國立交通大學

電信工程所

碩士論文

多核心管狀視訊解碼運算

Multicore Pipeline Video Decoding

研 究 生：江志偉

指導教授：張文鐘　教授

中 華 民 國 九 十 八 年 八 月

多核心管狀視訊解碼運算

Multicore Pipeline Video Decoding

研 究 生：江志偉　　Student：Chih-Wei Chiang

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學
電 信 工 程 所
碩 士 論 文

A Thesis
Submitted to Department of Communication Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering

August 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年八月

多核心管狀視訊解碼運算

學生：江志偉　　指導教授 ：張文鐘 博士

國立交通大學電信工程所碩士班

摘　　要

本論文的目標是整合多核心架構以及視訊解碼演算法，因此我們建構了一顆主核心與四顆副核心的多核心系統並在此系統架構下進行資料的平行處理，所以必須將運作於單核心的解碼程式改寫成多核心程式，而多核心程式的撰寫著重在兩大重心，找出可平行運算的解碼步驟、核心間的通訊機制與資料的傳輸。

首先從 MPEG-4 的完整解碼流程中找出可平行運算的解碼步驟，由於這些解碼步驟位於不同的副核心上，所以在副核心上是選擇管狀解碼運算而不是平行解碼運算。通訊機制方面，每個副核心有信號和信箱暫存區作為核心間的溝通管道，而暫存區均只有 32-bit 的大小，所以這些通訊機制主要是用來傳送資料的位址以及傳送工作運作時的狀態。為了在四顆副核心上達到管狀解碼運算，我們便利用這些暫存區於核心之間設計一個完整的溝通程序。資料的傳輸是透過 DMA 的方式取得資料，一旦知道資料位於記憶體的位址後，不管是主核心或副核心均可以到該位址進行資料存取，除此之外，資料傳輸會出現核心間資料覆蓋的問題，所以我們便利用多個資料暫存區來避免發生此錯誤。

實作上，主核心利用信箱詢問副核心的工作進度，由此決定是否分配工作給副核心協助解碼，由於副核心間是採用管狀解碼運算，所以主核心只需和第一顆副核心溝通而不需等待其他副核心執行結束。

Multicore Pipeline Video Decoding

Student：Chih-Wei Chiang　　Advisor：Wen-Thong Chang

Department of Communication Engineering

National Chiao Tung University

ABSTRACT

The goal of this thesis is to combine multi-core architecture with video decoding algorithm. We present a multi-core system with one main core and four secondary cores and compute data by parallel processing on this architecture. For this purpose, we must modify the single-core decoding program into multi-core decoding program. However, multi-core programming focus on finding the parallel decoding step, communicating and data transferring between cores.

First of all, we find parallel decoding steps from MPEG-4 decoding flow. These decoding steps are on the different secondary core, so we use pipeline decoding instead of parallel decoding. In communication protocol, every secondary core can use signal and mailbox register to communicate with each other. These registers only have 32-bit capability, so we use it to send address of data and status of task. For achieving pipeline decoding on four secondary cores, we design a communication procedure between each core by these registers. We transfer data through DMA command. Once we know the memory address, we can read or write data to memory no matter on the main core or secondary cores. Besides, transferring these data will introduce data overwriting between cores, so we use multiple data buffer to solve this problem.

In implementation, main core use mailbox to check task status of secondary core and then decide when to assign task to secondary core. Because of pipeline decoding between each secondary core, main core only need to communicate with first secondary core and don’t have to wait for other secondary core.

誌謝

在此最最最感謝的就是我的指導教授 張文鐘博士，感謝他提供我完善的資源，和一切所須的設備，讓我在兩年的碩士生涯中，不論在學業或是研究方面均無後顧之憂，並在論文的盲點上，點出了許多可能的解決方向。同時也謝謝黃仲陵教授、何文楨教授以及范國清教授在口試時的指導，老師們真是我研究過程中的一盞明燈，讓我順利完成此論文。

接著要感謝實驗室裡的同學和學弟們，盛如、小瘋、夸克、弘達、秉謙、建民、怡如、雅嵐、耀葦以及明穎，感謝他們陪我度過修課、報告、程式、寫論文的這兩年，沒有他們在我失落或瓶頸時的支持和鼓勵，我可能沒辦法支持下去，因此感謝你們給我這麼美好的回憶，期許我們一起成長前進。

