由資源使用的分佈平均與否以及 SP 的平行化效率來判斷我們的程式撰寫是否可以適應 DSP 特殊的硬體架構，透過這方面的微調，在某些情況下也可以得到很好的進步幅度。

4. 當前一個步驟 C 語言方面的最佳化已經沒有顯著的成效時，可以採用撰寫線性組合語言(Linear Assembly)的方式，因為組合語言的層級較低(相較於 C)，因此採用這種最佳化的手段是比較繁瑣而且耗費時間的，但其優點是可以直接指定每一個指令所採用的 DSP 功能單元(Functional Unit)、暫存器位址以及資料路徑。換句話說、即是捨棄軟體本身自動的指令排序，而採用人工手動的方式來做指令的平行化編排。一般的設計情況是不需要使用到這類技巧的，除非當我們發現程式在 CCS 的 SP 中的編排效率普遍很差時才有需要考慮使用。

以下將討論以本段所提前三點為基本方向進行的 DSP 程式最佳化。

B.2.1.1 里德・所羅門編碼之 Profile 分析:

圖 B-1左側所示為在 CCS Profiler 下，RS Codec (***Co***der/***Dec***oder)未最佳化前的 Profile (其中 GP Setup 是指 Generator Polynomial 的建構)；由於 RS Code 編碼有其特殊的演算法結構，故內部所使用的運算絕大部分都不是一般的代數

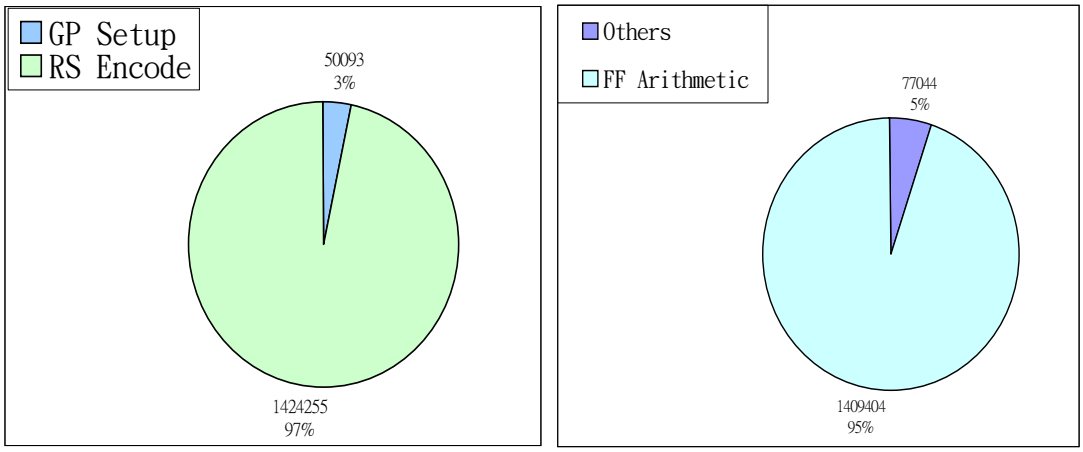


圖 B-1. RS 編碼器之 Profile 分析

運算，而是基於 FF 而發展出來的特殊運算法則(Finite Field Arithmetic)，而在 FF 中進行運算的元素通稱為 Finite Field Element。這種傾向可以由圖 B-2右側的分佈圖看出來，在整個 RS 編碼器的運算中，一般的代數運算與邏輯運算只佔了約 5%的運算量，而 FF 則佔了 95%以上。在實際上對程式進行最佳化修改之前，我們先考慮可以從哪些地方著手，首先是 RS 編碼器部分，佔了 97%的運算量，但是 RS 編碼的過程，事實上僅是一個簡單線性回饋的平移暫存器(Linear Feedback Shift Register)，因此在編碼的過程上並沒有發現特別有改進

空間的地方。值得特別注意的地方是在 FF 的運算部分，由於 FF 的運算我們是參考[3]中所提出的 Mastrovito 架構，使用「乘積矩陣」(Product Matrix)的概念，當元素 A 與元素 B 相乘，需將元素 B(事實上為一個 8 階的多項式)的各階係數寫成一個 8x1 的矩陣，再與乘積矩陣(此矩陣為元素 A 的各階係數的函數，參照圖 B-2所示)Z 進行矩陣乘法，最後才可得到 A 與 B 的乘積，這種運算方式，相較於最直觀的多項式相乘已經有大幅度的進步，不過每次計算前均需要先將各個參與運算的元素的各階係數擷取出來。DSP 對於這類單一位元的擷取並不擅長，因此會大量消耗 DSP 的運算資源。

