第一章　簡介

1.1　研究背景與動機

Ogg Vorbis[1]是一種音訊壓縮格式，有點類似 MP3 等現存的經過有損壓縮演算法（Lossy Compression）進行壓縮的音樂格式。但有一點不同的是，Ogg Vorbis 格式是完全免費、開放程式碼而且沒有專利的限制。Vorbis 是音訊壓縮演算法的名字，而 Ogg 是一個計畫的名稱，這個計畫最主要的目的是設計一個完全公開程式碼的多媒體系統。Ogg Vorbis 的目標是想要取代目前市面上所有有損音訊壓縮格式，當然包括了 MP3。

很多人以為 MP3 格式是免費的，其實是錯誤的，它本身是受到專利保護的。MP3 是由一個叫 Fraunhofer-IIS 公司的產物，從一問世就注定它是一種商業行為。當 MP3 開始流行之後，母公司 Fraunhofer 發表了一份聲明，內容提到了一個重點就是 MP3 格式要開始收專利費用。也就是說，對於 MP3 編解碼器（不管是否為商業或非商業）以及利用此格式來發佈音樂檔案都得要付出一筆版權費。而首當其衝的就是一些免費的 MP3 codec，以致於到目前只剩下 LAME 了。而這也是為何 MP3 codec 如此少的原因。

因此，打著開放程式碼的 Ogg Vorbis 的存在便有了充分的理由。對軟體公司來說，無論是免費或商業軟體都可以以遵循 GNU 的前提之下使用 Ogg Vorbir 之技術，而節省下來的專利費用可以轉變成產品在價格上的競爭力；另一方面，音樂發行商也可以隨心所欲的使用 Ogg 格式來發佈自己的產品而不用顧慮到專利的問題，也可比原本的 MP3 有著更好的音質以及

更小的容量。而消費者也能因各大廠商之間的競爭變大而造成軟硬體上有更低的價格。

在壓縮技術上，Ogg Vorbis 的最主要特點是使用了 VBR（可變位元速率）和 AB（平均位元速率）方式進行編碼。與 MP3 的 CBR（固定位元速率）相比可以達到更好的音質。Ogg Vorbis 其他技術特性還包括：支持類似 MP3 的 ID3 tag，但比 MP3 要靈活而又完整得多，實際上可以加註的 tag 數目是沒有限制的。Vorbis 還具有位元速率縮放功能，可以不經重編碼便可調整文件的位元速率。Vorbis 文件可以被分成小塊並以取樣粒度(granule)進行編輯；Vorbis 支持多個聲道（2 以上）音訊串流並使用了獨創性的處理技術；Vorbis 文件可以以邏輯方式相互連結等。

Ogg Vorbis 的發展並不是一帆風順的。由於它直接影響到傳統音樂工業的利益，因此很可能不會有任何標準化組織接納 Ogg Vorbis 成為標準。這或多或少地影響到 Ogg Vorbis 的應用。還有，作為開放程式碼項目 Ogg Vorbis 面臨的最大問題就是資金短缺。特別是在主要贊助商 IceCast 停止贊助後更是沉寂了一段時間，幾乎令人以為 Ogg Vorbis 會就此打住。但始終 Ogg Vorbis 的開發團體還是熬了過來。

隨著 Ogg Vorbis 的數個 rc（Release Candidate）[1]版本的發佈，它已經越來越成熟。近來發佈的 1.0 rc3 版本與 rc2 相比在性能上也有不錯的增強。在 rc2 中，Ogg Vorbis 增加了多種位元速率的支援，還有實現了針對多聲道音訊（比如 4 聲道）的“通道耦合”壓縮技術；而在 rc3 中除了音質進一步提升之外，最重要的就是增加了稱為 Bitrate Management 的平均位

元速率（ABR）編碼方式，使得基於 VBR 的 Ogg Vorbis 可以更適合利用網路串流來傳送。

在此論文中，我們設計了一個 Ogg Vorbis 音訊解碼器。根據複雜度分析的結果，解碼的過程主要可分為兩部分：較複雜控制部分（如反量化），以及大量計算的部分（如 IMDCT）。我們將控制的部分用軟體來實現，而大量計算的部分則用硬體來實現，利用軟硬體共同設計來完成整個 Ogg Vorbis 音訊解碼系統。我們選擇 LEON2 處理器及 Xilinx Multimedia Board 為發展平台來實現及驗證整個音訊系統。此平台為一個完整的 SoC 設計環境，使用 AMBA 匯流排來當作整個環境的傳輸協定。

1.2 章節安排

本論文的章節安排如下：

第一章說明研究背景與動機。

第二章介紹 Ogg Vorbis 音訊編解碼原理。

第三章介紹系統單晶片設計與發展平台。

第四章則為 Ogg Vorbis 解碼器實作過程。

第五章為結論。

第二章　Ogg Vorbis 音訊編解碼原理

2.1 聲響心理模型（The Psychoacoustic Model）

Ogg Vorbis 之所以能夠達到高壓縮率並維持一定的音訊品質，最主要的原因，就是它採用了聲響心理模型來模擬人耳的聽覺。利用人耳聽覺感知上形成的遮蔽效應（Masking Effect）所得到的遮噪門檻曲線（Masking Thresholds），決定所能容許的最大量化誤差，使得量化後的失真能不被聽見。

此模型皆先將音訊經傅立葉頻譜轉換，再對映到臨界頻帶（Critical Bands），並區分出單頻（Tonal, Sinusoid-like）及非單頻（Non-Tonal, Noise-like）成份，依其所在的頻率位置與強度大小，分別計算遮噪門檻曲線，而整合成整體遮噪門檻曲線，並對映成每個子頻帶訊號編碼時所需的位元數。在介紹人耳的聽覺特性之前，必須先定義一個表示音訊強度的名詞，SPL（Sound Pressure Level）。SPL 是一個評量聽覺刺激強度的標準[2]，單位為 dB。接著說明人耳的聽覺特性。

人耳對音訊的遮噪能力與訊號的能量大小、訊號的頻率位置，以及訊號的特性有關都有關係。能量大的訊號能遮蔽較大的噪音、非單頻訊號的遮噪性比單頻訊號來得好，而高頻訊號的遮蔽效應也比低頻訊號要強。此外，遮噪能力也與雜訊的頻率位置，以及雜訊發生的時間有關：對同一音訊而言，對高頻雜訊的遮噪能力優於低頻；對其後發生的雜訊之遮噪性優於較其前發生的雜訊。因此，經由以上各種的聲響心理實驗，推導出模擬

人耳聽覺的感知模式，建立各種遮噪門檻曲線[3]，以下就一一的來介紹。

2.1.1 靜音門檻曲線（The Absolute Threshold of Hearing）

在安靜的環境下，人耳能夠聽到較細微的音訊，例如針掉落地面的音訊，但仍有許多音訊無法被聽到。依照這樣的特性，我們可以找出一條靜音門檻曲線（又稱為 The Threshold in Quiet），如圖 2-1 所示，若音訊的強度低於靜音門檻曲線，表示人耳聽不到這個音訊。這條曲線可以由式（2-1）的非線性方程式來逼近[4]。

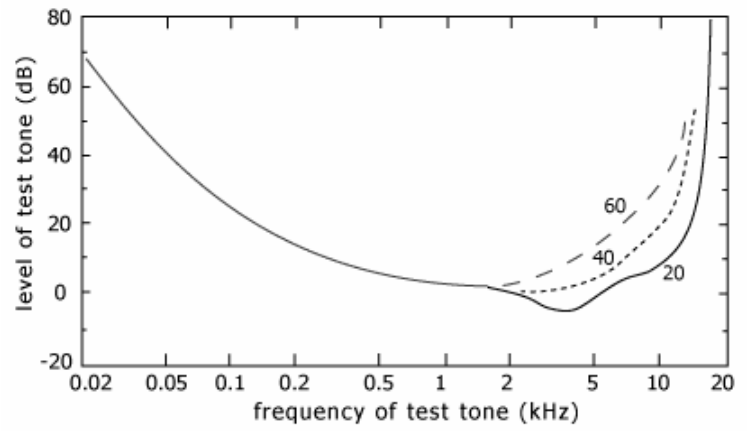


圖 2-1 靜音門檻曲線

$$T_q(f)\times 3.64-(\frac{f}{1000})^{-0.8}-6.5-e^{-0.6\times(\frac{f}{1000}-3.3)^2}=10^{-3}-(\frac{f}{1000})^4 \quad (2\text{-}1)$$

其中 f 代表頻率（Hz），一般而言，人耳所能感知的頻率範圍是 10~20,000 Hz。Tq(f)則是在頻率 f 處的靜音門檻值。 觀察這條曲線可以發現人類的聽覺在低頻與高頻的地方比較不敏銳，而在頻率大約 3~4 kHz 處最為敏銳。

2.1.2 臨界頻帶（Critical Bands）

在經過許多的生理聽覺實驗後發現，人的聽覺系統就像是一個頻率的分析器，範圍大致是從 20Hz 到 20kHz 左右，因人而異。我們可以將這個聽覺系統，看做是由許多的帶通濾波器所組成的，而且每個帶通濾波器的頻寬都不相同，這些帶通濾波器的頻寬大致如表 2-1 所示，我們稱之為臨界頻帶 (Critical Band)。臨界頻帶在低頻段的頻寬大約在數百 Hz 左右，但是在高頻段的時候，頻寬會大到數 kHz。

表 2-1 人類聽覺系統的臨界頻帶[5]

Band No.	Center Freq. (Hz)	Bandwidth (Hz)	Band No.	Center Freq. (Hz)	Bandwidth (Hz)
1	50	– 100	14	2150	2000 – 2320
2	150	100 – 200	15	2500	2320 – 2700
3	250	200 – 300	16	2900	2700 – 3150
4	350	300 – 400	17	3400	3150 – 3700
5	450	400 – 510	18	4000	3700 – 4400
6	570	510 – 630	19	4800	4400 – 5300
7	700	630 – 770	20	5800	5300 – 6400
8	840	770 – 920	21	7000	6400 – 7700
9	1000	920 – 1080	22	8500	7700 – 9500
10	1175	1080 – 1270	23	10500	9500 – 12000
11	1370	1270 – 1480	24	13500	12000 – 15500
12	1600	1480 – 1720	25	19500	15500 –
13	1850	1720 – 2000			

2.1.3 遮蔽效應（Masking）

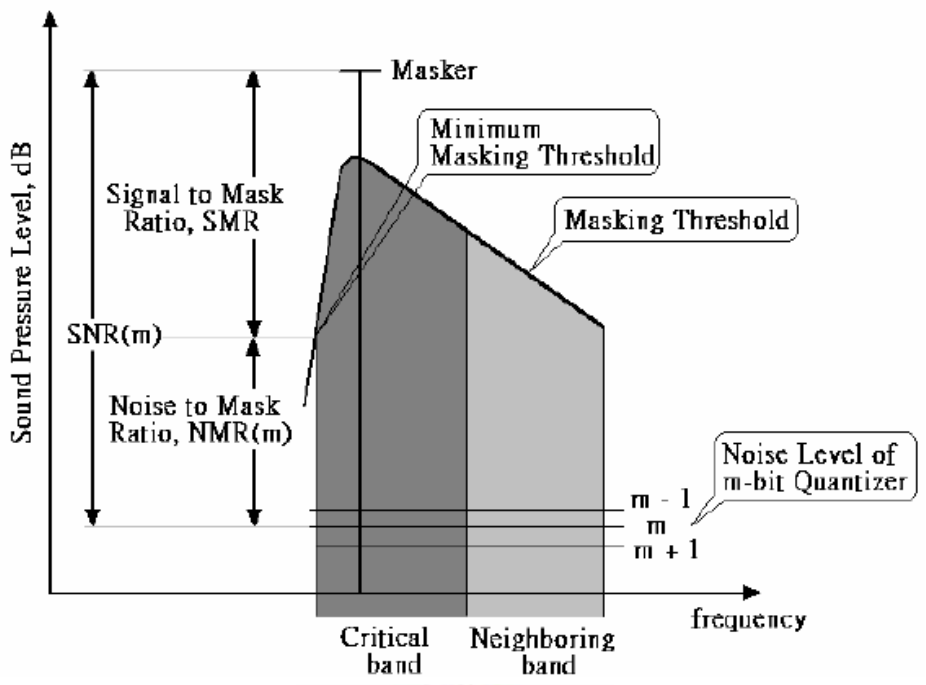


圖 2-2 遮蔽曲線和訊號遮蔽比[6]

除了絕對靜音遮蔽曲線外，科學家還進一步實驗出相鄰頻率間，也有同樣的遮蔽效應存在，請參考圖 2-2。當某個頻率的能量很突出的時候，它會對相鄰頻率的訊號產生遮蔽。這個會遮蔽其他頻率的訊號源，我們稱為遮蔽者 (Masker)，而遮蔽者對相鄰頻率的影響範圍和程度，則和遮蔽者本身是位在哪個臨界頻帶有關，不同臨界頻帶內的遮蔽者，對同一個頻帶內的其它訊號或相鄰頻帶內的訊號，會有不同的遮蔽效果，一般以展開函式 (Spreading Function) 或是遮蔽函式(Masking Function) 來計算。上述在頻域上所發現的兩種遮蔽效應，被通稱為頻域遮蔽效應 (Frequency or Simultaneous Masking)。

在頻域上，SPL 較大的訊號會對頻率相近的訊號產生遮蔽效應，如圖 2-3 所示。圖 2-3 中有三個訊號被 masker 遮蔽，其中一個訊號甚至低於靜

音門檻曲線。注意到圖 2-3 中的遮噪曲線在往低頻的方向較為傾斜，而往高頻的方向則較為平緩，這表示高頻的訊號比較容易被遮蔽。

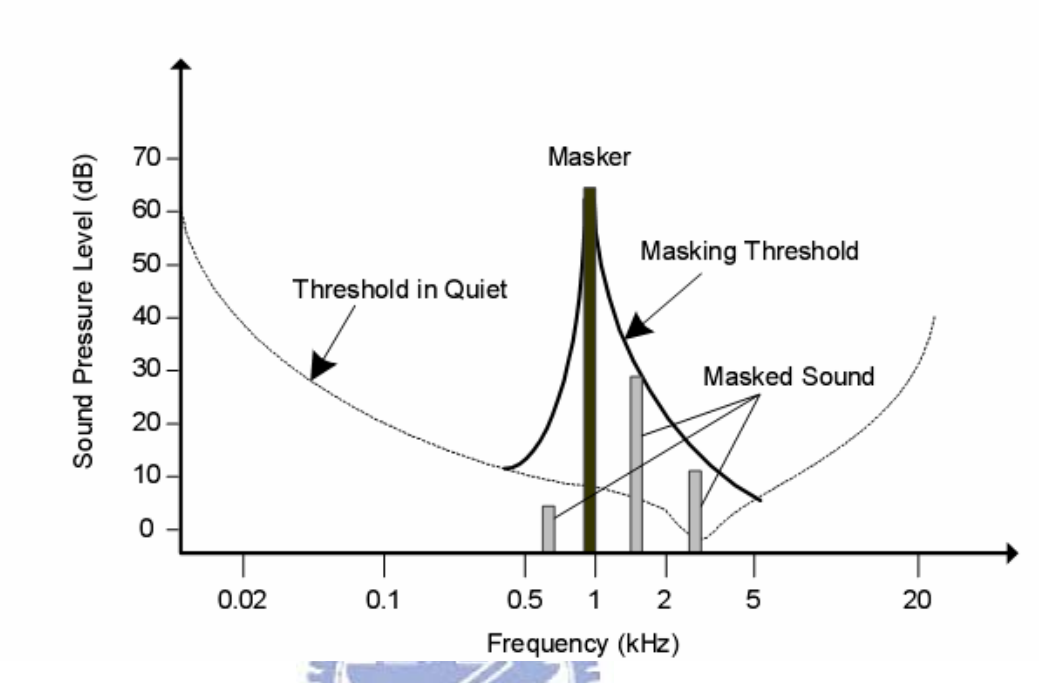


圖 2-3 遮噪門檻曲線與靜音門檻曲線

除了頻域遮蔽效應外，科學家也在時域上發現了類似的遮蔽效應，這些存在於時域的聽覺遮蔽效應，我們稱為時域遮蔽效應 (Temporal or Non-simultaneous Masking)，請參考圖 2-4。依時間發生的前後次序，時域遮蔽效應又被細分成前遮蔽效應 (Pre-masking) 與後遮蔽效應(Post-masking) 兩種[7]。其他的音訊若出現在圖 2-4 中灰色的地方就會被遮蔽。值得注意的是 post-masking 所影響的時間較長，至少有 160ms，而 pre-masking 大約只有 post-masking 十分之一的時間。