感謝我的家人，父母親、大妹和小妹，他們是我的支柱，在研究所兩年中，提供我一切生活所須，有如一股無形的力量支持著我，讓我可以心無旁騖地研究，順利地完成論文。

最後感謝我的室友們，昆儒、國哲、士琪以及承曄，在兩年的共同扶持和幫助，也感謝所有認識的知心好友、社團的朋友以及其他實驗室的朋友，謝謝你們陪我走過這段時光。謝謝！

誌於 2008.08 風城 交大

志偉

目錄

表目錄

圖目錄

第一章 緒論

1.1 研究背景與動機

隨著科技的演進，處理器已由單核心漸漸走向多核心架構，原因在於多核心處理器較為省電，且具有指令平行化的功用。

多核心處理器之所以能夠節省電量，原因在於處理器脈衝頻率的限制，當頻率加快到一定速率之後，為了繼續增加成效而加快速率時，成效與所消耗的電量不成比例，此時，若改用兩個處理器，耗電量相當，可是運算效能卻可以多出更多。而指令平行化的優點在於，軟體同時執行多項作業時，各個核心處理器可以獨立執行一項作業，因此多核心將提供較大的計算量，為了讓軟體具有更多可平行運算的指令，就必須讓現有程式中，找出可平行運算的指令。

若要使多核心處理器達到高效能的運算，就必須將單核心用的原程式重新撰寫，故須對原程式有一定的了解，並妥善分配運算的工作量，解決資料同步等複雜性的問題。

另一方面，隨著影片畫質越來越好，檔案大小相對的也越來越大，為了降低檔案大小，並保持影片畫質與原本的一樣好，便有了影像壓縮技術的出現，視訊壓縮為一種高複雜度的編解碼演算法，藉由運算時間換取檔案的資料空間，所以需要的計算量相當大，因此，透過研究 Xvid 開發團隊所提供的 Open Source，深入探討 MPEG-4 解碼技術，目的為了解如何使用軟體實現演算法，以及視訊壓縮演算法是如何達到資料壓縮的目的，並從中找出可透過平行運算的解碼元件，將原本作用於單核心的 Xvid Codec 程式碼，修改成可在多核心處理器上運作，經由視訊壓縮演算法的平行化，降低影像解碼的時間。

目前較為普及的視訊壓縮技術為 MPEG-4 和 H.264，H.264 可說是 MPEG-4 下一代的影音壓縮標準，但是，現階段 MPEG-4 比 H.264 較被廣泛使用著，因此先從 MPEG-4 的視訊壓縮技術開始著手，多核心處理器方面，則是使用 Sony Cell Broadband Engine (PS3)，它包含一顆主核心以及八顆副核心，並在 Linux 環境下進行實驗。

1.2 論文架構

本論文的目標是整合 Cell Engine 多核心架構以及視訊壓縮演算法，將其編解碼指令平行化，以達到高效率的運算結果，因此，分成幾個章節來做詳細的闡述。第二章先介紹多核心處理器的軟體與硬體架構，以及 porting 部分程式至副核心時，必須用到的 API 之使用方法；第三章藉由探討 XVID CODEC 的解碼流程，找出可平行運算的指令，主要著重在 I、P、B-Frame 的解碼；第四章討論主題為結合二、三章之後，重新撰寫了哪些程式、如何在 SPE 端實現 pipeline 運作流程以及討論 porting 過程中所遭遇到的問題；第五章是實驗的部份，藉由數據的分析，討論 DMA 傳輸的問題以及改善的解碼時間；第六章則是本論文的結論以及未來發展。

第二章 Cell Broadband Engine

本論文以 Cell Broadband Engine (PS3) 為實驗的週邊設備，因此，此章對於多核心處理器做細部的分析與討論。2.1 節簡單介紹 PS3 的硬體以及系統架構；2.2 節說明 SPE 程式運作時，整個詳細的硬體與軟體流程；2.3 節將深入討論核心之間資料傳輸的各種問題；程式運作時，為了解決同步問題，核心間的通訊方式必須非常清楚，2.4 節則是針對通訊機制來做詳細的說明；2.5 節透過程式編譯的流程了解核心間的關係。

2.1 多核心架構與系統說明

PS3 是一個多核心處理器 (Cell Broadband Engine : Cell/B.E.)，Cell/B.E. 本身包含了一顆 64 位元的 PowerPC 處理器 (Power Processor Element : PPE)、八顆協同處理器 (Synergistic Processor Elements : SPEs)，而 PPE 與八顆 SPEs 是透過元件互連匯流排 (Element Interconnect Bus : EIB) 和主記憶體 (Main Memory) 以及 I/O 裝置連接在一起，其架構圖如圖 1 所示。