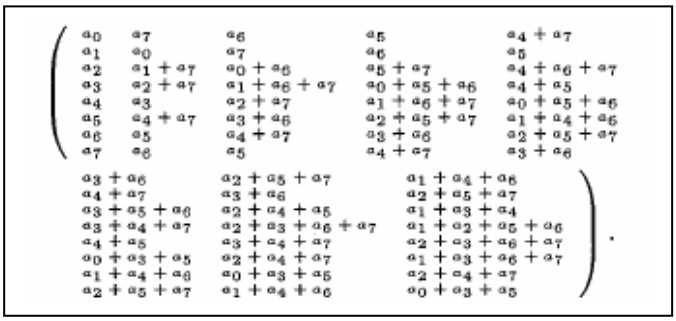


圖 B-2. 乘積矩陣 Z 示意圖

為了突破這個瓶頸，我們希望能找到其它在 DSP 上能更簡單計算的 FF 運算架構。這裡我們參考了[4]所提出的另一種 FF 運算方法，稱為對數表格搜尋法(Logarithmic Table Lookup Method)，這種方法是利用 FF 中的各個元素、都可以用編號 1~(m-1)元素、包含α^0、α^1…α^{m-1}的線性組合來表示的特性來加以利用的，由於每個元素在 FF 中都有其獨一無二的編號(或者可稱做"指數")，因此可以形成一對一的對照表格，以 FF(2^8)為實例說明；若元素 A 與 B 進行乘法，只需要事先建立一個 FF(2^m)的對數對照表格(包括使用元素編號搜尋多項式係數的對照表、稱做 alog table；以及使用多項式係數搜尋編號的對照表、稱做 log table)，接著將 A 與 B 經由 log table 查出相對應的元素編號，那麼乘法就變成很簡單的編號相加(或稱做指數相加)，接著對相加後的結果取 mod(2^m-1)(這是為了防止出現 FF(2^m)中不存在的元素)，最後的步驟是利用 alog table 查出這個相乘後的編號所對應的多項式係數，便可計算出 A*B 的結果。相較於之前的矩陣乘法，這種單純查表的方式對 DSP 的負荷明顯的降低許多，唯一的一個缺點是需要花費一些記憶體空間來儲存事前計算出來的對照表格，以及需要多花費時間進行記憶體的讀取動作(DSP 讀取記憶體所需花費的時間較讀取暫存器要多出不少)；但是總結起來，依然比之前的矩陣運算快了不少。

到此在 RS 編碼器演算法上的改進告一段落，接下來我們轉從針對 DSP 硬體架構的特徵來進行改進；首先我們發現在 DSP 上的資料型態與在個人電腦上有些許不同，在 DSP 上宣告為 long 的變數需要佔去 40 個位元，而在 C 上只需要 32 個位元，這之間的差距使得宣告為 long 的變數在 DSP 上執行時需要多花費一些指令來處理，因此、若沒有使用到 40 位元的需要，我們都避免使用 long 宣告變數。此外針對不同領域使用者的需求傾向，TI 在一系列的 DSP 晶片上提供了一些特殊應用(Application Specific)的內建指令(Intrinsic Function)，這類指令的特徵是 TI 提供直接相對應的組語指令來支援這類的特殊指令，換句話說，TI 在其 DSP 上為一些常用的應用功能、設計了特殊的硬體架構來實現，這種硬體架構可以大幅度地提升 DSP 在該應用的處理速度。很幸運地、TI 最新款的 DSP TMS320C6416 提供了針對 RS 編碼應用的內建指令：GMPY4，這個指令提供使用者可以同時快速處理四個 FF(2^8)的元

素乘法，由於這是 TI 針對 FF 運算所設計的特殊"硬體"架構，相較於之前我們從演算法上進行"軟體"的改進而言，特殊硬體所帶來的優勢是大很多的。但是這並不意味著之前的努力是白費的，因為 TI DSP 提供的內建指令僅支援 FF 的乘法，這對 RS 編碼器而言是足夠的，但對於 RS 解碼器來說，還需要 FF 的倒數運算，這時候之前的對數表格搜尋法便需要被使用了。

接下來 RS 解碼器的最佳化也遵循著類似之前編碼器的流程來進行，由於之前進行 RS 編碼器的最佳化時，發現 FF 運算使用 Mastrovito 演算法的速度過慢，為了讓實驗數據較有可比性，因此在 RS 解碼器的部分，我們一開始就採用對數表格搜尋的方法來實現 FF 運算；圖 B-3 中 RS 解碼器的初始 Profile 即是採用對數表格搜尋法後的結果，由圖中可以發現、在 RS 解碼器的運作中，Syndrome 的計算與 RS 的解碼部分是佔最多運算量的兩個部分，其中 Syndrome 的計算是在重複地做如　　圖 B-4電路的動作，將α^1、α^2...α^{16}代入所收到的資料串中，以求出 S_1 到 S_{16} 的數值後，送到 RS 解碼部分進行解碼以及錯誤的更正，這一部份的運算其實很單純，沒有什麼改進空間，只是因為需要代入 16*255(16 個 Syndrome 和 255 個 Codeword)次，因此才會耗去相當多的運算時間。