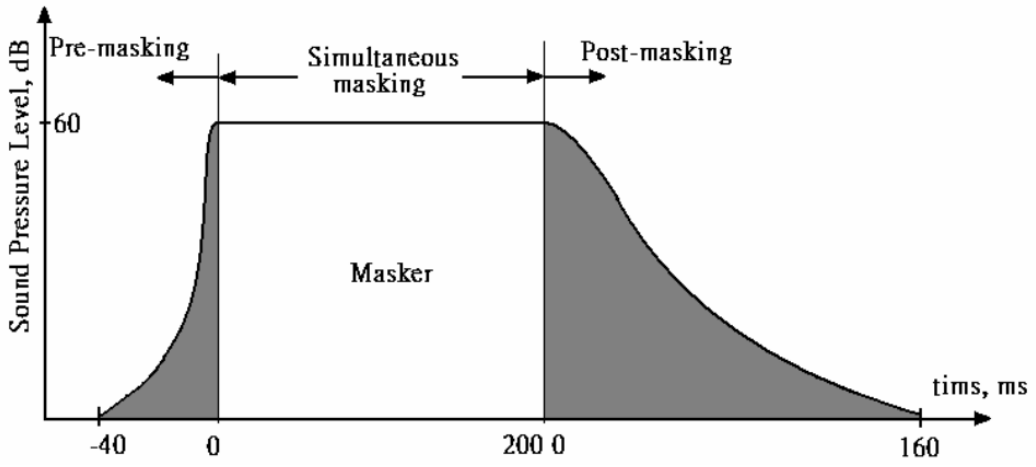


圖 2-4 時域遮蔽效應之前遮蔽與後遮蔽效應

2.2 Ogg Vorbis 封裝格式與解碼過程

在描述 Ogg 的封裝過程之前[8]，必須先了解其中的一些專有名詞，詳情請參照附錄 A。在經過 Ogg 封裝後會得到所謂的“實體位元串流（Physical Bitstreams）“，它是把編碼器所產生的“邏輯位元串流（Logical Bitstreams）“進行封裝的動作。接著邏輯位元串流又被切割成連續的“Packets “。packets 裡並沒有包含邊界的相關資訊，並以不同長度綁在一起形成串流。

2.2.1 Ogg 位元串流

Ogg 的位元串流可分成實體位元串流與邏輯位元串流。實體位元串流是由多個以分頁交錯的邏輯位元串流所組成的，且邏輯位元串流裡的分頁依照一定順序排列。在實體位元串流裡的每個邏輯位元串流都有個獨一無

二的識別序號(位於第一個分頁的 Header 裡)，此識別碼是隨機產生且與編碼的方式無關。每個分頁只能有一種資料型態，但可以有不同的長度，而其中的 Header 包含了封裝及錯誤更正的資訊。實體位元串流中，每個邏輯位元串流的開始都是從一個特殊的分頁 – 起始分頁(Begining of Stream)，而以另一個分頁 – 結尾分頁 (End of Stream) – 結束。

起始分頁除了包含 codec 類型之外，可能還有關於解碼前設定的相關資訊。而且必須還要說明這個媒體是如何被編碼，若以音訊檔為例，起始分頁裡會有取樣頻率、聲道個數等。一般來說起始分頁的第一個位元組裡應該會有 magic data 做為辨識之用(說明需要哪個 codec)。

2.2.2 Ogg Vorbis 封裝過程

從 Ogg 的觀點來看，packets 可以是任何的大小。但是對於每個不同的 codec，它們應該自行定義如何處理 packets，例如將它們分組或著是再切割。因為分頁的最大的長度是 64K 位元組，有時候一個 packet 必須被分散到多個分頁中。為了簡化這過程，Ogg 把每個 packet 切成數個長度為 255bytes 的小段（最後一段小於 255bytes）。這些小段被稱為“Ogg 區段（Segment）“，不過這一步驟僅僅是邏輯上的分段，事實上並未真的發生。

然後一組連續的區段被包成了一個以 header 為首的分頁。在分頁 header 中的區段表（Segmentation Table）告訴 codec 此分頁裡所有區段的“Lacing 值“（代表各 Segment 的長度）。圖 2-5 說明了如何將邏輯位元

串流經過一連串的處理之後轉換成實體位元串流。

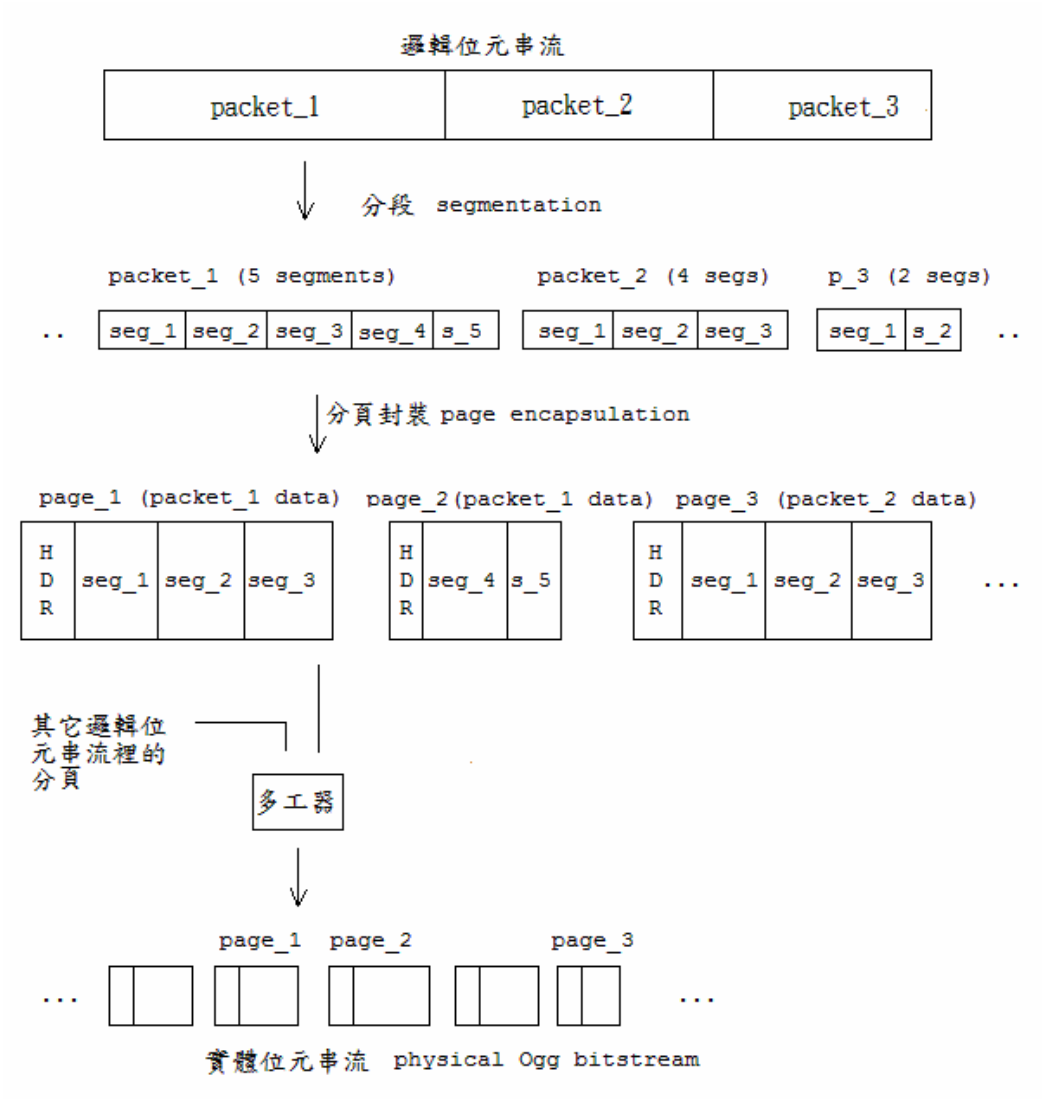


圖 2-5 封裝流程圖

2.2.3 分頁結構

一個實體位元串流由許多大小不同的分頁所組成，通常是 4 到 8kB，最多可以到 65307 個位元組。每個分頁的檔頭都包含了如何還原成原來的邏輯位元串流，以及簡單的錯誤偵測、尋找邊界等相關的訊息。要解讀一

個分頁，只要從此分頁所包含訊息來分析而不需要整個位元串流。圖 2-6 是分頁檔頭的格式。以下說明個欄位所代表的意義：

圖 2-6　分頁檔頭結構圖

1. Magic Number：此字串“OggS“讓解碼器知道各個分頁的邊界，如果遇到毀損的資料，則可用此字串來進行同步。一旦發現是一個新的分頁，會先檢查 CRC 看看此分頁是否完整。
2. 版本：一個位元組大小，用來表示這個 Ogg 檔案所使用的版本。
3. 檔頭類型：一個位元組大小。其中：
   - 第一個位元：是否跟上一個分頁同屬一個 Packet。
   - 第二個位元：是否為邏輯位元串流的第一個分頁。
   - 第三個位元：是否為邏輯位元串流的最後一個分頁。

4. granule 位置：八個位元組的欄位，包含位置資訊。對一個音訊串流來說，它可能包含著 PCM sample 的總數。
5. 位元串流序號：四個位元組，邏輯位元串流的識別號碼。
6. 分頁次序：四個位元組，此分頁在邏輯位元串流裡的次序，可用來檢查有沒有分頁遺失。每個邏輯位元串流的分頁次序是個別計算的。
7. CRC 校驗和：32 位元，檢查分頁的完整性。
8. 分頁-分段數：一個位元組，表示分段表中有幾個分段。
9. 分段表：大小由上一個欄位來決定。此分頁中所有分段的 lacing 值。每一個位元組表示一個 lacing 值。所以檔頭的大小為：

檔頭的大小 = 分頁-分段數＋27 [位元組]

分頁的大小 = 檔頭的大小＋lacing 值的總合

2.3 解碼過程

在解碼之前，必須確認位元串流的檔頭與欲解碼的串流檔相符合以進行初始的動作。Vorbis 裡包含三種檔頭，依規定有先後順序，而且缺一不可；等到準備工作完成，可能會從 Vorbis 串流的任一個音訊 Packet 開始解碼。在 Vorbis I 規格書中，最前面三個初始檔頭之後的 Packet 都屬於音訊 Packet。而這三個檔頭依序分別是識別檔頭(identification Header)、註解檔

頭(comments Header)以及設定檔頭(setup Header)。請參見表 2-2。當準備工作完成後就可進行解碼的動作，大致可分成以下幾個步驟，如圖 2-7。

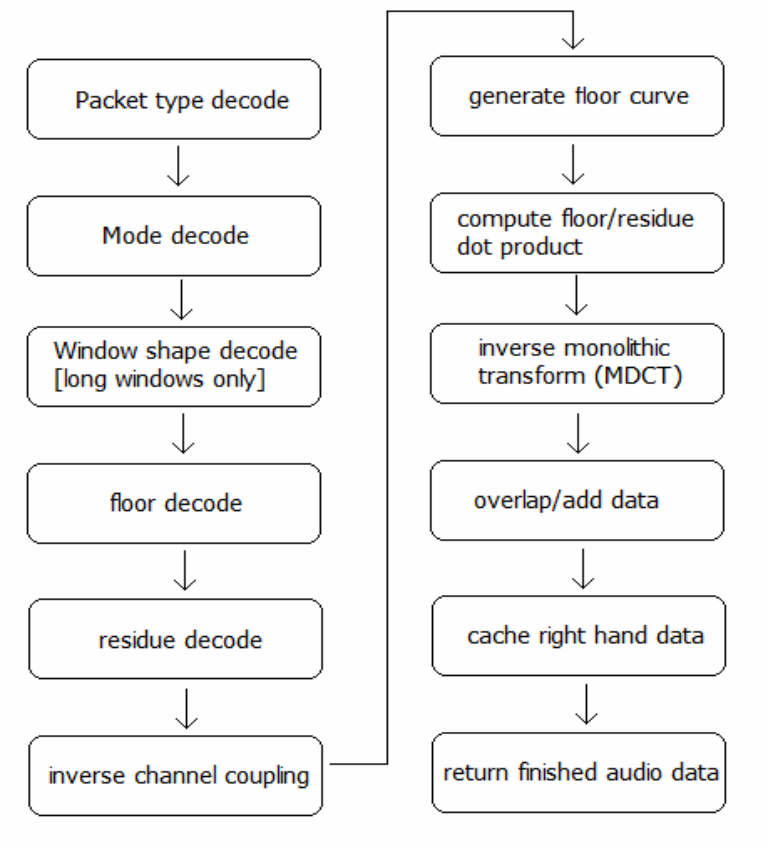


圖 2-7 解碼過程

表 2-2　Ogg 各種檔頭所代表的意義

識別檔頭	從此檔頭可得知此位元串流是否為 Vorbis 格式，版本以及一些此音訊檔的基本特性，像是取樣率與聲道的個數等。
註解檔頭	包含了使用者註解並說明此檔案是如何產生的。
設定檔頭	此檔頭包含了許多解碼時要用到的 codec 設定資訊，VQ 和 Huffman codebooks 等。

第三章 系統單晶片設計與發展平台

隨著半導體技術的快速發展及電路設計軟體的自動化合成，使得晶片設計者能快速地將更多的功能放入單一晶片中，如此一來，不但降低了整個硬體的面積外，更讓所花費的成本變小，而這就是近年來設計業界爭相發展的趨勢：系統單晶片(System-on-a-Chip，簡稱 SoC)設計。

3.1 系統單晶片

對系統單晶片[9]而言，我們可以簡單地解釋為一種將許多擁有完整功能性的電路整合於一顆積體電路的應用。它可以放入中央處理器、記憶體、溝通界面、應用電路、數位/類比轉換器及類比/數位轉換器等一同整合之，如圖 3-1 所示。

系統單晶片具有輕薄短小的特點，使得產品可攜帶，加大系統的實用性與方便性，相對地，要設計整個系統卻是需要花費很大工夫，因此，如何將每次所設計的電路妥善保存與管理，以供下次設計使用，已成了重要學問，亦導致了矽智產(Silicon Intellectual Property，簡稱 SIP)的出現。

3.2 系統單晶片之設計流程

近年來，由於晶片中可容納的電晶體數量愈來愈多，導致IC 設計的

發展快速變化，傳統設計已不敷使用，而系統觀念的加入，更使得IC 設計變得複雜，因此軟/硬體共同設計已成目前主流。如今，系統單晶片的概念出現，對IC 設計工程師而言，更是一大挑戰。以下我們便針對傳統設計、軟/硬體共同設計及系統單晶片設計等流程做個簡單的介紹, [9]。

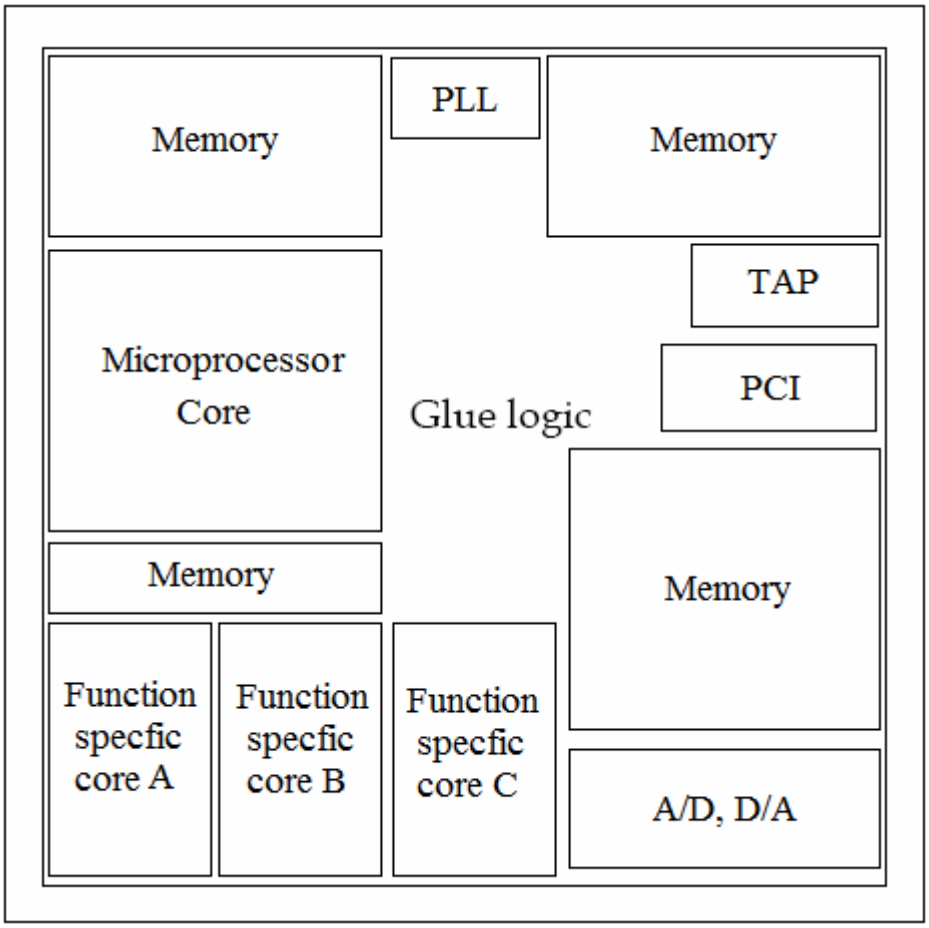


圖 3-1 一般系統單晶片的架構

3.2.1 傳統設計(Traditional design)