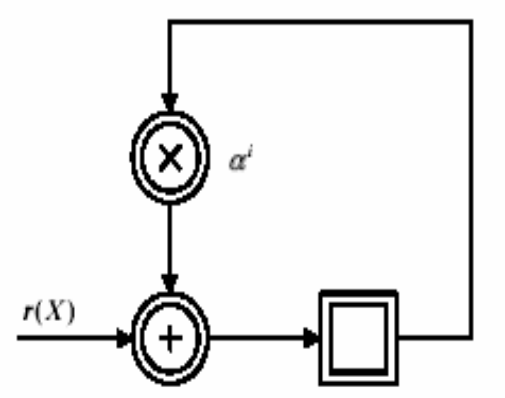


圖 B-3. RS 解碼器之 Profile 分析

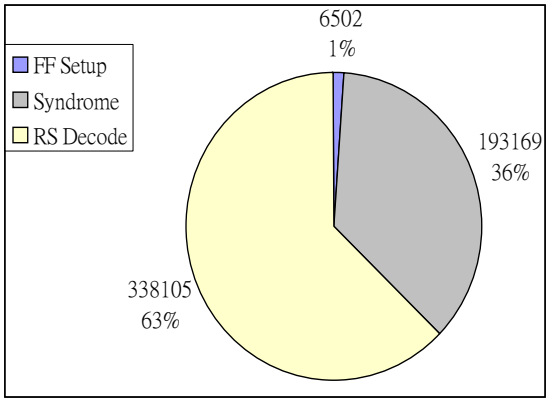


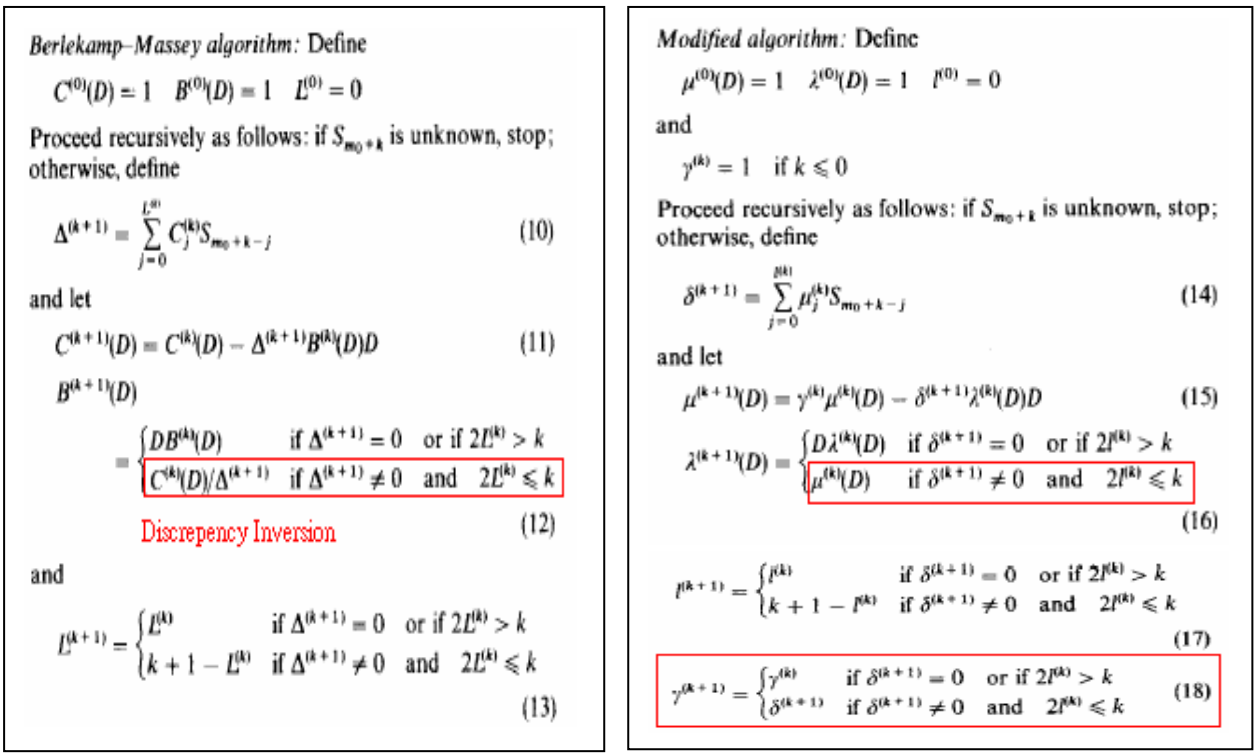
圖 B-4. Syndrome 運算架構

而 RS 解碼部分，以功能而言，可區分為三個主要的區塊，分別是 Berlekamp Massey (BM) 演算法、Chien Search 以及 Forney 演算法，BM 演算法的目的在於求得錯誤位置多項式(Errata Locator Polynomial，ELP)、Chien Search 的目的在於求得 ELP 的根、Forney 演算法則是利用上述兩個部分得出的結果來計算錯誤的數值，以達到更正錯誤資料訊號的目的，根據分析的結果，BM 演算法約佔 14%運算量，Chien Search 佔 77%運算量，而 Forney 演算法僅佔 9%運算量，我們首先由最耗費運算量的 Chien Search 來修改，Chien Search 的作法是使用 Exhaustively 的方式來算出 ELP 的根，換言之、就是需要將所有 FF(2^8)中的元素(共 255 個)都代入 ELP，若是總和為零則代表這個元素是 ELP 的其中一個根。這種繁瑣的運算是導致 Chien Search 消耗大量運算時間的原因，在此我們是使用提早終結(Early Termination)的技巧來降低運算量，因為我們的 RS 解碼器是用來解縮短(Shortening)及穿刺(Puncturing)過的 RS Code，因此最後的幾個 Code Symbol 必須標示成 Erasure，這就代表在求 ELP 根的時候，最後的那幾個 Code Symbol 的位置，一定會是 ELP 的其中幾個根，如此一來，我們可以事先設定一個數值，此數值為 ELP 的階數減去 Erasure 的數目，當 Chien Search 求出等同此數值數量的根之後，就代表已經找出 ELP 所有的根，可以跳出 Chien Search 的迴圈運算，這樣

便可以有效降低這一部份的運算量。

接下來我們針對 BM 演算法的部分做修改，由於 BM 演算法中每一 iteration 都需要計算一次 discrepancy (△)的倒數，而倒數這種運算在一般的 FF 運算中是比乘法還要複雜許多，因此我們採用[5][6][7]所提出、改良過的 BM 演算法，如圖 B-5所示，左邊為原始的 BM 演算法，右邊為修改後的版本，我們可以看到修改後的版本不需要計算△的倒數，這對程式的運算量有一定程度的幫助。

最後、為了大幅度加快 RS 解碼器的運算速度，我們在 FF 乘法的部分也和 RS 編碼器一樣，採用 TI 所提供的內建指令 GMPY4，但是與 RS 編碼器不同的地方在於，RS 編碼器使用 GMPY4 之後，便可以不需要再建立對照表格(Lookup Table)，這是因為 RS 編碼器中只需要用到 FF 的乘法，而 GMPY4 已經提供這種功能了。但對於 RS 解碼器而言，雖然在前面的步驟中我們消除了 BM 演算法中需要使用的 FF 倒數運算，但是在 Forney 演算法的部分，無可避免地必須使用到 FF 的倒數運算，這時候就必須在使用到對照表格。



Berlekamp–Massey algorithm: Define

$C^{(0)}(D) = 1 \quad B^{(0)}(D) = 1 \quad L^{(0)} = 0$

Proceed recursively as follows: if S_{m_0+k} is unknown, stop; otherwise, define

$$\Delta^{(k+1)} = \sum_{j=0}^{L^{(k)}} C_j^{(k)} S_{m_0+k-j} \tag{10}$$

and let

$$C^{(k+1)}(D) = C^{(k)}(D) - \Delta^{(k+1)} B^{(k)}(D) D \tag{11}$$

$$B^{(k+1)}(D) = \begin{cases} DB^{(k)}(D) & \text{if } \Delta^{(k+1)} = 0 \text{ or if } 2L^{(k)} > k \\ C^{(k)}(D)/\Delta^{(k+1)} & \text{if } \Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)} \leqslant k \end{cases} \tag{12}$$

Discrepency Inversion

and

$$L^{(k+1)} = \begin{cases} L^{(k)} & \text{if } \Delta^{(k+1)} = 0 \text{ or if } 2L^{(k)} > k \\ k + 1 - L^{(k)} & \text{if } \Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)} \leqslant k \end{cases} \tag{13}$$

Modified algorithm: Define

$\mu^{(0)}(D) = 1 \quad \lambda^{(0)}(D) = 1 \quad l^{(0)} = 0$

and

$\gamma^{(k)} = 1 \quad \text{if } k \leqslant 0$

Proceed recursively as follows: if S_{m_0+k} is unknown, stop; otherwise, define

$$\delta^{(k+1)} = \sum_{j=0}^{l^{(k)}} \mu_j^{(k)} S_{m_0+k-j} \tag{14}$$

and let

$$\mu^{(k+1)}(D) = \gamma^{(k)} \mu^{(k)}(D) - \delta^{(k+1)} \lambda^{(k)}(D) D \tag{15}$$

$$\lambda^{(k+1)}(D) = \begin{cases} D\lambda^{(k)}(D) & \text{if } \delta^{(k+1)} = 0 \text{ or if } 2l^{(k)} > k \\ \mu^{(k)}(D) & \text{if } \delta^{(k+1)} \neq 0 \text{ and } 2l^{(k)} \leqslant k \end{cases} \tag{16}$$

$$l^{(k+1)} = \begin{cases} l^{(k)} & \text{if } \delta^{(k+1)} = 0 \text{ or if } 2l^{(k)} > k \\ k + 1 - l^{(k)} & \text{if } \delta^{(k+1)} \neq 0 \text{ and } 2l^{(k)} \leqslant k \end{cases} \tag{17}$$

$$\gamma^{(k+1)} = \begin{cases} \gamma^{(k)} & \text{if } \delta^{(k+1)} = 0 \text{ or if } 2l^{(k)} > k \\ \delta^{(k+1)} & \text{if } \delta^{(k+1)} \neq 0 \text{ and } 2l^{(k)} \leqslant k \end{cases} \tag{18}$$