特殊用途的積體電路(Application Specific Integrated Circuit，簡稱 ASIC)之傳統設計流程如圖 3-2 所示。首先，我們利用硬體程式語言(HDL)撰寫

所需要的功能函數，接著利用特定的軟體來製作合成電路、驗証硬體功能、模擬合成結果與探討延遲時間等，最後再配置與繞線以轉換成實際的電路佈局。然而傳統設計都只考慮到硬體實現，對於軟體部份，則是要求軟體自行配合，導致可由軟/硬體共同分工的部份，依舊交給煩重的處理器來處理，反而拖慢了處理器的整體速度，因此，才有軟/硬體共同設計的方法出現。

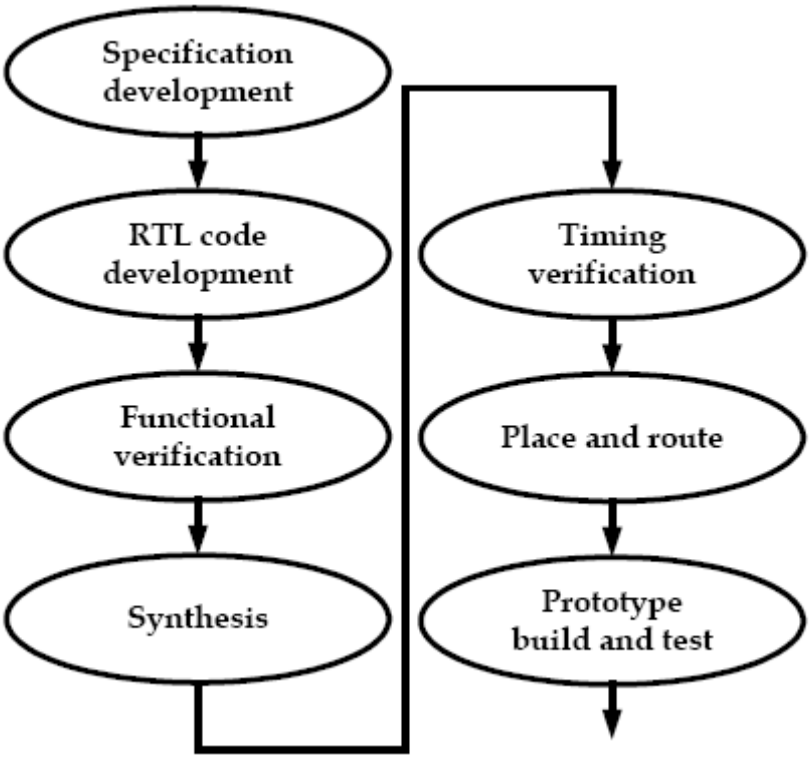


圖3-2 傳統設計流程

3.2.2 系統單晶片設計(SoC Design)

晶片中可容納的電晶體數量不斷提升，導致了系統單晶片的構思出現，其想法就是把構成整個系統的所有模組完完整整地放入於晶片中，以減少系統面積與接線問題。除此之外，當然還要考慮到每個模組間的溝通介面、整體面積及繞線問題等，而系統單晶片的設計流程，如圖 3-3，共分成四個主要部份一同設計。

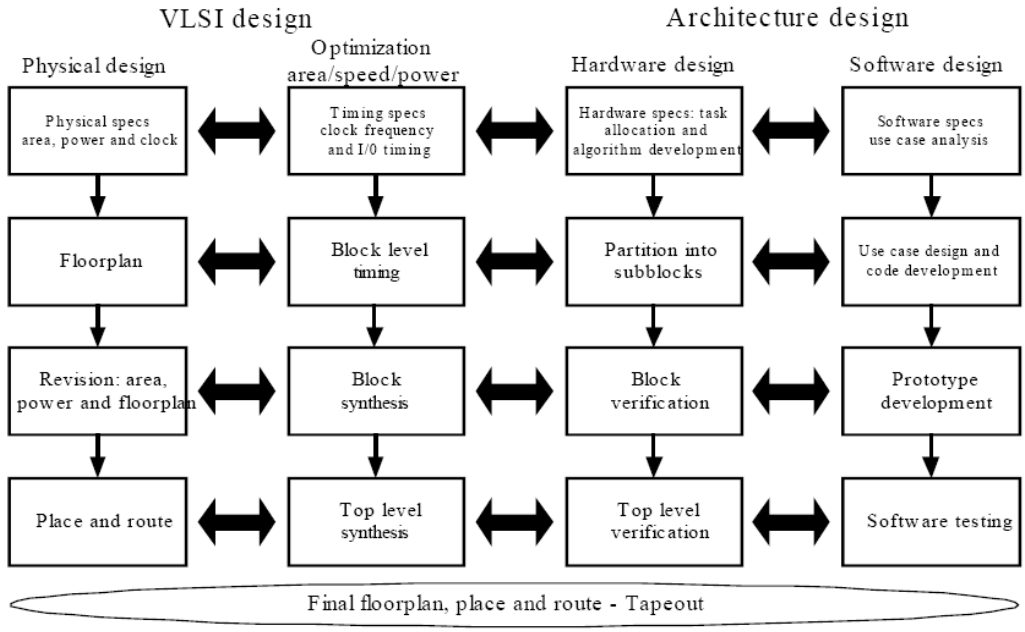


圖3-3 系統單晶片設計流程

3.2.3 軟/硬體共同設計(Hardware/Software Co-design)

現今的設計趨勢已轉變成軟體與硬體的共同設計流程，並漸漸增加硬體的複雜度與功能，由圖3-4來說明。先將整個所設計的系統功能定義後，再依其工作內容化分成三個部分：軟體、硬體與界面，之後再經過個別設

計與驗証，得到所需的軟體及硬體規格，最後再將三者整合並進行整個系統的模擬與實現。現今，軟/硬體共同設計已成主流，然而，微處理器、記憶體與特殊用途IC 等硬體的分開設計，仍導致整個系統的面積十分龐大，所以近年來提倡利用軟/硬體共同設計的概念，搭配最新的製成技術，將系統逐漸整合於一顆單晶片中。

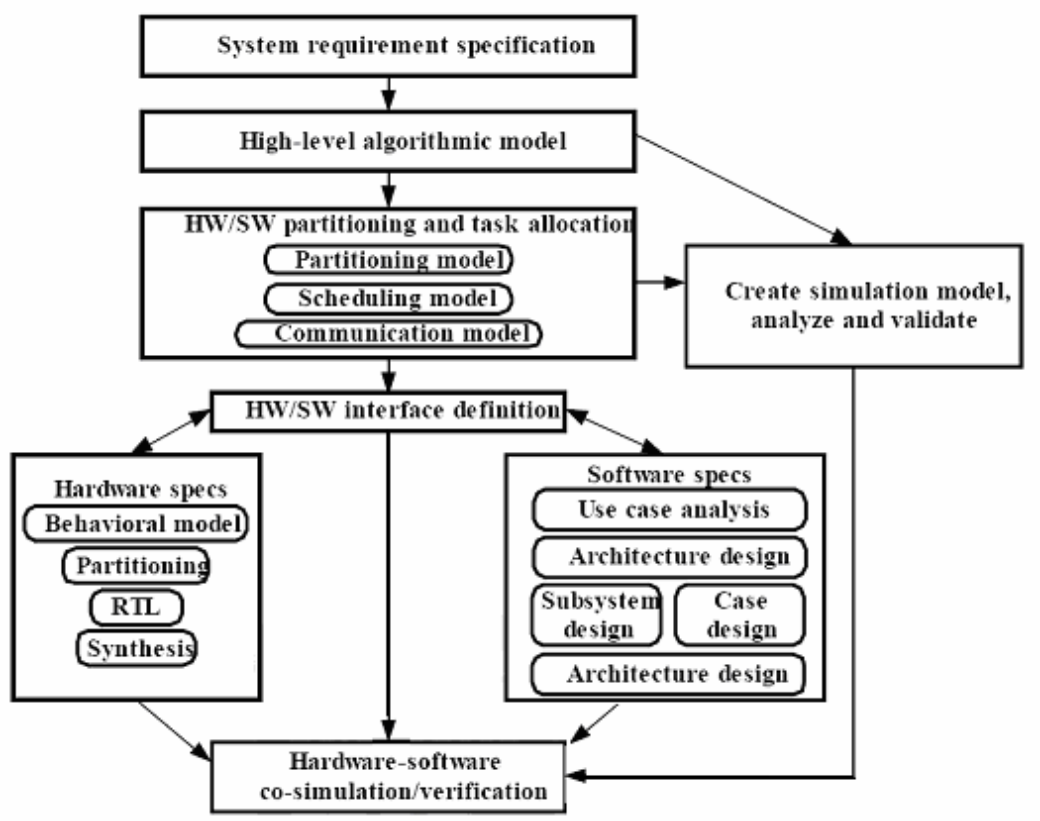


圖 3-4 軟/硬體共同設計流程

3.3 SoC 發展平台

以系統單晶片的觀點來設計 Ogg Vorbis 解碼系統，其晶片內需要有微處理器、專門處理計算複雜度高的特殊硬體、記憶體與數位/類比轉換器

等，但是這需要整間公司的人力才可完成。因此，我們利用以 LEON 為處理器的 SoC 發展平台，並設計一顆特殊用途的硬體，來驗証軟/硬體共同設計的 Ogg Vorbis 解碼系統。此論文中採用的是由 Xilinx 公司生產的發展平台— Multimedia Board[10]，如圖 3-5，它提供我們一顆 Virtex II FPGA 以供驗証。

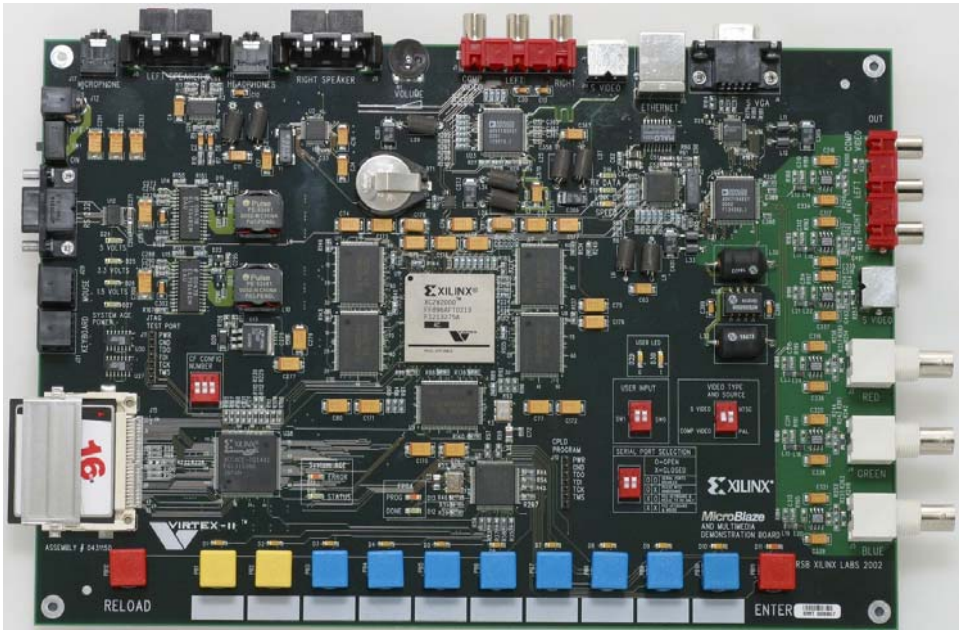


圖 3-5 Xilinx Multimedia Board

系統整合平台主要是提供電源與一些 I/O 介面，並利用匯流排將核心模組與邏輯模組整合在一起，平台上亦有許多輸出界面來協助驗証。核心模組提供 MicroBlaze 微處理器，負責程式的運算及系統的控制，而邏輯模組則有 Xilinx FPGA 以供燒錄所設計的硬體。除此之外，發展平台上亦有 SystemACE，讓我們可將設計好的 bit 檔儲存在 CF 卡中。

3.4 LEON2 處理器

雖然 Multimedia Board 本身有提供一個 soft core 微處理器，但是基於 open source 的精神，我們將採用 Gaisler 實驗室所開發的 LEON2[11][12]的微處理器。LEON2 是相容於 SPARC v8 架構的 32 位元處理器，而且是可合成的 VHDL 模組，圖 3-6 為基本架構。此模組具有高度的可組態性，特別適合拿來當做 System-on-a-Chip 設計的處理器。就如先前所提到的，它的原始程始碼遵照著 GNU LGPL 的協定，允許免費的開發以及重製，不論是否屬於商業或非商業的行為，這也是它在處理器的領域中能夠快速崛起與發展的相當重要的原因之一。除了處理器本身，Gaisler LAB 也提供了完整的文件檔，以及在開發時常用的工具。包括 GRMON、CCS 等。

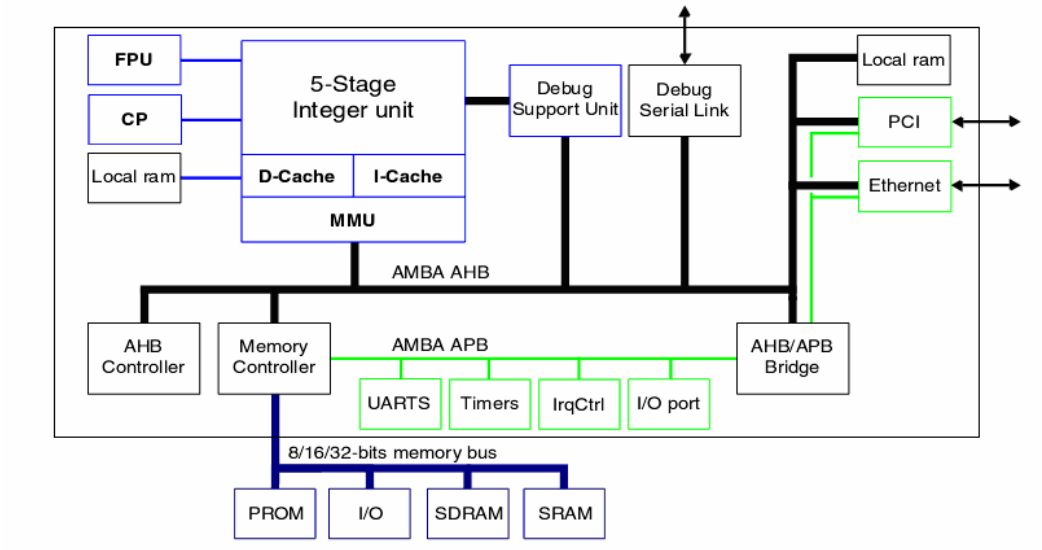


圖 3-6 LEON2 架構圖

GRMON 的全名為 GRMON Debug Monitor for LEON Systems，顧名思義，此工具最主要的功能為除錯，除此之外，還具有模擬的功能。在除錯

這方面，是透過一條 RS232 Cable 連接 FPGA 板與 PC，此通道是由 LEON2 處理器上的 DSU（Debug Support Unit）所建立的，使得我們能夠從 PC 上面下指令或是下載應用程式到板子上運行，同時它也支援 GDB。至於模擬則是將我們寫好的程式經過 CCS 的交叉編譯後，先在 PC 上進行模擬的動作，完全使用軟體來驗證應用程式是否達到預期的結果—但是 LEON 模擬器並不能保證模擬的結果與在實際在硬體上跑完全相同。

Gaisler LAB 提供了三種 CCS（Cross-Compiler System），分別是 Bare-c CCS、eCos Real-time Kernel CCS、RTEMS CCS，這三種 CCS 有各自的函式庫，依不同的 CCS 會在編譯時去連結各自的函式庫。因為我們的撥放程式需具備多執行緒的能力以及作業系統來協調各個模組之間的運作，所以選擇了 RTEMS CCS 作為主要的交叉編譯器。