圖 B-5. Berlekamp Massey's Algorithm 之比較與改進

B.2.2 迴旋編碼之最佳化研究：

在迴旋編碼(Convolutional Code)的部分，和 RS 編碼相同，也有一個迴旋編碼器與一個迴旋解碼器，但是在迴旋編碼器的部分，由於原始的架構已經很簡單而且速度已經很快，因此我們決定不在這上面花費多餘的力氣，而把重心擺在整個 FEC 結構中的瓶頸：迴旋解碼器之上，迴旋編碼器有時也稱為 Viterbi 解碼器(Viterbi Decoder、之後將簡稱為 VD)，這是因為大部分的迴旋編碼都是採用 Viterbi 演算法來解碼。Viterbi 演算法大致上可以區分成三個部分，第一部份是讀取 Branch Metric 以及 State Metric、第二部分是"加-比-選"(Add-Compare-Select，簡稱 ACS))，這一部份的工作是將相對應的 Branch Metric 與 State

Metric 相加，之比較兩者大小，再選取較小的那個結果來更新 State Metric 並且記錄對應的路徑。第三部分是追溯 (Traceback)，Traceback 的功用是讀取之前 ACS 部分所記錄下來的解碼路徑，最後便可以得出解碼後的資料。由上述的簡單說明，不難發現 Viterbi 演算法其實並不是一個十分複雜的理論，所使用的運算也是簡單的相加減的運算，但是為何 VD 會成為整個 FEC 結構中的瓶頸呢？這是因為雖然 VD 的單一運算都很簡單，但是需要計算很多次，以我們的 FEC 規格為例，K=7 的情況下，我們需要做 $2^{K-1} = 64$ 次的 ACS 運算才能解碼出一個位元的資料，當 K 更大的時候，運算量也會呈指數成長。因此，我們發現 VD 在 DSP 上的最佳化和之前的 RS 編碼有些不同，並不著重在演算法的部分（因為演算法本身並不複雜），而是著重在程式的寫作與微調上，必須要讓我們的程式能夠盡量適應 DSP 特殊的硬體架構，才能充分的利用 DSP 的運算資源。

在 VD 的最佳化中，為了讓 VD 能夠充分的利用 DSP 的運算資源，我們參考[8]做了四個主要的修改，第一個是 Branch Metric 運算的資料型態，由於我們所使用的 VD 是 Soft-Decision 解碼的類型，因此當我們進行 ACS 的運算時，會出現浮點數的運算，而我們所使用的 DSP 晶片是專門針對固定點(Fixed Point)運算所設計的，對於浮點數的運算非常地緩慢；為了避免這種情形，我們設法將浮點數的運算改變成固定點的運算，由於 ACS 的運算中，Branch Metric 的絕對值大小並不重要，僅僅是用來比較誰大誰小而已，因此，只要能夠維持 Branch Metric 間相對的大小關係，那麼即使不是使用浮點數也不會對錯誤更正能力產生明顯的影響，因此我們將原本的浮點數值乘上 1000，換言之就是把小數點往後再移動三位，之後捨去小數點後的數值，這樣就可以進行固定點的 Branch Metric 運算。

其次我們進行的第二個改進是針對程式中存取記憶體的部分做修改，由於在 TI DSP 的 Pipeline 層級中，存取記憶體是在第五個 Execution Phase 才能完成，因此盡量減少記憶體方面的存取可以對程式的效率有相當大的助益，在這裡我們將原本 ACS 中用來記錄解碼路徑的媒介，由 Pointer 改為 DSP 內部的暫存器，原本使用 Pointer 的狀況下，每記錄一筆路徑就需要存取記憶體一次，而在經過修改後，我們使用 8 位元的暫存器來存放路徑，這樣一來變成每記錄八筆路徑後才需要存取記憶體一次，可以避免多餘的記憶體存取所浪費的時間。在經過第一次的修改後。我們更進一步把暫存記憶體的大小放大為 32 位元(這也是我們所採用的 DSP、其暫存器最大的容量)，將儲存解碼路徑的記憶體存取次數又降低了許多。