3.5 AMBA匯流排

在LEON2處理器中採用了AMBA[13]當做整個系統的匯流排協定，AMBA意謂「前瞻微處理器匯流排架構（Advanced Microprocessor Bus Architecture, AMBA）」，此匯流排協定規格係由ARM所提出，目前訂定之規格為2.0版。圖3-7為典型之AMBA系統架構圖。其中包含了有兩種主要的匯流排：AHB與APB。AHB主要是針對高效率、高頻寬以及快速系統模組所設計之匯流排，它可以連結如微處理器，晶片上或晶片外之記憶體模組或直接記憶體存取裝置等高效率模組。AHB規格中支援多重主模組端（AHB Master），而期間AHB匯流排的控制權歸屬則藉由仲裁者（AHB

arbiter）決定。APB則是應用於低速與低功率的周邊模組，可針對週邊作功率消耗及複雜介面之最佳化。APB與AHB的資料傳送係透過APB橋接器（AHB to APB Bridge）而達成。APB橋接器為AHB的從模組端（AHB Slave），卻是APB匯流排中唯一的主模組端。

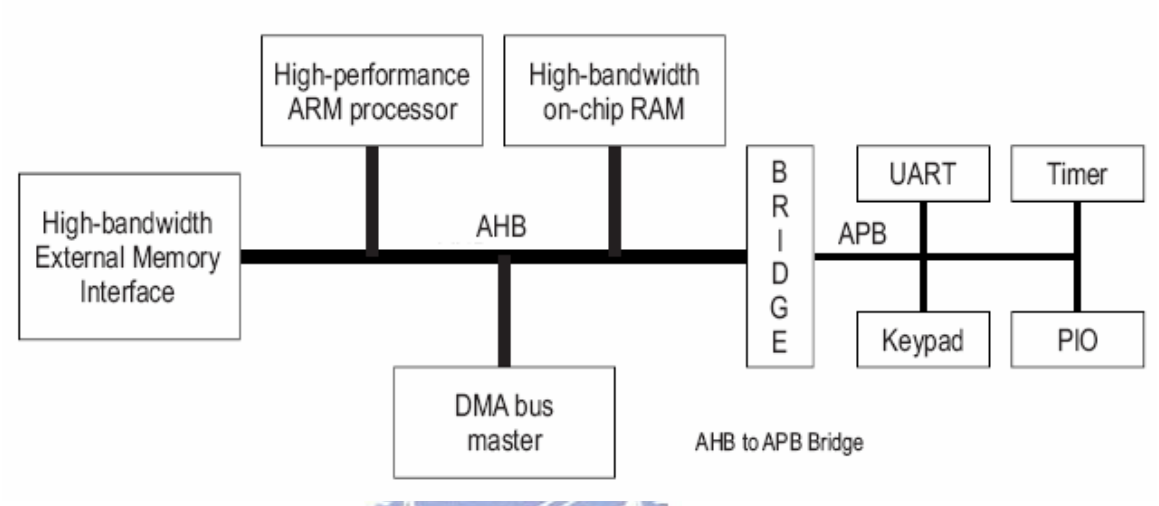


圖 3-7 典型的 AMBA 系統架構圖

3.5.1 AHB

AHB 是一種針對高效率，高頻寬及快速系統模組所設計的匯流排，用來連接微處理器、記憶體、以及PLD 上的硬體模組等等。AHB 匯流排具有幾個特性：

一、支援主模組端發出連續筆資料（Burst）的傳輸。

二、支援分離式（Split）的資料傳送。

三、主模組端 在一個週期完成工作移交。

四、只在時脈週期的正緣觸發作資料傳遞。

五、不使用三態電路。

六、支援64 及128 位元組的資料傳輸。AHB 匯流排上的元件均為同步電路，包含AHB 主模組端、AHB 從模組端、仲裁者以及AMBA decoder。其中主模組端主要負責提供位址和控制訊號去啟動一個讀或寫的資料傳送。從模組端會依主模組端發出的讀寫要求來進行資料的傳送，並作出如成功、失敗或資料傳遞中的訊號回應。仲裁者負責在眾多主模組端同時發出資料傳送要求訊號時，保證只有一個主模組端可以啟動資料的傳輸，雖然說這個仲裁者的介面協定是固定的，但仲裁的演算法可以依不同應用而有不同的實現方法。AMBA decoder負責從資料傳遞的位址中解碼出一致能訊號（Enable Signal）來選擇一個可進行資料傳送的從模組端。

3.5.2 AHB主模組端&從模組端互動方式

AHB 中最基本的訊號線有以下幾個：HCLOCK 為AHB 中的時脈訊號；HADDRESS 為32-bit 的位址線；HWDATA 為主模組端寫入至從模組端的資料；HRDATA為主模組端從從模組端讀取的資料；HWRITE用來控制主模組端目前要進行讀取或寫入的動作；HREADY係從模組端用來告知傳送至目的地的資料是否完成或是需要額外的等待時間。

在對其架構有個大概的了解後，我們再來看一下其基本的運作情形。AMBA AHB主要分為兩個階段，第一階段為位址的傳遞，另一階段為資料的傳遞。其中，資料傳遞所需要的clock cycle可能會因為需要等待而有所差異。資料階段的結束與否取決於從模組端所送出的HREADY 訊號來判

斷，若從模組端正確接收主模組端送出的資料或準備好主模組端需要的資料，則HREADY為1，且在下一次HCLOCK 正緣觸發去擷取從模組端送出的資料或回應訊號。假設從模組端無法完成資料接失收或資料準備，則HREADY 為0，且須額外的延遲等待時間來完成資料階段的資料傳送。

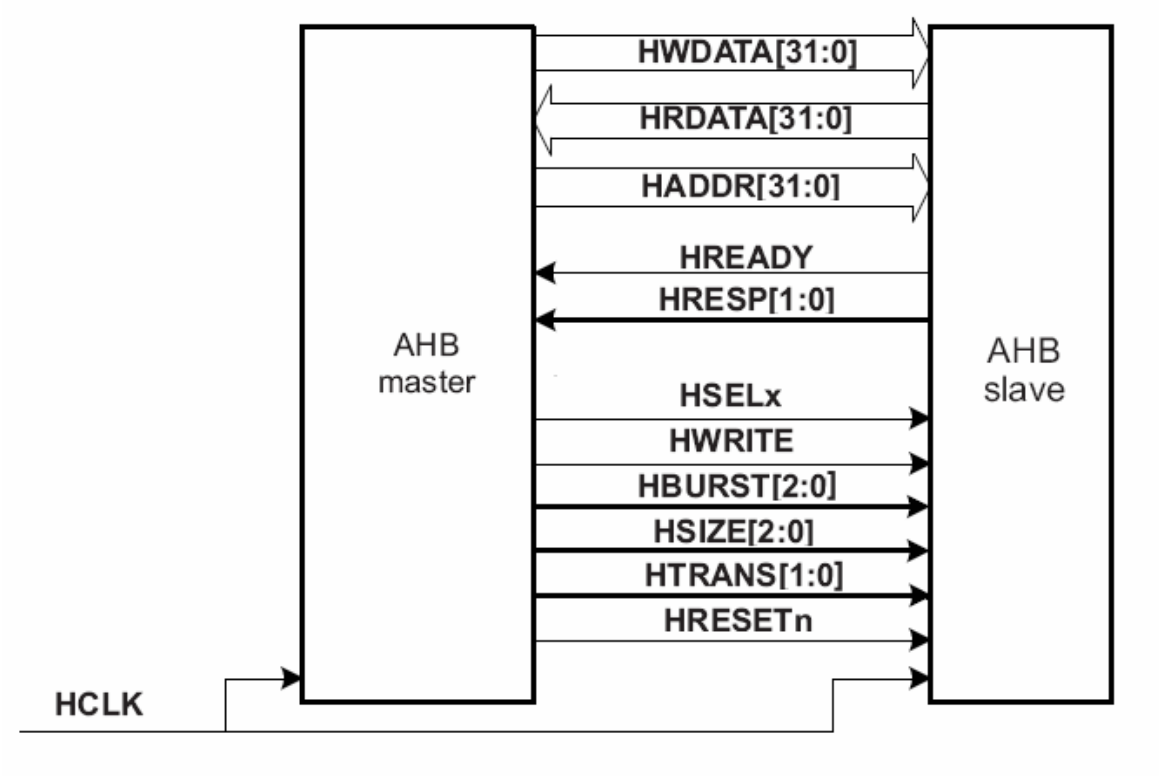


圖3-8 AHB 主模組端及從模組端間溝通主要訊號

3.6 RTEMS嵌入式即時作業系統

RTEMS[14]誕生於 20 世紀 80 年代，最早用於美國國防部的導彈控制系統，它在誕生時就瞄準了高即時性和高可用性的標準，不但有非常好的即時性和穩定性，也具有非常好的可重複使用性。當時 RTEMS 的全名是“即時導彈控制執行體”。現在 RTEMS 的全名則是“多處理器即時操作

系統”，是一個純粹的 32 位元系統（由於充分利用了 32 位的特性，所以無法像 uCOSII 那樣用於 8 位和 16 位元系統）。經過了 20 多年的持續開發，RTEMS 擁有非常廣泛的客戶，也具有豐富的開發資源。尤其在即時性、穩定性、開發速度和多處理器支援上面都是非常優秀的。

即時系統是一種相當特殊的系統，他在任務的分派上比普通作業系統如 Linux 和 Windows 有更為嚴格的要求。即時系統最為關鍵的特性是他的即時性，所謂即時性指的是能夠在規定的時間內處理外部的資訊。有時候需要處理的資訊可能是突發的大量資訊，既可能是一串毫不相關的資訊，也可能是一些相互關聯具有同步關係的資訊，還可能是迴圈資訊。

即時嵌入式系統裏任務的即時性可以分為硬即時（Hard Real-time）和軟即時（Soft Real-time），區分的標準是看任務在規定時間裏面沒有完成的時候的返回值。硬即時任務的規定時間到來的時候如果沒有結束沒有返回值，或者返回的值會導致系統異常。軟即時則在任務沒有結束的時候會有可供系統處理的返回值。

即時系統的另外一個特性是管理協調大規模併發行程的能力。任務是併發的，但是處理器以及硬體資源有限，指令只能一條一條的執行。雖然在現在的作業系統理論中提出了很多多行程併發執行的方法，但是多個行程間的同步和非同步處理仍然讓大多數設計者頭疼，如果有了 RTOS，設計者的工作就簡單了很多。在 RTEMS 中一個執行程式可以包含了若干的行程，每個行程都是可控的。每個行程在邏輯上都是同步執行。系統元件

提供了為多個行程管理協調硬體資源的能力。同時，行程管理元件也能接受硬體中斷以及驅動程式中斷。經過 RTEMS 的擴展，即時系統的設計變得簡單。如果有多個處理器，或者一個處理器中有多個核心（比如 ARM+DSP），設計就變得更為麻煩。多個處理器間的通訊以及多個處理器間的資源分享需要 RTOS 進行調度，在這一點上面，RTEMS 做得比較好。RTEMS 的多處理器擴展能協調分佈在不同處理器上面的任務。透過擴展，系統的即時性更強，更加有效，也更為可靠。

RTEMS 提供了多種 API 的介面，除了 RTEMS 本身的 API 以外，還提供了 POSIX 介面（Linux 以及 VxWorks 的程式能方便移植），pSOS 的 API 以及 ITRON 的 API。這些 API 能讓軟體設計者擺脫對多工和多處理器控制的底層細節。此外 RTEMS 還提供記憶體管理、訊息管理、互斥變數管理、GDB 除錯等一系列的 API。這些 API 使用戶能專心開發應用程式，大大提高了效率。

第四章　Ogg/Vorbis 解碼器實作

4.1 軟體模擬與系統架構

LEON2 是一個可組態的處理器，依照不同的需求來增加或減少內部的模組。此小節將探討如何找出一個最適合我們的設定。要達到能夠即時地播放 Ogg Vorbis 檔，我們必須先分析整個解碼流程中每個模組的特性，並且以此當做軟硬體 partitioning 的主要根據。除此之外也同時計算解碼時所消耗的記憶體量。當此步驟完成後再以 TSIM 模擬器進行模擬驗證整個演算法是否達到預期的結果。

首先在工作站上透過執行一個簡單範例 iVorbisfile_example 來大概評估它所需要的資源。

```
top - 22:29:37 up 7 days, 18:49,  2 users,  load average: 0.18, 0.22, 0.24
Tasks:  42 total,   2 running,  40 sleeping,   0 stopped,   0 zombie
Cpu(s):   1.0% user,  20.7% system,   0.0% nice,  78.3% idle
Mem:    191888k total,   179168k used,    12720k free,     8752k buffers
Swap:   465844k total,    15400k used,   450444k free,    42036k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
23903 root      19   0   416  416  248 S  3.7  0.2   0:00.11 vorbisfile_exam
 7755 root      14   0  4908 4900 2872 S  1.7  2.6   0:22.61 svn
 8945 baga      17   0  1936 1892 1684 S  1.0  1.0   0:51.05 sshd
 7703 baga       9   0  1920 1876 1680 R  0.3  1.0   0:03.27 sshd
    1 root       9   0   504  476  452 S  0.0  0.2   0:02.25 init
    2 root       9   0     0    0    0 S  0.0  0.0   0:00.01 keventd
    3 root      19  19     0    0    0 S  0.0  0.0   0:01.31 ksoftirqd_CPU0
    4 root       9   0     0    0    0 S  0.0  0.0   0:26.48 kswapd
    5 root       9   0     0    0    0 S  0.0  0.0   0:00.00 bdflush
    6 root       9   0     0    0    0 S  0.0  0.0   0:05.23 kupdated
   53 root      10   0     0    0    0 S  0.0  0.0   0:18.66 kjournald
  415 root       9   0     0    0    0 S  0.0  0.0   0:00.00 khubd
  713 root       9   0   652  640  540 S  0.0  0.3   0:03.89 dhclient
  797 root       9   0   612  612  520 S  0.0  0.3   0:03.93 syslogd
  800 root       9   0  1224 1224  472 S  0.0  0.6   0:00.10 klogd
  852 root       9   0   520  508  436 S  0.0  0.3 186:24.89 vmware-guestd
  883 Debian-e   8   0  1488 1404 1260 S  0.0  0.7   0:00.90 exim4
```

圖 4-1 解碼所消耗資源

從圖 4-1 可以粗略的算出執行時所需的資源，測試的平台為 Intel Pentium-II 500，256MB RAM，作業系統為 Debian，核心版本是 2.4.27。從 top 的結果顯示，執行時使用了 3.7%的 CPU 時間，記憶體使用量大約為 1.54MB(256 * 0.6%)。

從記憶體使用量來看，大概至少需要 1.54MB，但是此為純粹的解碼程式，我們的撥放程式還加入了 RTEMS 的即時作業系統以及具備 POSIX 多執行緒的能力，除此之外，我們打算把 Ogg Vorbis 的資料放入記憶體中，讓程式可以更方便的讀取欲被解碼的 Ogg Vorbis 資料，藉此減少程式撰寫上的複雜度。所以把測試的結果當成所需記憶體的最低限度，再加上以上所述幾項的要求，初步估記將至少需要 2MB 以上的記憶體容量才能在真正的硬體上實現即時播放的效果。

但在 CPU 時脈上的分析較為困難。主要原因是我們採用的 LEON2 處理器是屬於 Sparc V8 架構，與測試平台上的 CPU 有著相當大差異存在。雖然經過之前的測試結果，CPU 的時脈大約為 500 MHZ * 3.7% = 18MHz，不過這只能當做參考。測量時脈比較準確的方法是 CPU 在同樣架構下所跑的真正速度。所幸在網路上可以找到類似的參考資料。在 Vorbis-dev 的 mailing list 裡有人成功地在一顆 ARM 的處理器上實現了 Ogg Vorbis 的解碼器，該處理器的時脈為 68MHz，播放時的使用率為 100%。

在了解程式執行時所需要資源後，可知單純將此軟體移至實際的硬體上是無法滿足即時播放的要求，因此我們將尋求其它增進解碼速度的方法。方法主要可以分成針對軟體（改善演算法、程式碼或編譯時所下的旗標）或著是硬體（處理器、把程式中某部分改用硬體來執行）來做最佳化。若要從硬體這方面著手，我們必須先知道軟體中哪個部分消耗了較多的 CPU 時間並經過複雜度分析之後才能決定是否要把它用硬體來實現。

GNU 提供了一個稱為 gprof[15]的工具來幫助我們完成複雜度分析的任務。只要在重新編譯時加上 -pg 這個旗標，所生成的程式即可拿來做程式執行 profiling 的動作。結果分成兩部分，其中 flat profile 告訴我們每個函式執行各花了多少時間以及被呼叫了多少次。另外，call graph 則是包含了每個函式被誰呼叫，或是呼叫了哪些函式及總共的呼叫次數。