第三步我們先仔細瀏覽 CCS 編譯我們的 DSP 程式後所提供的 Compiler’s Feedback，發現在 ACS 的部分 Software Pipelining 的效率很差，追究其原因是出在 ACS 部分的指令間有很強的關連性存在，使得 CCS 的編譯器沒辦法為我們做出好的 Pipeline 切割。而這個關連性出現的原因主要是因為 VD 的 ACS 部分、兩兩特定 State 間有類似 Butterfly 的結構存在，而為了利用這個 Butterfly 結構，我們宣告了兩個計數器(Counter)分別為 i 與 i/2。這樣宣告的結果雖然可以使 ACS 部分的程式變的更簡單，但是卻出現一個意料之外的問題，那就是 i 與 i/2 之間是有關連性存在的，這樣一來、只要是使用這兩個 counter 當作 index 的變數，彼此之間也都會產生關連性。為了改善這種現象，我們另外宣告一個計數器 j，這個計數器每當 i+2 的時候就會跟著 j+1。換言之，j 實際上的身份等同於 i/2，但是由於我們並不是將 j 直接宣告為 i/2，所以彼此之間不會產生關連性，CCS 編譯器也可以順利地替這部分的程式安排 Software Pipelining。

最後一步我們所做的最佳化是針對 ACS 中、每經過一級(Stage)就需要更新 State Metric 的部分做調整。起初我們的作法是宣告一個 next_state_metric 與一個 current_state_metric，當進行 ACS 運算時新產生的 state metric 先存在 next_state_metric 中，然後每經過一級，就將之前暫存的 next_state_metric 更新到 current_state_metric 中。這麼做的好處是在閱讀程式

時很容易看懂，但缺點是每經過一級就要特地做一次更新的動作，很沒有效率。於是我們採用類似迴圈展開(Loop Unroll)的技巧，把全部的 Stage 分成兩半，區分成奇數與偶數兩種，當處於偶數級的時候，ACS 運算產生的 state metric 放至 next_state_metric 中，而在奇數級的時候，則是放至 current_state_metric 中。這樣一來在 ACS 運算的過程中，同時也實現了原本要做的 state 更新動作，節省了之前特地去做 state 更新所浪費的運算時間。

B.3 實驗與結果

B.3.1 里德‧所羅門編碼器最佳化之實驗數據：

這裡將列出我們使用 CCS Simulator 模擬 RS 編碼器與解碼器的數據結果，首先我們先列出 RS 編碼器的部分，表 B-1是一開始使用 Mastrovito 演算法來進行 FF 運算的結果，表中建立 GP 一欄指的是產生出 Generator Polynomial 所需花的 Cycle 數，而 RS 編碼一欄中的 Cycle 數是有包含部分 FF 乘法運算的部分。此外表 B-1中第五及第六列所列的，是在進行 Mastrovito 演算法之前以及之後所需額外處理的步驟。整體運算速度約為 155kbps，並不是很快。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**
整體	1	1486448	155.00
FF 乘法運算	3959	1409404	Improvement (%)
建立 GP	1	50093	Initial
RS 編碼	1	1424255	
取得元素之各階 Binary 係數	7918	316720	
將 Binary 乘積係數還原為乘積元素	3959	237540	

表 B-1. 採用 Mastrovito 演算法實現 FF 運算之 RS 編碼器 Profile

表 B-2是當我們把 FF 運算由 Mastrovito 演算法修改為 Logarithmic Table Lookup 後模擬所得的數據，表中第三列的建立 FF 對照表是指建立 Logarithmic Table Lookup Method 所需的對照表，相較於採用 Mastrovito 演算法的架構，使用對照表搜尋法的整體速度明顯快了許多，進步幅度達 5 倍之多，達到 974.86kbps，我們認為這是因為 Mastrovito 演算法在進行矩陣乘法之前，需要先取出乘數與被乘數的各階係數，而這並不適合 DSP 來處理，所以導致運算緩慢。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**

整體	1	236341	974.86
FF 乘法運算	3959	159275	Improvement (%)
建立 GP & FF 對照表	1	22097	528
RS 編碼	1	208646	

表 B-2. 採用 Logarithmic Table Lookup 實現 FF 運算之 RS 編碼器 Profile

表 B-3部分所列出的為採用 TMS320C64x 系列 DSP 內建指令 GMPY4 來處理 FF 運算所得的模擬數據(包含將之前 Data Type 宣告為 long 的變數改為 integer)，從數據上可以看到極大幅度的進步，達到約 18Mbps 的速度，這裡我們得到一個很好的經驗，那就是直接採用特殊的硬體設計來實現複雜的運算會比僅用軟體上的演算法改進來的有效率的多。表中省去了之前表 B-1與表 B-2中的"FF 乘法運算"一列，這是因為乘法部分都採用 GMPY4 這個內建指令來處理了，因此不需要另外建構 FF 乘法的模組。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**
整體	1	12600	18285.71
建立 GP	1	1707	Improvement (%)
RS 編碼	1	4856	1775

表 B-3. 採用內建指令 GMPY4 實現 FF 運算之 RS 編碼器 Profile

從這裡開始我們將列出 RS 解碼器部分的模擬分析結果，首先表 B-4所列為 RS 解碼器未做最佳化前的 Profile，請注意 RS 解碼器部分我們一開始就採用對照表搜尋法來處理 FF 的乘法與倒數運算。一開始未做最佳化前的執行速度約為 416.24kbps。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**
整體	1	553527	416.24
建立 FF 對照表	1	6502	Improvement (%)
Syndrome 計算	1	193169	Initial
RS 解碼	1	338105	