Flat Profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
31.58	**0.12**	**0.12**				**mdct_backward**
15.79	0.18	0.06				mdct_butterfly_generic
10.53	0.22	0.04				Vorbis_book_decodevv_add
7.89	0.25	0.03				_Vorbis_apply_window
7.89	0.28	0.03				mdct_bitreverse
7.89	0.31	0.03				render_line
5.26	0.33	0.02				decode_packed_entry_number
5.26	0.35	0.02				mapping0_inverse

5.26	0.37	0.02	oggpack_look
2.63	0.38	0.01	mdct_butterfly_32
0.00	0.38	0.00	main

Call Graph

granularity: each sample hit covers 4 byte(s) for 2.63% of 0.38 seconds

index	% time	self	children	called	name
		0.00	**0.00**		**mapping0_inverse [8]**
[1]	**31.6**	**0.12**	**0.00**		**mdct_backward [1]**
		0.00	**0.00**		**mdct_butterfly_generic [2]**
		0.00	**0.00**		**mdct_butterfly_32 [10]**
		0.00	**0.00**		**mdct_bitreverse [4]**
		0.00	0.00		mdct_backward [1]
[2]	15.8	0.06	0.00		mdct_butterfly_generic [2]
					<spontaneous>
[3]	10.5	0.04	0.00		Vorbis_book_decodevv_add [3]
		0.00	0.00		decode_packed_entry_number [7]
		0.00	0.00		mdct_backward [1]
[4]	7.9	0.03	0.00		mdct_bitreverse [4]
					<spontaneous>
[5]	7.9	0.03	0.00		render_line [5]
					<spontaneous>
[6]	7.9	0.03	0.00		_Vorbis_apply_window [6]
		0.00	0.00		Vorbis_book_decode [452]
		0.00	0.00		Vorbis_book_decodevs_add [455]
		0.00	0.00		Vorbis_book_decodev_add [453]
		0.00	0.00		Vorbis_book_decodev_set [454]
		0.00	0.00		Vorbis_book_decodevv_add [3]
[7]	5.3	0.02	0.00		decode_packed_entry_number [7]

```
                0.00    0.00                    oggpack_look [9]
                0.00    0.00                    oggpack_adv [299]
-----------------------------------------------

                                                <spontaneous>
[8]     5.3     0.02    0.00            mapping0_inverse [8]
                0.00    0.00                    memset [237]
                0.00    0.00                    mdct_backward [1]
-----------------------------------------------

                0.00    0.00            decode_packed_entry_number [7]
[9]     5.3     0.02    0.00            oggpack_look [9]
-----------------------------------------------

                0.00    0.00                    mdct_backward [1]
[10]    2.6     0.01    0.00            mdct_butterfly_32 [10]
                0.00    0.00                    mdct_butterfly_16 [227]
-----------------------------------------------
```

從 profiling 的結果可知,，mdct_backward 函式 (即 inverse MDCT) 執行時間總共占了全部的 31.6%。因此我們將針對此函式來作進一步的分析，並評估如何用硬體的方式來實現其演算法。

4.2 系統架構介紹

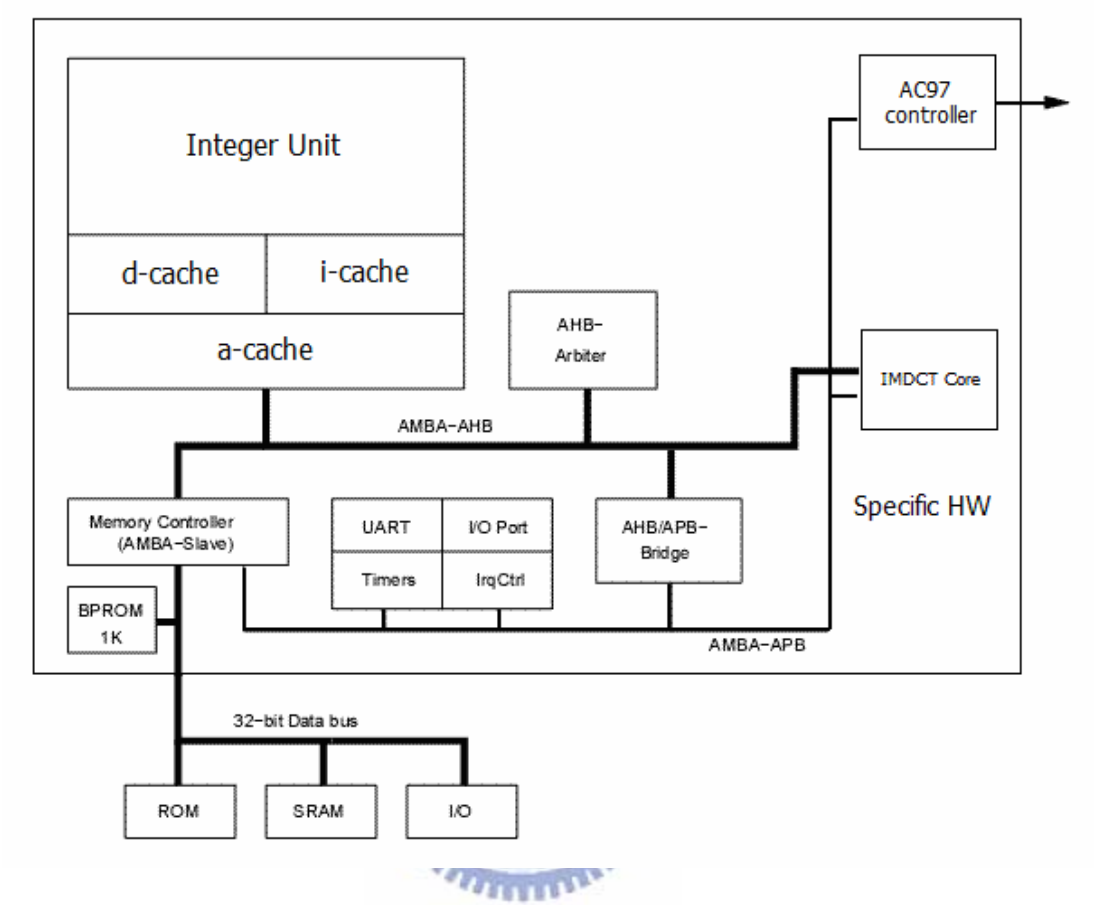


圖 4-2 系統架構

圖 4-2 為整個解碼器的系統架構，主要是由 LEON2 處理器，IMDCT core，AC97 core，memory controller 組成。在系統中利用 Gaisler LAB 所提供的 DSU 除錯與下載介面模組，將 Ogg Vorbis 檔案在電腦上使用 C 處理後，輸出為一個陣列。並利用電腦的 COM1 port 把資料附在程式中，下載到 SRAM 裡，之後再對 SRAM 抓取所需的資料來做運算，且也在 SRAM 中規劃一塊空間 Ogg Vorbis 音訊資料，剩餘的空間則當做每一模組運算後的儲存空間以及最後的 PCM 信號輸出的 buffer。

本系統的工作頻率設定在 27MHz，因此在解碼過程中必須掌握系統中每個部份所需花費時間和各個模組之間的整合，以及考慮各個模組所需花的 gate count 數目，在有限的系統資源下達到最大的效能。

4.3 硬體實現部分

接下來介紹如何把 IMDCT 演算法用硬體來實現，並且加入 AC97 的控制器。

4.3.1 MDCT 演算法

許多較先進的音訊編碼器都使用到 MDCT，像是 MPEG-1 Layer-III， Dolby AC/3， MPEG AAC，當然也包含了 Ogg Vorbis。MDCT 是一種線性正交重疊轉換法，最早是由 Princen 在 1986 年所提出來的 [16]。

假設 $X_t(k)$ 是在頻域上的某個取樣，$x_t(k)$ 則代表是在時域上，n 是 Frame 的大小，t 為 Frame 的個數，$f_t(k)$ 為視窗係數，則 MDCT 及 IMDCT 的主要公式可以表示成：

MDCT：

$$X_t(m) = \sum_{k=0}^{n-1} f_t(k) x_t(k) \cos\left(\frac{\pi}{2n}(2k+1+\frac{n}{2})(2m+1)\right)$$

$$(m = 0,\ldots,\frac{n}{2}-1)$$

IMDCT：

$$y_t(p) = f_t(p)\frac{n}{4}\sum_{m=0}^{\frac{n}{2}-1} X_t(m)\cos\frac{\pi}{2n}(2p+1+\frac{n}{2})(2m+1)$$

$$(p = 0,...,n-1)$$

$$f_{t-1}(\frac{n}{2}-k)^2 + f_t(k)^2 = 1$$

$$(k = 0,...,\frac{n}{2}-1)$$

透過重疊方式來達到時域混疊消除的效果：

$$\tilde{x}_t = y_{t-1}(q+\frac{n}{2}) + y_t(q) \quad \text{for} \quad q = 0...\frac{n}{2}-1$$

接著介紹 MDCT 的四種窗框：長窗框（Normal Window）、長短窗框（Start Window）、短窗框（Short Window）與短長窗框（Stop Window），如圖 4-3(a)~(d)所示。使用長窗框做轉換，可得較好的頻譜解析度，而使用短窗框做轉換，則可得較佳的時間軸解析度。至於長短窗框是用於長窗框要轉換到短窗框時的過渡窗框，短長窗框則相反，四種窗框的切換方式如圖 4-3(e)所示。5-x 式為長短窗框之間轉換的公式：

$$f_{t-1}(\frac{n}{2}+k)^2 + f_t(k)^2 = 1 \quad \text{for} \quad k = 0...\frac{n}{2}-1$$

且每個窗框必須為左右對稱：

$$f_t(k)^2 + f_t(\frac{n}{2}-k)^2 = 1 \quad \text{for} \quad k = 0...\frac{n}{2}-1$$

$$f_t(k+\frac{n}{2})^2 + f_t(n-1-k)^2 = 1 \quad \text{for} \quad k = 0...\frac{n}{2}-1$$

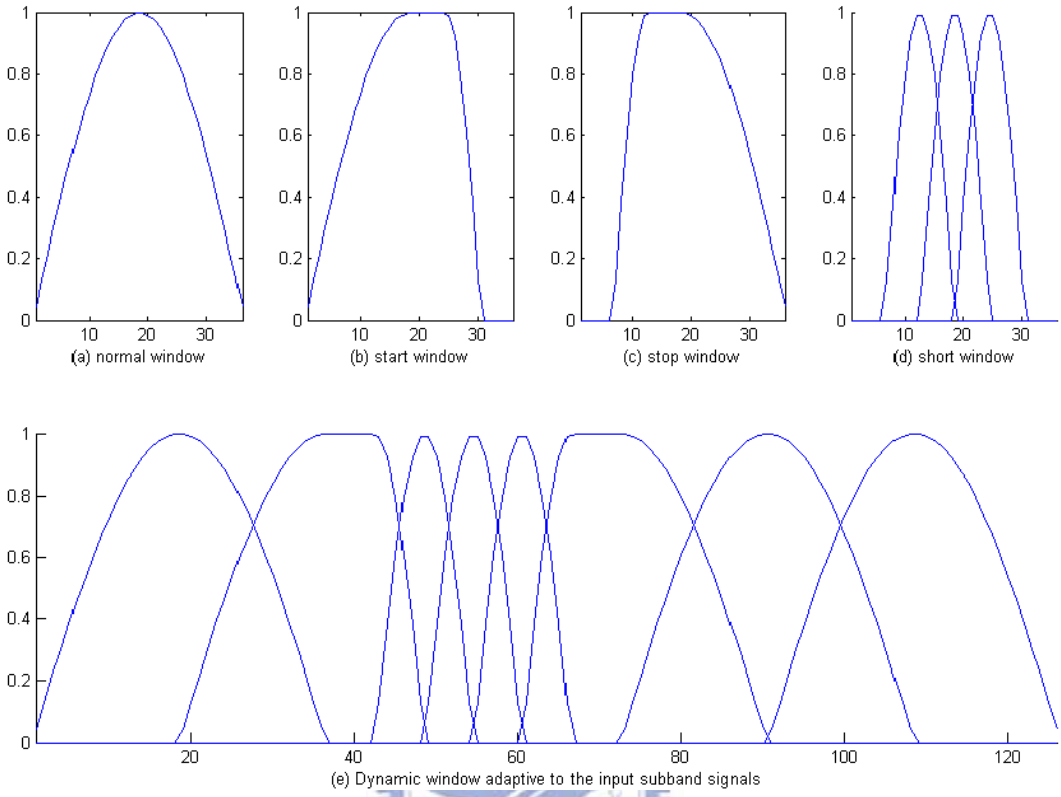


圖 4-3 MDCT 的各種窗框

在最近這數十年來，已經發展出不少用來計算 MDCT 的快速演算法。特別是在音訊編碼上，大多數都是採用 FFT 或是 DCT 為基礎的演算法。而除了上述的演算法外之，也有人提出了遞迴的方式來計算 MDCT。

FFT-based 的演算法一開始先把 n 點的 MDCT 係數轉換成 n/2 點的 DFT 係數，然後再使用 FFT 來計算，不過需要較多的 RAM。DCT-based 的演算法前半步驟與 FFT-based 的相同，不過後半段是用遞迴的 butterfly 來運算，如圖 4-4。

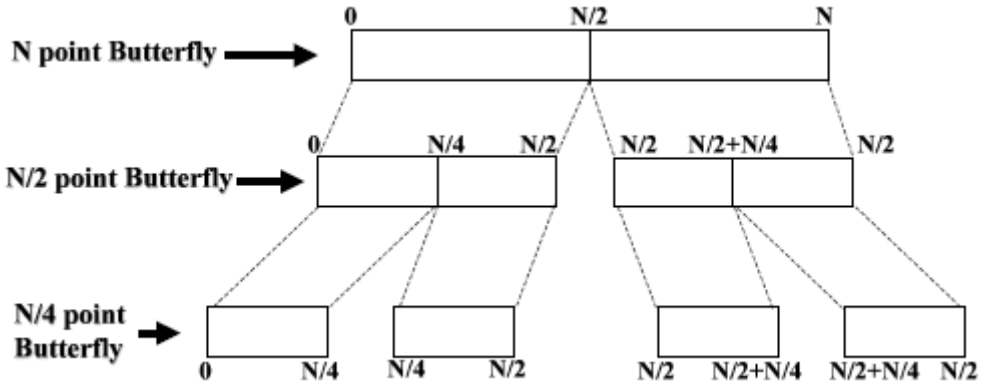


圖 4-4 遞迴 butterfly 運算

在 Ogg Vorbis 中是採用了 Sporer[17]所提出的演算法，用之前所敘述的 butterfly 來減少運算的時間，不過在資料的儲存及定址的過程較為複雜。

4.3.2 IMDCT 模組分析

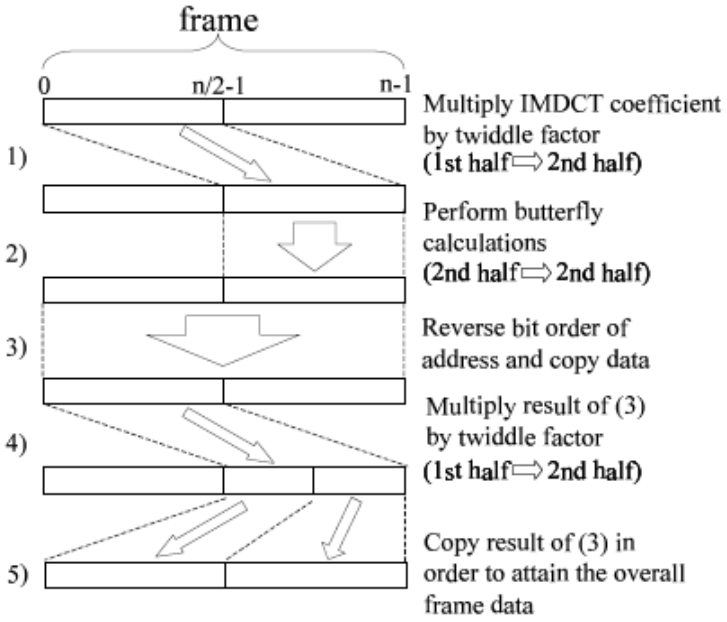


圖 4-5 IMDCT 計算過程[18]

在決定把 IMDCT 用硬體的方法來實作之後，接下來我們必需先將原始碼中的 backward_mdct 函式做分析。在 Sporer 所提出的演算法中，把整個 MDCT 及 IMDCT 的過程分成八個步驟，不過我們可以根據實際的計算過程依功能分成五個步驟，以下是兩者之間的關聯性，並且將 Sporer 演算法相對應的部分列出，原始碼請參照附錄 B。

1) pre-twiddling：把資料的前半段乘以 twiddle 因子，只需用到加法及減法，結果放在後半段。此步驟在 mdct_backward()完成。而在 Sporer 演算法中的步驟為：