表 B-4. RS 解碼器未進行最佳化前之 Profile

表 B-5列出的是在進行一連串最佳化過程(包含BM演算法的改進、Chien Search的Early Termination、將 Data Type 宣告為 long 的變數改為 integer 以及採用內建指令 GMPY4 處理 FF 乘法)後所得的模擬數據 Profile，最後的處理速度約為 6.2Mbps，較最初進步了約 14 倍。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**
整體	1	36877	6247.80
建立 FF 對照表	1	4624	Improvement (%)
Syndrome 計算	1	3436	1400
RS 解碼	1	28574	

表 B-5. RS 解碼器最佳化後之 Profile

B.3.2 Viterbi 解碼器最佳化之實驗數據：

在這一段中我們將列出 Viterbi 解碼器最佳化過程中的各項模擬數據 Profile，首先表 B-6 為未經最佳化前的 Profile，此時 Branch Metric 的計算是採用浮點數運算。處理速度只有約 30.29kbps。表中第三列的 SDD 代表的是 Soft-Decision Decoding。

種類	**Count**	**Cycles**	**Process Rate (kbits/sec)**
整體	1	1901653	30.29
初始化 State Metric	1	340	Improvement (%)
SDD Viterbi 解碼	2	1900825	Initial

表 B-6. Viterbi 解碼器最佳化前之 Profile

表 B-7中列出的為經過初步最佳化後的 Viterbi 解碼器 Profile，我們初步的改變是將 ACS 部分的浮點數 Metric 運算修改為固定點運算。由於 TI TMS320C64x 系列的 DSP 是專門針對固定點應用而設計的處理器，因此當進行浮點運算的時候需要使用很多的固定點指令去模擬浮點運算，導致耗費大量的運算資源。所以當我們把浮點運算的部分修正為固定點運算後，可以發現處理速度有很大的進步，達到約 893kbps 的速度，比浮點運算的版本快了近 30 倍。表中需要注意的地方為這裡的 Cycle 數是處理 96bytes 的資料量的總和，和表 B-6時處理的資料量並不相同(因為 CCS Simulator 無法輸入太多的浮點數樣本)。

種類	Count	Cycles	Process Rate (kbits/sec)
整體	1	526632	893
初始化 State Metric	1	197	Improvement (%)
SDD Viterbi 解碼	17	515622	2976

表 B-7. Viterbi 解碼器最佳化之 Profile 版本 1

表 B-8所示為將解碼路徑的儲存媒介由 Pointer 改為 32 位元暫存器後所得 Profile，經過減少存取記憶體的次數之後，加速的幅度約為 1.82 倍，達到 1.6Mbps 的處理速度。

種類	Count	Cycles	Process Rate (kbits/sec)
整體	1	294039	1628
初始化 State Metric	1	197	Improvement (%)
SDD Viterbi 解碼	17	283029	182

表 B-8. Viterbi 解碼器最佳化之 Profile 版本 2

表 B-9所示為經過 i/2,i 計數器分離後的 Profile，經由手動去除 ACS 部分的指令關連性(Dependancy)後，可以達到約 13.66 倍的進步幅度，達到 2.225Mbps 的處理速度。

種類	Count	Cycles	Process Rate (kbits/sec)
整體	1	218134	2225
初始化 State Metric	1	197	Improvement (%)
SDD Viterbi 解碼	17	207107	1366

表 B-9. Viterbi 解碼器最佳化之 Profile 版本 3

表 B-10所示為經過迴圈展開(Loop Unroll)後的 Profile，在我們移除每一級多餘的 metric update 動作之後，可以達到約 13.11 倍的進步幅度，最終達到 2.919Mbps 的處理速度。

種類	Count	Cycles	Process Rate (kbits/sec)
整體	1	168892	2919
初始化 State Metric	1	197	Improvement (%)
SDD Viterbi 解碼	17	157865	1311

表 B-10. Viterbi 解碼器最佳化之 Profile 版本 4

B.4 結論與未來工作

在經過我們逐步的最佳化後的 FEC 結構，RS 編碼器部分已經達到約 18Mbps 的處理速度，RS 解碼器部分也有近 6.2Mbps 的速度，迴旋編碼器部分我們沒有特別做最佳化的處理，但是也有 6.5Mbps 的速度。而最關鍵的 Viterbi 解碼器部分，經過我們微調程式的結構之後，也從原先的 30kbps 提升到將近 3Mbps 的速度，不過相較於其他三個部分都達到 6Mbps 以上，Viterbi 解碼器依然是全 FEC 結構中的瓶頸。