STEP 1：

for $k = 0..\frac{n}{4}-1$

$$v_{n-4k-1} = (u_{4k} - u_{n-4k-1})A_{2k} - (u_{4k+2} - u_{n-4k-3})A_{2k+1}$$

$$v_{n-4k-3} = (u_{4k} - u_{n-4k-1})A_{2k+1} - (u_{4k+2} - u_{n-4k-3})A_{2k}$$

STEP 2：

for $k = 0..\frac{n}{8}-1$

$$w_{\frac{n}{2}+3+4k} = v_{\frac{n}{2}+3+4k} + v_{4k+3}$$

$$w_{\frac{n}{2}+1+4k} = v_{\frac{n}{2}+1+4k} + v_{4k+1}$$

$$w_{4k+3} = (v_{\frac{n}{2}+3+4k} - v_{4k+3})A_{\frac{n}{2}-4-4k} - (v_{\frac{n}{2}+1+4k} - v_{4k+1})A_{\frac{n}{2}-3-4k}$$

$$w_{4k+1} = (v_{\frac{n}{2}+1+4k} - v_{4k+1})A_{\frac{n}{2}-4-4k} + (v_{\frac{n}{2}+3+4k} - v_{4k+3})A_{\frac{n}{2}-3-4k}$$

此步驟是把輸入向量的前半段乘以 twiddle 因子，並將結果依照一特定的順序存入輸出向量的後半段。只需要用到乘法及加法。參考圖 4-6。

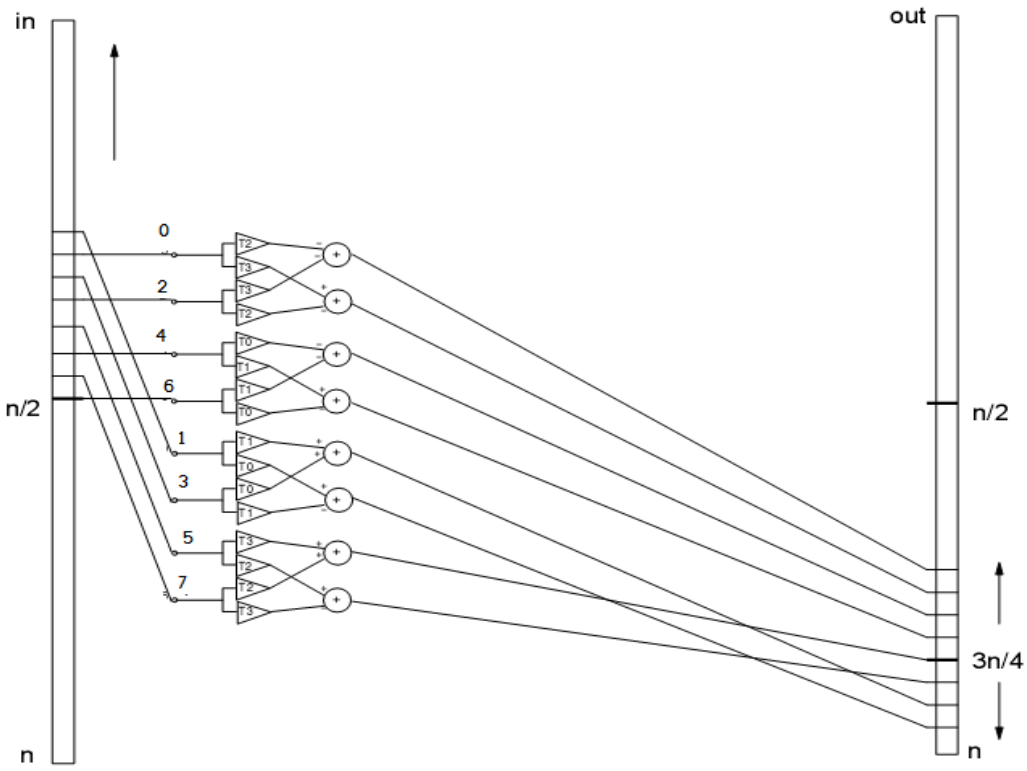


圖 4-6 pre-twiddling

一開始先使用其中 8 個元素當成輸入，從 n/2-7 到 n/2。第奇數個元素(n/2-7, n/2-5, n/2-3, n/2-1)，依圖 4-6 的方法來計算。結果存至 3n/4 到 3n/4+3。第偶數個元素(n/2-6, n/2-4, n/2-2, n/2)，則依圖 4-6 的方法，結果存至 3n/4-4 到 3n/4-1。然後處理下八個元素，即從 n/2-15 到 n/2-8，奇數部分放至 3n/4+4 到 3n/4+7，偶數部分則是 3n/4-8 到 3n/4-5，以下依此類推。由以上的步驟可觀察出偶數部分在 Output 從 3n/4 的位置往上移動至 n/2，相對的奇數則是從 3n/4 向下移動到 n。當分別到達 n/2 及 n 時表

示 pre-twiddling 已完成。

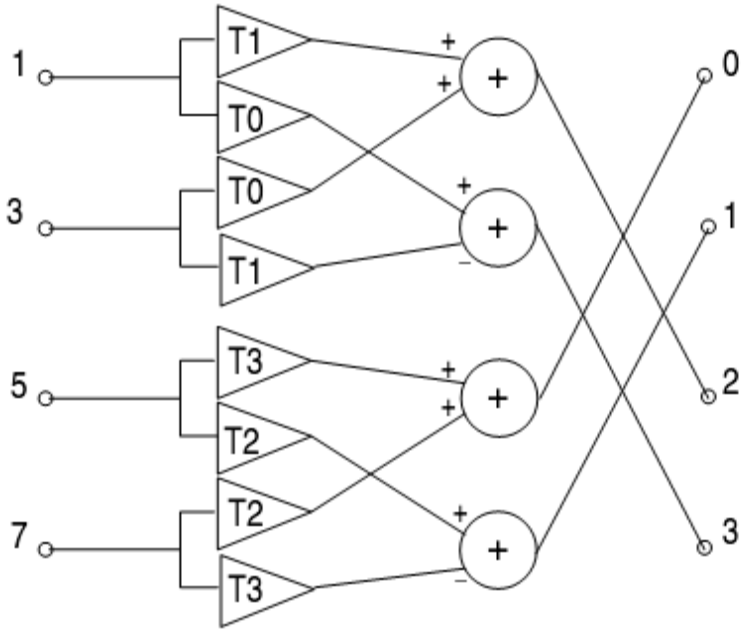


圖 4-7 奇數 pre-twiddling

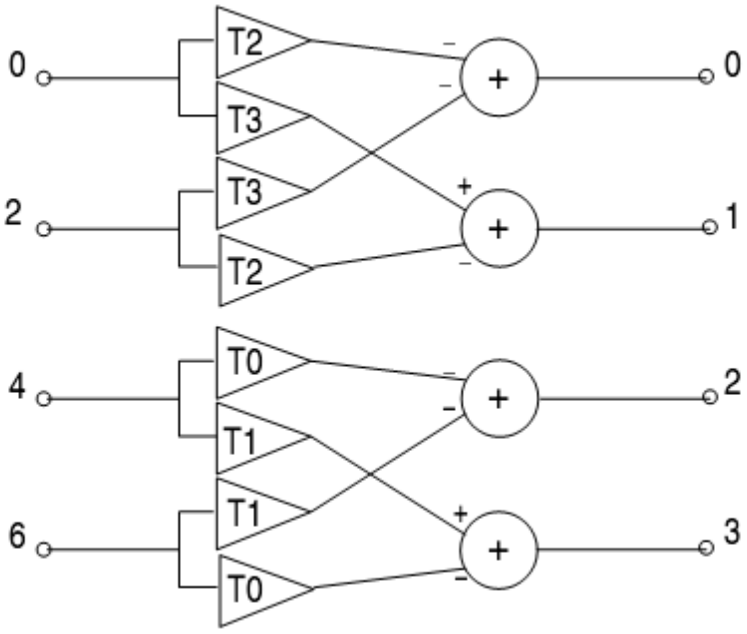


圖 4-8 偶數 pre-twiddling

一開始先使用其中 8 個組成部分當成輸入，從 n/2-7 到 n/2。第奇數個元素(n/2-7, n/2-5, n/2-3, n/2-1)，依圖 4-6 的方法來計算。結果存至 3n/4 到 3n/4+3。第偶數個元素(n/2-6, n/2-4, n/2-2, n/2)，則依圖 4-6 的方法，結果存至 3n/4-4 到 3n/4-1。

回到圖 4-6，其中三角形部分表示的是乘法器。即把輸入乘以三角形內標示的常數 Trig。圓形部分代表加法器，兩個輸入各有其正負號。例如要計算第三個元素的話，先把輸入 1 乘以 T0 再減去輸入 3 乘以 T1。圖 4-7 及圖 4-8 之間有些許差異，必須注意。

2) butterfly 運算：資料的後半段用 butterfly 遞迴運算，使用到的函式有：

- mdct_butterfly_8()：8 個點 butterfly 運算
- mdct_butterfly_16()：16 個點 butterfly 運算
- mdct_butterfly_32()：32 個點 butterfly 運算
- butterfly_generic()：N/2～64 個點 butterfly 運算
- mdct_bitreverse()：Bitreverse 運算

STEP 3：

ld 表示以 2 為底的對數

for $l = 0..ld(n) - 4 \quad k_0 := \frac{n}{2^{l+2}} \quad k_1 := 2^{l+3}$

for $r = 0..\frac{n}{2^{l+4}} - 1$ and $s = 0..2^{l+1} - 1$

$$\hat{u}_{n-1-k_0 2s-4r} = w_{n-1-k_0 2s-4r} + w_{n-1-k_0(2s+1)-4r}$$

$$\hat{u}_{n-3-k_0 2s-4r} = w_{n-3-k_0 2s-4r} + w_{n-3-k_0(2s+1)-4r}$$

$$\hat{u}_{n-1-k_0(2s+1)-4r} = (w_{n-1-k_0 2s-4r} + w_{n-1-k_0(2s+1)-4r})A_{rk_1} - (w_{n-3-k_0 2s-4r} + w_{n-3-k_0(2s+1)-4r})A_{rk_1+1}$$

$$\hat{u}_{n-3-k_0(2s+1)-4r} = (w_{n-3-k_0 2s-4r} + w_{n-3-k_0(2s+1)-4r})A_{rk_1} + (w_{n-1-k_0 2s-4r} + w_{n-1-k_0(2s+1)-4r})A_{rk_1+1}$$

經過 pre-twiddling 之後，output 的後半段會拿來當成 butterfly 運算的 Input。計算結果會再度存至 output 向量的後半段，總共要經過 m 次的 butterfly 運算。每經過一次 butterfly，輸入會變成一半，直到輸入變成 8 的時候才會停止。所以可知 m = ln(n/8)，其中 ln 是以 2 為底的對數。見圖 4-9。

每兩個 stage 之間，至少會有一個 butterfly 的輸入是兩個 8 個元素的群組。輸出也是兩個 8 個元素的群組。處理完這兩個群組的運算後，再往下兩個群組。同一個 Stage 中，下面的 butterfly 的結束點是在上面那個的開始點。由左往右每個 stage 的 butterfly 會以 2 的倍數遞增，相反

地，butterfly 的長度是以 2 的倍數遞減。當到達倒數第三個 stage 時，group1 到 group2 之間的距離會變成 0。

圖 4-10 是圖 4-9 裡橢圓部分的放大圖。最基本的 butterfly 有兩個輸入(8 個元素裡的其中之一)，兩個輸出(也是 8 個元素裡的其中之一)，同樣的只需要乘法及加法。

在 Sporer 的 Step 3 中，最後一個 stage 會使用到 8-point 的 butterfly 運算，其中只用到加法及減法器而不需要乘法累加器與 twiddle factor 表，而且 8-point 的 butterfly 運算能夠在單一個 cycle 裡完成。圖 4-11 為其計算的流程圖。

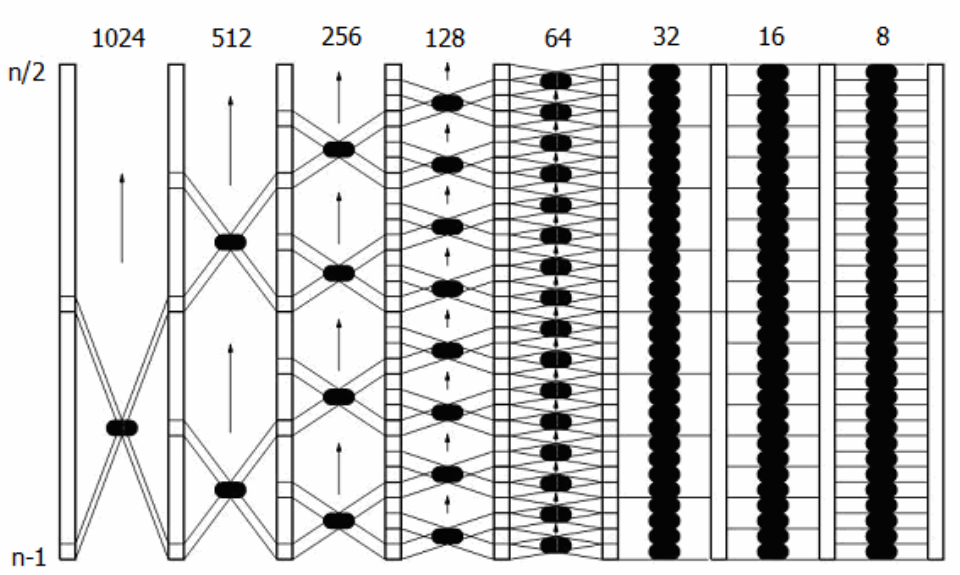


圖 4-9 n/2-point butterfly 運算

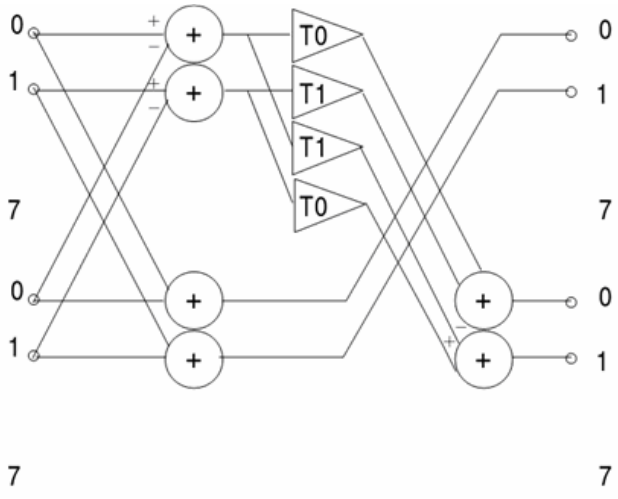


圖 4-10 16-point butterfly 運算

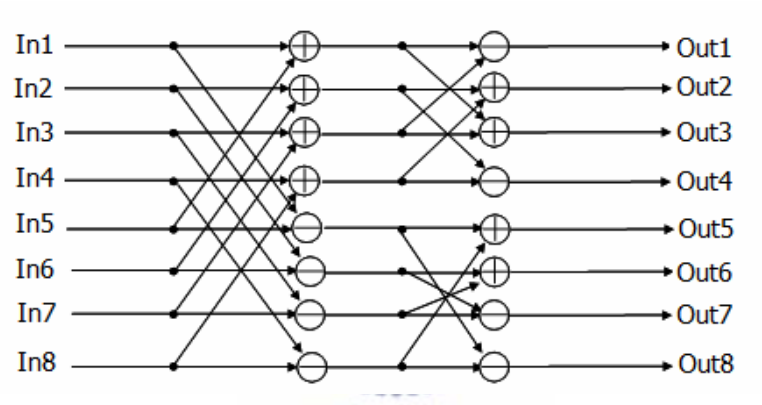


圖 4-11 8-point butterfly 運算

3) bit reversal：最後結果的前半段的計算方法是將原先資料的後半段的位元順序反轉並且乘上根據 block size 所求出的常數，使用到的函式為 bit_reverse。

STEP 4：

for $i = 1..\frac{n}{8} - 2$

j = BITREVERSE(i)

IF (i < j) THEN

$\hat{v}_{8j+1} = \hat{u}_{8i+1}$ $\hat{v}_{8i+1} = \hat{u}_{8j+1}$

$\hat{v}_{8j+3} = \hat{u}_{8i+3}$ $\hat{v}_{8j+3} = \hat{u}_{8i+3}$

$\hat{v}_{8j+5} = \hat{u}_{8i+5}$ $\hat{v}_{8j+5} = \hat{u}_{8i+5}$

$\hat{v}_{8j+7} = \hat{u}_{8i+7}$ $\hat{v}_{8j+7} = \hat{u}_{8i+7}$

STEP 5：

$\hat{w} = \hat{v}_{2k+1}$ for $k = 0..\frac{n}{2} - 1$

STEP 6：

for $k = 0..\frac{n}{8} - 1$

$\tilde{u}_{n-1-2k} = \hat{w}_{4k}$ $\tilde{u}_{n-2-2k} = \hat{w}_{4k+1}$

$\tilde{u}_{\frac{3n}{4}-1-2k} = \hat{w}_{4k+2}$ $\tilde{u}_{\frac{3n}{4}-2-2k} = \hat{w}_{4k+3}$