針對 Viterbi 解碼器的部分，我們目前還有想到一些可以繼續做改進的部分，由於 Viterbi 演算法中，運算最頻繁的為 ACS 的部分，而 ACS 這種單純但大量的運算其實並不是特別適合由 DSP 來實現，反而比較適合 ASIC 的設計。而我們所使用的 DSP 平台上有附帶一顆 Xilinx 的 FPGA 晶片，經由這個 FPGA 晶片與 DSP 晶片間的連動，我們可以考慮使用類似 ASIC 的設計方式在 FPGA 上實行 ACS 的運算，由於 ACS 部分的運算在同一級(Stage)的 state 之間並無關連性存在，所以我們可以在 FPGA 上大量實現 ACS 單元，然後採用平行化的處理來同時運作這些 ACS 單元，如果設計得當、可以在原來只能處理一次 ACS 運算的時間內，一次完成同一級所需要的 64 次 ACS 運算，藉此消除 Viterbi 解碼器所遭遇的瓶頸。

C. 參考文獻:

[1] *ISO/IEC 13818-7: 1997, Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)*

[2] *ISO/IEC 14496-3, 1999, Information technology –Coding of audio-visual objects – Part 3: Audio*

[3] M. K. Rudberg and L. Wanhammer, "New Approaches to High Speed Huffman Decoding", *IEEE Proc. ISCAS 1996, May. 1996.*

[4] P. Duhamel, Y. Mahieux, J.P. Petit, "A Fast Algorithm for the Implementation of Filter Banks Based on 'Time Domain Aliasing Cancellation'", *IEEE Proc ICASSP May. 1991.*

[5] P. Duhamel, H. Hollmann, "Split-radix FFT Algorithm", *Electronic Letters, vol.20, No.1, pp.14-16, Jan. 1984.*

[6] S. He, M. Torkelson, "A New Approach to Pipeline FFT Processor", *in Proc. 10th International Parallel Processing Symposium, IPPS Apr. 1996.*

[7] S. He, M. Torkelson, "Designing pipeline FFT processor for OFDM (de)Modulation", *IEEE Proc. URSI Int. Symp. Signals, Syst., Electron., pp. 257-262, 1998.*

[8] *IEEE Standard for local and metropolitan area networks, Part 16, Amendment 2.*

[9] *Texas Instruments, TMS320C6000 Programmer's Guide.*

[10] C. Paar, "A new architecture for a parallel finite field multiplier with low complexity based on composite fields," *IEEE Trans. on Comp., vol. 45, pp. 856-861, Jul 1996.*

[11] E. Savas and C.K. Koc, "Efficient Methods for Composite Field Arithmetic," *Electrical & Computer Engineering, Oregon State University, Dec 1999.*

[12] I. S. Reed, M. T. Shih, and T. K. Truong, "VLSI design of inverse-free Berlekamp-Massey algorithm," *Proc. Inst. Elect. Eng., vol. 138, pt. E, pp. 295-298, Sep 1991.*

[13] T. K. Truong, J. H. Jeng, and K. C. Hung, "Inversionless Decoding of Both Errors and Erasures of Reed–Solomon Code," *IEEE Trans. Comm., vol. 46, no. 8, Aug 1998.*

[14] T.K. Truong and J.H. Jeng, "On Decoding of Both Errors and Erasures of a Reed–Solomon Code Using an Inverse-Free Berlekamp–Massey Algorithm," *IEEE Trans. Comm., vol. 47, no. 10, Oct 1999.*

[15] J. S. Lin, *DSP Implementation and error performance study on speech source/channel coding, M.S Thesis, Department of Electronics Engineering,* National Chiao Tung University, Jun 2002.

D. 計畫成果自評

無線通訊為國家重點發展的科技項目，而多媒體服務是寬頻無線網路的最重要應用。然而在無線網路上傳送串流多媒體數據有許多困難，本專題研究將承繼我們過去的經驗與前人的成果，進一步設計發展解決方式。所發展出的技術、經驗及成品極具實用價值，可促進國內工業研發技術開發。

參與工作人員(研究生與博士後)在學理上習得聲訊與語音編碼技術與國際標準。針對寬頻無線網路，設計開發可調式編碼等演算法，成員得到此課題研究與開發產品的經驗與知識。畢業後進入產業，直接有助於產業界開發新產品，提昇我國工業技術能力，達到人才培育之目的。期間研究成果論文兩篇，已發表於國際學術會議，並準備延伸成為期刊論文。

綜合評估：研究內容與原計畫進度與內容大致相符，已達成學術研究創新與人才培育之預目標。整體成效良好。研究成果頗具學術與應用價值，專利一項申請中，學術論文投稿一篇以及碩士學位論文二冊如下表。

Publications:

[1] C.-H. Yang, H.-M. Hang and T. Chiang, *Fast Bit Allocation Algorithm for Audio Coding, ROC Patent filed, March 2004.*

[2] C.-H. Yang and H.-M. Hang, "Bit-weighted Inter-channel Prediction for Subband Audio Coding," *submitted to Audio Engineering Society Convention 2004.*

[3] Chien-tung Tseng 曾建統, *MPEG-4 AAC Implementation and Optimization on DSP/FPGA, MS Thesis,* NCTU, June 2004.

[4] Young Tse Lee 李仰哲, *DSP Implementation and Optimization of the Forward Error Correction Scheme in IEEE 802.16a Standard,* NCTU, June 2004.