STEP 7：

for $k = 0..\frac{n}{8} - 1$

$$\tilde{v}_{\frac{n}{2}+2k} = (\tilde{u}_{\frac{n}{2}+2k} + \tilde{u}_{n-2-2k} + C_{2k+1}(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}) + C_{2k}(\tilde{u}_{\frac{n}{2}+2k+1} + \tilde{u}_{n-2-2k+1}))/2$$

$$\tilde{v}_{n-2-2k} = (\tilde{u}_{\frac{n}{2}+2k} + \tilde{u}_{n-2-2k} - C_{2k+1}(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}) - C_{2k}(\tilde{u}_{\frac{n}{2}+2k+1} + \tilde{u}_{n-2-2k+1}))/2$$

$$\tilde{v}_{\frac{n}{2}+1+2k} = (\tilde{u}_{\frac{n}{2}+1+2k-} + \tilde{u}_{n-1-2k} + C_{2k+1}(\tilde{u}_{\frac{n}{2}+1+2k} - \tilde{u}_{n-1-2k}) - C_{2k}(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}))/2$$

$$\tilde{v}_{n-2k-1} = (-\tilde{u}_{\frac{n}{2}+1+2k} + \tilde{u}_{n-1-2k} + C_{2k+1}(\tilde{u}_{\frac{n}{2}+1+2k} + \tilde{u}_{n-1-2k}) - C_{2k}(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}))/2$$

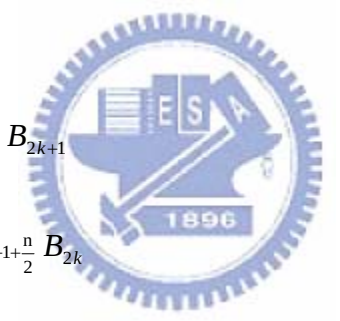
4) post-twiddling：把之前計算得到的結果前半段乘以 twiddle 因子後再存入後半段。

STEP 8：

for $k = 0..\frac{n}{4}-1$

$$X_k = \tilde{v}_{2k+\frac{n}{2}} B_{2k} + \tilde{v}_{2k+1+\frac{n}{2}} B_{2k+1}$$

$$X_{\frac{n}{2}-1-k} = \tilde{v}_{2k+\frac{n}{2}} B_{2k+1} - \tilde{v}_{2k+1+\frac{n}{2}} B_{2k}$$



Twiddle-Factors：

for $k = 0..\frac{n}{4}-1$

$$A_{2k} = \cos(\frac{4k\pi}{n}) \qquad A_{2k+1} = -\sin(\frac{4k\pi}{n})$$

for $k = 0..\frac{n}{4}-1$

$$B_{2k} = \cos(\frac{(2k+1)\pi}{2n}) \qquad B_{2k+1} = \sin(\frac{(2k+1)\pi}{2n})$$

for $k = 0..\frac{n}{8}-1$

$$C_{2k} = \cos(\frac{2(2k+1)\pi}{n}) \qquad C_{2k+1} = -\sin(\frac{2(2k+1)\pi}{n})$$

4.3.3 IMDCT 硬體模組架構

IMDCT 模組對於記憶體存取的頻率很高，因此如何有效管理記憶體是一個很重要的課題。如果記憶體的管理出了錯誤，則會造成整個運算結果的錯誤。 圖 4-12 為 IMDCT 模組的架構圖。

- AMBA 介面：負責把從 AHB 及 APB 匯流排上接收到的 CPU 發出的命令以及記憶體資料轉換成控制單元所需要的資料。
- 運算單元：根據 Ogg Vorbis 的演算法來對收到的資料做運算。
- 控制單元：由於 IMDCT 的五個計算模組有執行上的順序性，所以該控制電路事實上就是一個有限狀態機，擁有五個狀態，分別表示 IMDCT 的五個計算模組的執行順序。以 Verilog-HDL 的觀念來看，該控制電路就是最上層的一個模組，也就是說，整個 IMDCT 的資料流程全靠這個控制電路來掌握。

其工作流程如下：撥放程式把需要的資訊透過 AMBA APB 匯流排寫入相對應的暫存器的位址(區塊大小，輸入及輸出資料的位址，Trig 的位址等)。然後由 IMDCT 模組的 AMBA 介面把收到的資料再寫入實際的暫存器中。當控制單元收到開始的訊號後，先利用運算單元把真正需要的資料的位址算出。接著 AMBA 介面抓取資料存入控制單元的一個 buffer 中。此時控制單元就可把這些資料丟給運算單元，等到運算單元

完成它的任務後，便將結果存至另外一個 buffer，最後控制單元把這個 buffer 內的資料傳回 RAM 中。當所有的工作完成後，控制單元把自己的狀態設成等待，表示等待下一個區塊的到來。

在 IMDCT 模組在處理某個區塊時，它會把 AHB 匯流排的使用權鎖住，以防止其它 AHB 模組使用匯流排。即使是處理器本身也必須要等待，換句話說，此時程式會停止執行直到它結束任務。

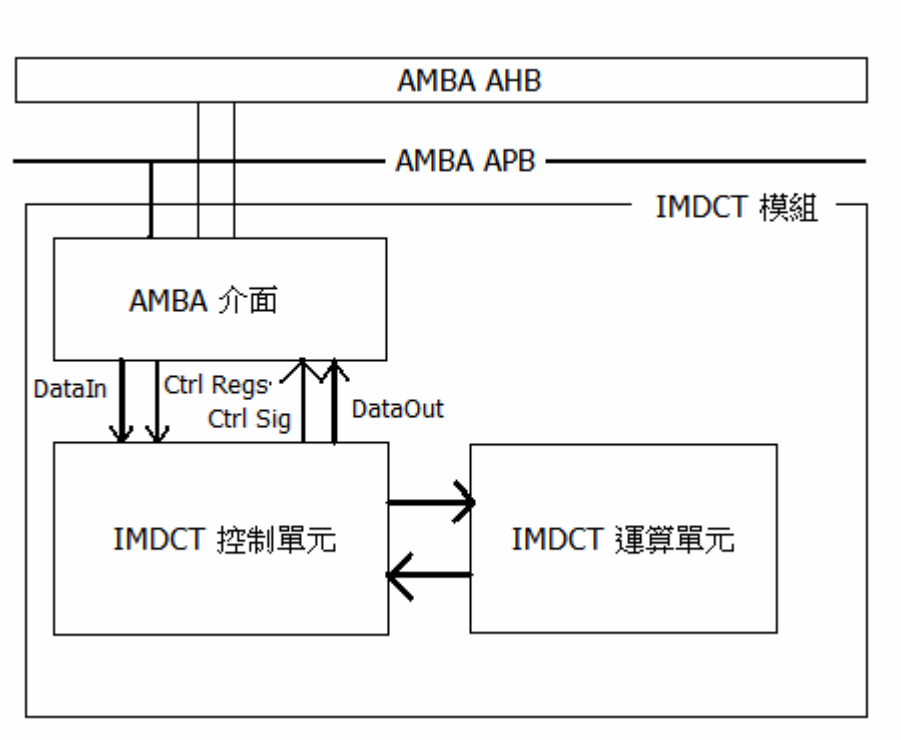


圖 4-12 IMDCT 模組示意圖

4.3.4 AC97 控制器

除了 IMDCT 模組之外，為了能夠把解碼後的 PCM sample 播放出來，我們也在 LEON2 中加入了 ac97 控制器，主要的功能是對板子上的

ac97 codec 做初始化的工作以及將從記憶體中收到解碼出來的 PCM sample 傳給 ac97 codec。在這控制器中，與 ac97 codec 相連接的共有五個訊號，如表 4-1，圖 4-13 為其架構圖。

表 4-1 AC97 控制器與 codec 連接訊號

名稱	I/O	描述
RESETn	I	active-low 的重置訊號。
BIT_CLK	I/O	頻率為 12.288MHz 的時脈訊號。
SYNC	I	48kHz 的同步訊號，表示開始接收或傳送 SDATA_OUT 及 SDATA_IN 上的資料，必須與 BIT_CLK 同步。
SDATA_OUT	I	包含了控制訊號以及 DAC 音訊資料，通常是在 BIT_CLK 為負緣時取樣。
SDATA_IN	O	包含了控制訊號以及 ADC 音訊資料，通常是在 BIT_CLK 為負緣時取樣。

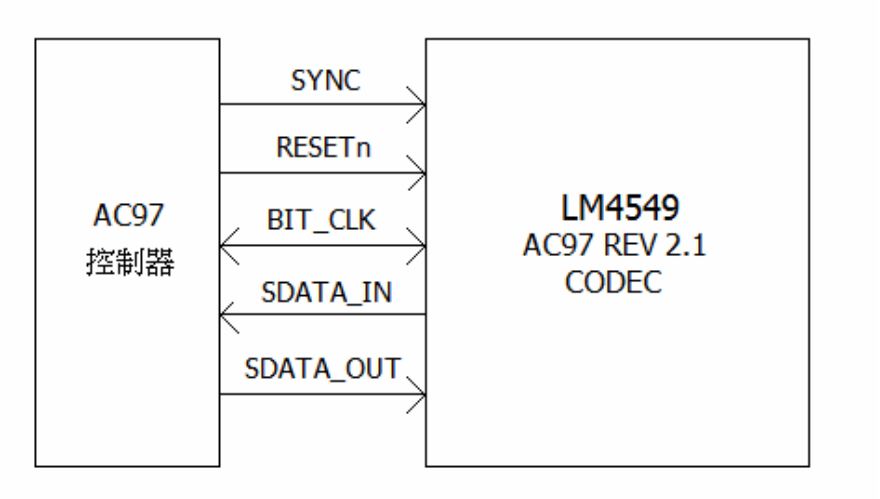


圖 4-13 AC97 控制器與 codec 連接架構圖

我們也在 LEON2 處理器中加入了五個暫存器來與 AC97 控制器做溝通，藉著這些暫存器就可對 codec 下達命令並且告知它 PCM sample 存在記憶體中的某個位址，接著進行播放的動作。以下為各暫存器所代表的意義：

- PCM 資料暫存器：AC97 控制器中有兩個 buffer，大小為 4096 個×16 位元的 sample（左右各一個 buffer）。AC97 會不斷產生中斷，而驅動程式裡中斷處理程式則是依其在 buffer 中的位址及 PCM 資料寫至這兩個暫存器，如圖 4-14。

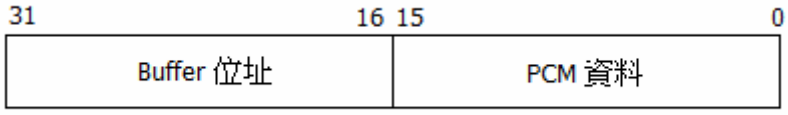


圖 4-14 PCM 資料暫存器格式

- 命令暫存器：此暫存器允許我們對 AC97 codec 下命令。例如寫入 0x180000（位址為 0x18）會把 PCM 的音量設成 0db

attenuation，如圖 4-15。

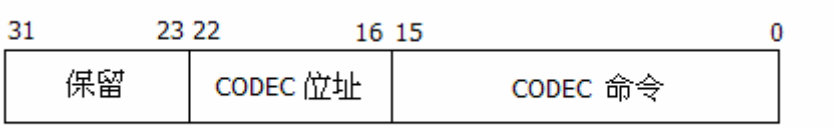


圖 4-15 命令暫存器格式

- 狀態暫存器：驅動程式可以透過檢查 BITCOUNT 的值來判斷是否 codec 正在讀取 PCM buffer，如果為是（BITCOUNT 的值為 0xF0 到 0xFF 之間），則驅動程式必須等到 buffer 的狀態為閒置時才能進行寫入的動作，如圖 4-16。

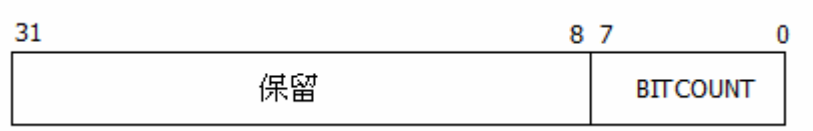


圖 4-16 狀態暫存器格式

- 除數暫存器：此暫存器是根據 sample rate 來計算出時脈：除數 = LEON2_CLOCK / SAMPLE_RATE + 1。舉例來說，若 LEON_CLOCK = 27Mhz，SAMPLE_RATE = 44,1Khz，然後經由驅動程式把 0x265 寫入此暫存器，如圖 4-17。

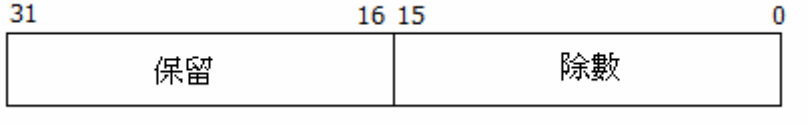


圖 4-17 除數暫存器格式

4.4 軟體實現部分

在此節會提到如何替 AC97 控制器撰寫在 RTEMS 上的驅動程式，以供播放程式來使用。另外，我們將利用 POSIX 多執行緒的功能，建造出一個可同時解碼以及播放音樂的播放程式。

4.4.1 AC97 驅動程式

在 LEON2 中加入 AC97 控制器後，大致上可以透過兩種方式來控制它。第一種當然就是直接針對暫存器做讀取寫入的工作，但是如果是利用這種方法，則當程式變得越來越複雜時，很容易就會搞混。另外一種方法是在 RTEMS 中加入 AC97 控制器的驅動程式，無論是對它初始化，或是寫入、讀取時，只需要利用它所提供的函式即可。就跟在 linux 作業系統一樣，把每種設備都當成是檔案來存取。大大地節省了開發應用程式的時間。

RTEMS 提供驅動程式位址表（Driver Address Table）來讓使用者加入額外 I/O 的驅動程式。驅動位址表用來通知 I/O manager 系統中每個設備驅動程式組態的 entry points。它的結構如圖 4-18：

- initialization：entry point 的位址，rtems_io_initialize 呼叫此位址上的函式來對相關的設備做初始化的動作。
- open：rtems_io_open 呼叫此位址上的函式進行開啟的動作。相對應於 C 語言中的 open 函式。

- close：rtems_io_close 呼叫此位址上的函式進行關閉的動作。相對應於 C 語言中的 close 函式。

```
typedef struct{
rtems_device_driver_entry initialization;
rtems_device_driver_entry open;
rtems_device_driver_entry close;
rtems_device_driver_entry read;
rtems_device_driver_entry write;
rtems_device_driver_entry control;
}rtems_driver_address_table;
```

圖 4-18 驅動程式位址表之結構

- read：rtems_io_read 呼叫此位址上的函式進行讀取的動作。相對應於 C 語言中的 read 函式。
- write：rtems_io_write 呼叫此位址上的函式進行寫入的動作。相對應於 C 語言中的 write 函式。
- control：rtems_io_contrl 呼叫此位址上的函式進行控制的動作。相對應於 C 語言中的 ioctl 函式。

圖 4-19 為寫入的流程圖，包括從應用程式呼叫 write 函式把 PCM 資料所在起始位址到結束位址寫入驅動程式，一直到 AC97 codec 播放出音樂為止。

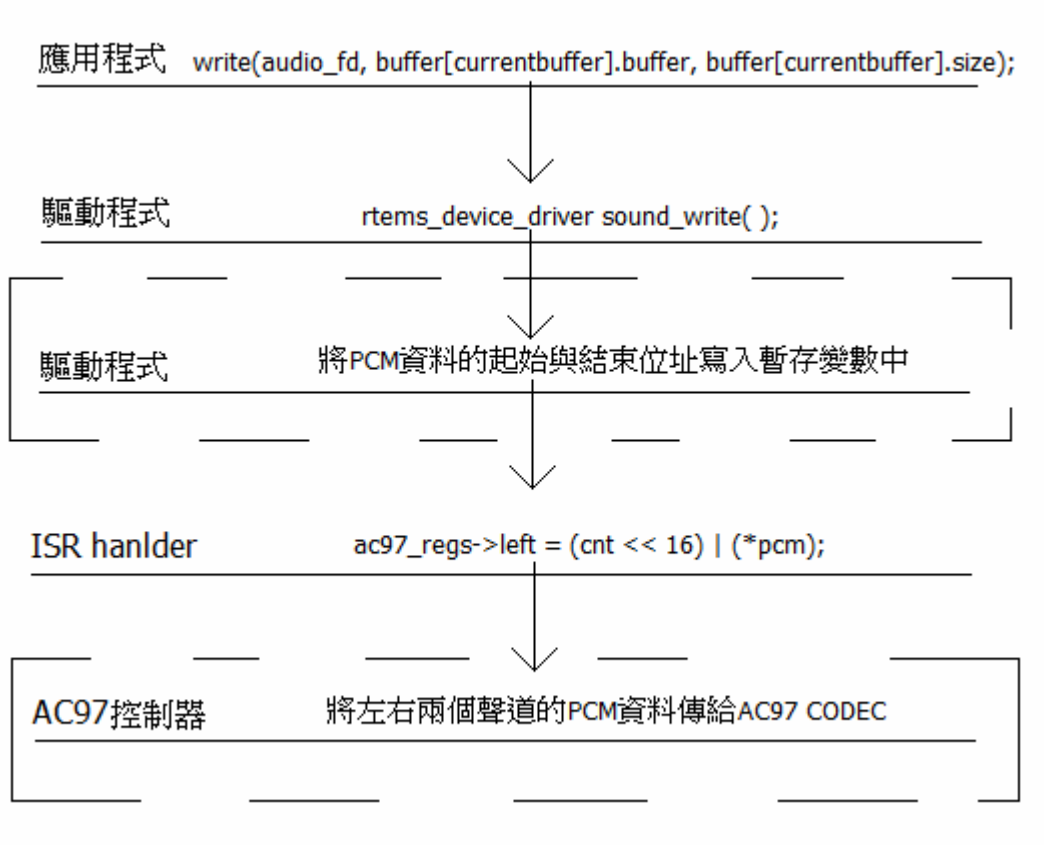


圖 4-19 寫入 PCM 資料流程圖

4.4.2 播放程式運作流程

在之前所提到的 RTEMS CCS 裡，已經包含了預先編譯好的函式庫。但是因為播放程式需要 POSIX 執行緒的支援，所以必須重新編譯，並且在編譯時，加入--enable-posix 的選項，所下的指令如下：

$ cd <原始碼所在目錄>

$./configure --target=sparc-rtems --enable-posix --prefix=<安裝位置>

$make all install RTEMS_BSP=”leon2”

一個典型的 RTEMS 應用程式可以選擇三種 API（application programming interfaces），分別是 POSIX 標準、RTEMS classic API 及 ITRON 3.0 API，我們的程式只使用了 POSIX 與 RTEMS classic API。播放程式主要分成兩個部分，除了原本的行程之外建立了一個執行緒，主要行程的任務為將 PCM 資料經由驅動程式的協助寫入到 AC97 控制器，而另一個執行緒負責 Ogg Vorbis 串流的解碼工作。兩者之間的關係以及運作的流程如圖 4-20。

4.5 系統驗證

當所有準備工作都完成後，即可進行最後的驗證工作。方法是先將播放程式在工作站上進行交叉編譯的動作，編譯完成後會得到一個能在 LEON2 上執行的檔案。至於 FPGA 上的設定是透過 SystemACE 界面，把合成好的.bit 檔案存在 CF 卡內，當板子開機會自動到 CF 卡內下載檔案並且將 FPGA 組態完成。然後我們可以利用 GRMON 工具裡的 DSU 將應用程式下載到板子上執行。表 4-2 為最後的執行結果，測試的檔案長度為 20 秒，從執行結果可知的確達到了即時播放的目的。

表4-2 工作站模擬與實際執行結果

	模擬結果	Multimedia Board
純軟體播放程式	30.1（秒）	34.7（秒）
加入 IMDCT 模組	＜20（秒）	＜20（秒）

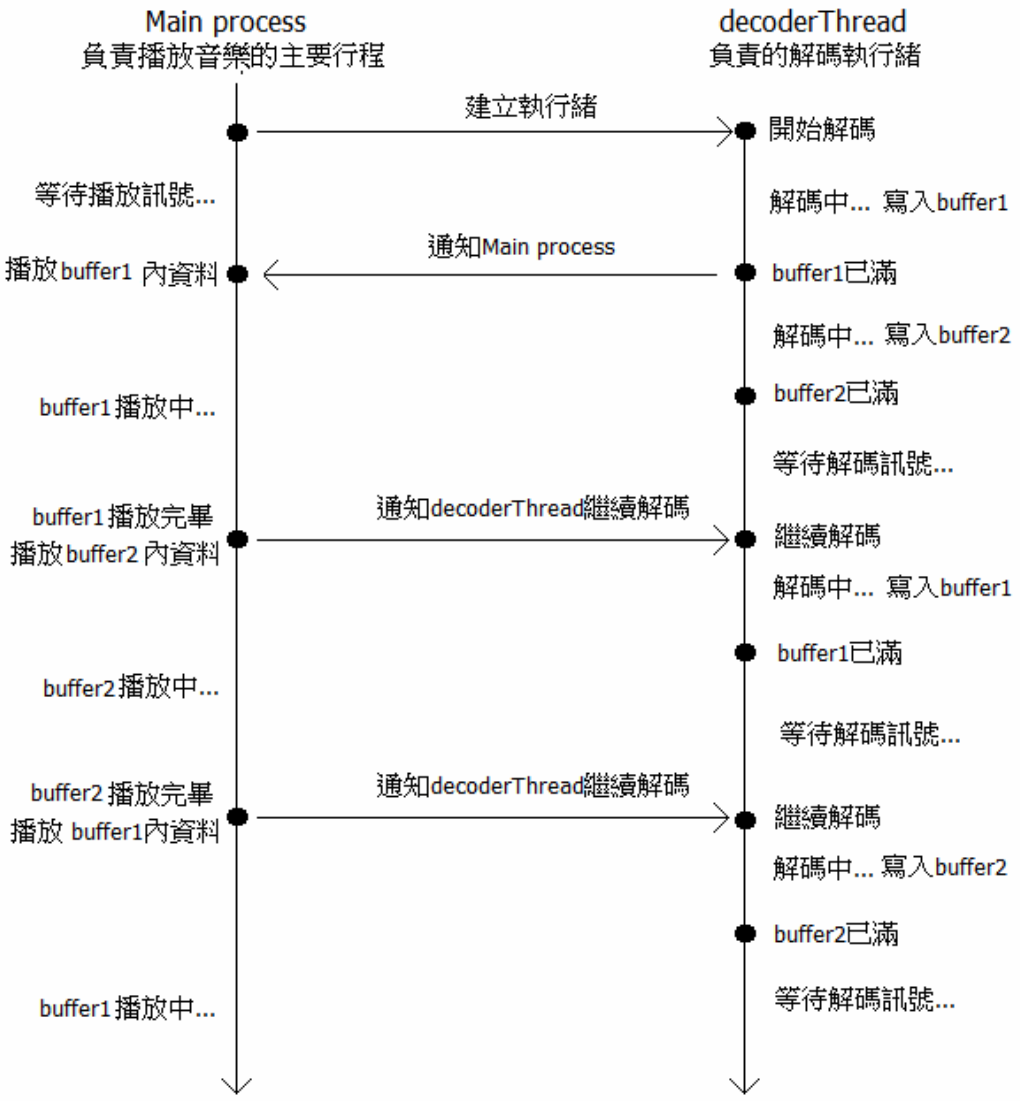


圖 4-20 播放程式運作流程

第五章　結論與未來工作

在本論文中提出了一個適用於Ogg Vorbis音訊解碼的IMDCT電路，搭配軟體完成了一個Ogg Vorbis解碼的工作。使用軟硬體共同設計後，這些原本龐大的計算量都交給硬體去做，進而達成即時播放的效果，雖然在音質上與原本在電腦上有一段差距，但因為時間上的關係，沒能找出其真正的原因。在加入了特定的硬體來改善運算效能後，LEON2本身處理器的負載也因此減輕，若此平台上還有其他的應用存在，則處理器也較有餘力來處理其他的工作。

至於在未來工作方面，將可考慮將此系統移植至別的硬體平台上或者是在原來的系統內加入檔案系統。本論文一開始使用的是另一塊FPGA板，也就是XILINX ML310，此塊板子具有較多的介面以及更強的功能，但是LEON2處理器本身並不支援板子上的DDR記憶體，而且在除錯上比較困難，所以增加了移植上的複雜度。在加入檔案系統這部分則是因為我們原本是將Ogg Vorbis的檔案附加應用程式裡，這方法有兩個缺點，第一是因為應用程式是透過RS232介面來傳輸，速度相當慢，每秒約在10KB左右，所以會使得下載的時間變得很久，拖慢了整體開發應用程式的時間。第二個缺點是會浪費額外的記憶體空間來儲存Ogg Vorbis的檔案。所以我們可以考慮使用板子上的CF卡介面來當做主要的檔案系統，或者是利用嵌入式作業系統的網路功能到遠方的伺服器端來抓取被解碼的資料。

Appendix A. Glossary of terms and abbreviations

bos page: The initial page (beginning of stream) of a logical bitstream which contains information to identify the codec type and other decoding-relevant information.

chaining (or sequential multiplexing): Concatenation of two or more complete physical Ogg bitstreams.

eos page: The final page (end of stream) of a logical bitstream.

granule position: An increasing position number for a specific logical bitstream stored in the page header. Its meaning is dependent on the codec for that logical bitstream and specified in a specific media mapping.

grouping (or concurrent multiplexing): Interleaving of pages of several logical bitstreams into one complete physical Ogg bitstream under the restriction that all bos pages of all grouped logical bitstreams MUST appear before any data pages.

lacing value: An entry in the segment table of a page header representing the size of the related segment.

logical bitstream: A sequence of bits being the result of an encoded media stream.

media mapping: A specific use of the Ogg encapsulation format together with a specific (set of) codec(s).

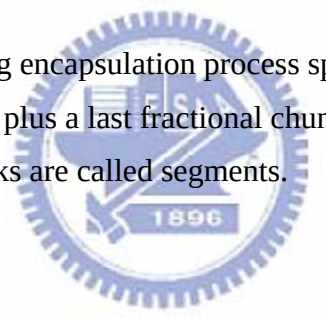
(Ogg) packet: A subpart of a logical bitstream that is created by the encoder for that bitstream and represents a meaningful entity for

the encoder, but only a sequence of bits to the Ogg encapsulation.

(Ogg) page: A physical bitstream consists of a sequence of Ogg pages containing data of one logical bitstream only. It usually contains a group of contiguous segments of one packet only, but sometimes packets are too large and need to be split over several pages.

physical (Ogg) bitstream: The sequence of bits resulting from an Ogg encapsulation of one or several logical bitstreams. It consists of a sequence of pages from the logical bitstreams with the restriction that the pages of one logical bitstream MUST come in their correct temporal order.

(Ogg) segment: The Ogg encapsulation process splits each packet into chunks of 255 bytes plus a last fractional chunk of less than 255 bytes. These chunks are called segments.



Appendix B. backward_mdct function

```
void mdct_backward(mdct_lookup *init, DATA_TYPE *in, DATA_TYPE *out){
   int n=init->n;
   int n2=n>>1;
   int n4=n>>2;
   /* rotate */
   DATA_TYPE *iX = in+n2-7;
   DATA_TYPE *oX = out+n2+n4;
   DATA_TYPE *T    = init->trig+n4;

   do{
      oX               -= 4;
      oX[0]            = MULT_NORM(-iX[2] * T[3] - iX[0]    * T[2]);
      oX[1]            = MULT_NORM (iX[0] * T[3] - iX[2]    * T[2]);
      oX[2]            = MULT_NORM(-iX[6] * T[1] - iX[4]    * T[0]);
      oX[3]            = MULT_NORM (iX[4] * T[1] - iX[6]    * T[0]);
      iX               -= 8;
      T                += 4;
   }while(iX>=in);

   iX                  = in+n2-8;
   oX                  = out+n2+n4;
   T                   = init->trig+n4;

   do{      T               -= 4;
      oX[0]            =    MULT_NORM (iX[4] * T[3] + iX[6] * T[2]);
      oX[1]            =    MULT_NORM (iX[4] * T[2] - iX[6] * T[3]);
      oX[2]            =    MULT_NORM (iX[0] * T[1] + iX[2] * T[0]);
      oX[3]            =    MULT_NORM (iX[0] * T[0] - iX[2] * T[1]);
      iX               -= 8;
      oX               += 4;
   }while(iX>=in);

   mdct_butterflies(init,out+n2,n2);
   mdct_bitreverse(init,out);

   /* roatate + window */
```

```
{
  DATA_TYPE *oX1=out+n2+n4;
  DATA_TYPE *oX2=out+n2+n4;
  DATA_TYPE *iX =out;
  T = init->trig+n2;

    /* post twiddling*/
  do{
    oX1-=4;

    oX1[3]   =   MULT_NORM (iX[0] * T[1] - iX[1] * T[0]);
    oX2[0]   = -MULT_NORM (iX[0] * T[0] + iX[1] * T[1]);

    oX1[2]   =   MULT_NORM (iX[2] * T[3] - iX[3] * T[2]);
    oX2[1]   = -MULT_NORM (iX[2] * T[2] + iX[3] * T[3]);

    oX1[1]   =   MULT_NORM (iX[4] * T[5] - iX[5] * T[4]);
    oX2[2]   = -MULT_NORM (iX[4] * T[4] + iX[5] * T[5]);

    oX1[0]   =   MULT_NORM (iX[6] * T[7] - iX[7] * T[6]);
    oX2[3]   = -MULT_NORM (iX[6] * T[6] + iX[7] * T[7]);

    oX2+=4;
    iX      +=     8;
    T       +=     8;
  }while(iX<oX1);
    /*post twiddling*/

  iX=out+n2+n4;
  oX1=out+n4;
  oX2=oX1;

  do{
    oX1-=4;
    iX-=4;

    oX2[0] = -(oX1[3] = iX[3]);
    oX2[1] = -(oX1[2] = iX[2]);
```

```
            oX2[2] = -(oX1[1] = iX[1]);
            oX2[3] = -(oX1[0] = iX[0]);

            oX2+=4;
        }while(oX2<iX);

        iX=out+n2+n4;
        oX1=out+n2+n4;
        oX2=out+n2;
        do{
            oX1-=4;
            oX1[0]= iX[3];
            oX1[1]= iX[2];
            oX1[2]= iX[1];
            oX1[3]= iX[0];
            iX+=4;
        }while(oX1>oX2);
    }
}
```

參考文獻

[1] XIPH, “Ogg Vorbis,” http://www.xiph.org/ogg/Vorbis/, 2002.

[2] E. Zwicker and H. Fastl, “Psychoacoustics Facts and Models,” Berlin, Germany: Springer-Verlag, 1990.

[3] P.-Y. Lin, “Audio Coding Using Multi-Subband Hybrid Trees,” National Chiao Tung University, June 1998.

[4] E. Terhardt, “Calculating virtual pitch,” *Hearing Res.* , vol. 1, pp.155–182, 1979.

[5] T. Painter, “Perceptual Coding of Digital Audio,” *Proceedings of IEEE*, Vol. 88, No. 4, April 2000.

[6] T. Painter and A. Spanias, “Perceptual Coding of Digital Audio”, *Proc. of IEEE*, vol. 88, pp. 451-515, April 2000.

[7] E. Ambikairajah and A.G. Davis, “Auditory Masking and MPEG-1Audio Compression,” *Electronics & Communication Engineering Journal*, Aug 1997.

[8] S. Pfeiffer, “The Ogg Encapsulation Format Version 0,” RFC3533, May 2003

[9] Rajsumam and Rochit, “System-on-a-Chip：Design and Test,” Norwood, MA：Artech House, 2000.

[10] Xilinx, http://www.xilinx.com/products/boards/multimedia/

[11] LEON2 Web Site, http://www.gaisler.com/, 2003.

[12] “The LEON2 User’s Manual,” http://www.gaisler.com/, 2003.

[13] ARM Corporation, “AMBA Specification 2.0,” http://www.arm.com, 1999.

[14] OAR Corporation, RTEMS Web Site, http://www.oarcorp.com, 2002.

[15] Man page of gprof command, 2002.

[16] J. P. Princen and A. B. Bradley, “Analysis/synthesis filter bank design based on time domain aliasing cancellation,” *IEEE Trans. Acoust. Speech Sig. Proc.*, pp. 1153-1161 (1986).

[17] B. Edler , K. Brandenburg and T. Sporer, “The use of multirate filter banks

for coding of high quality digital audio," *6th European Signal Processing Conference*, EUSIPCO, 1992.

[18] Atsushi Kosaka, Satoshi Yamaguchi, Hiroyuki Okuhata, Takao Onoye, Isao Shirakawa, "SoC design of Ogg Vorbis decoder using embedded processor," 2004.