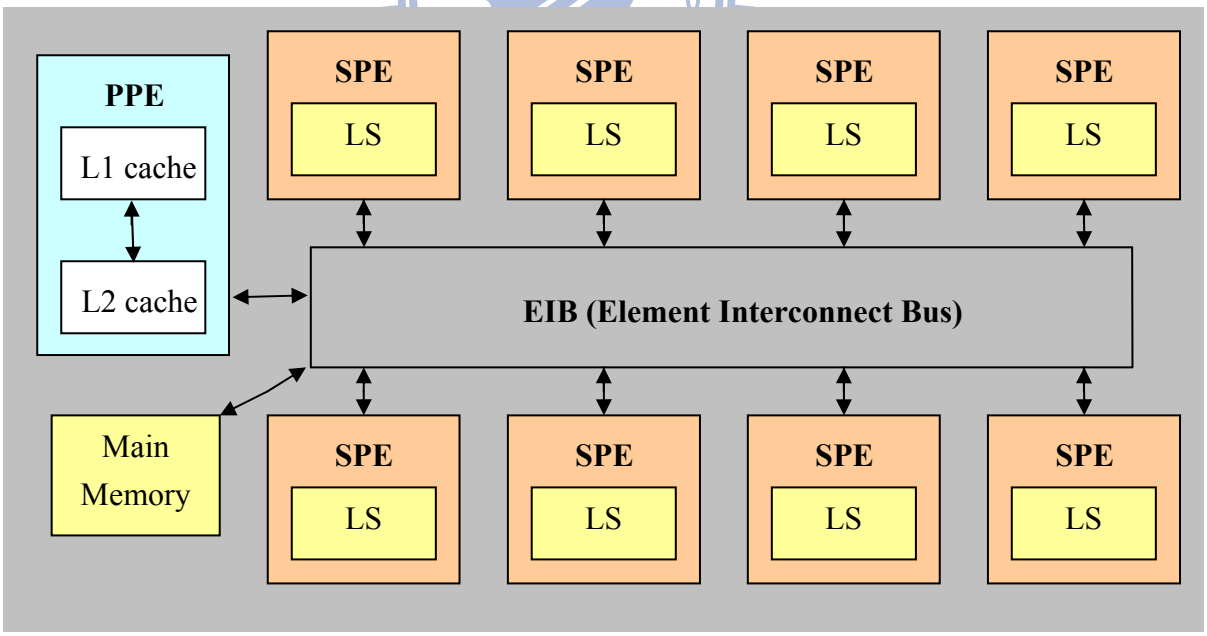


圖 1. 多核心架構圖

架構上，PPE 有一對 32KB 的一級快取記憶體 (L1 cache)，分別是指令快取記憶體 (instruction cache) 和數據快取記憶體 (data cache)，以及一個 512KB 的二級快取記

憶體 (L2 cache)，周圍的八顆 SPE 裡面都有一個 256KB 的區域記憶體 (Local Store : LS) 可供存取，至於 EIB 的功用，在於接收 PPE 或 SPE 發出直接記憶體存取 (Direct Memory Access : DMA) 的指令，以讀寫主記憶體上的資料。

系統上，PPE 主要是用來運行操作系統，而 SPE 則是負責執行 PPE 所分配的計算任務，PPE 會先將資料放置主記憶體上的某個暫存區，再由 SPE 從主記憶體搬運資料回區域記憶體，等 SPE 拿到資料並計算好之後，再將位於區域記憶體的結果由 EIB 寫回主記憶體。因此在實作上，我們便在 PS3 上安裝 Yellow Dog Linux 6.0 作業系統，並藉由 IBM 提供的 SDK (Software Development Kit) 函式庫來達到 SPE 和 PPE 之間系統上的實際操作。

2.2 SPE 程式的執行

當一個系統程式在執行時，大致上可分為主程式和子程式，系統程式又可分為在單一核心及多核心上運作。以 PS3 此多核心架構為例，當主程式和子程式同時運作於 PPE 時，就稱為單核心運作。若主程式在 PPE 上，而其他子程式則是在 SPE 上，此稱為多核心運作。但是要在 PPE 主程式中執行 SPE 子程式，需要調用一些必要的 API (Application Programming Interface) 函數，詳細的硬體及軟體架構流程如圖 2 所示。

一開始，需先將各個撰寫好的 SPE 原始碼透過 Compiler 編譯成可執行檔，此時的可執行檔位於硬碟中，接著使用以下的 API 函數。

1. 先使用 **spe_image_open()** 函數開啟位於硬碟中的 SPE 執行檔至主記憶體，此函數回傳指向該 SPE 程式的指標。

2. 接著，PPE 主程式需要執行 SPE 程式時，使用 **spe_context_create()** 函數建立一個 SPE 內容 (context)，此函數回傳該 SPE 內容的 handle，此 handle 可以當作此 SPE 內容的 ID，以提供使用者辨別、控制 SPEs。

3. 得到 SPE 內容之後，使用 **spe_program_load()** 函數，將程式從主記憶體載入至 SPE 內容中的區域記憶體，換句話說，256 KB 大小的區域記憶體，不只是儲存待處理的資料，還包含了程式碼的大小。

4. 此時 PPE 就透過已建立好的 SPE 程式內容，利用 **spe_context_run()** 函數去執行該程式，等 SPE 執行結束後，會回傳終止訊息給 PPE。

5. 當子程式 SPE 不再需要被用到時，使用 **spe_context_destroy()** 函數將 SPE 內容銷毀掉，此時原本的 SPE 內容就可再一次等待被建立及使用。

6. 最後，還要使用 **spe_image_close()** 函數將存在於主記憶體的 SPE 程式的指標關閉，以等待下一次 SPE 程式的開啟，程式碼的實現如圖 3 所示。

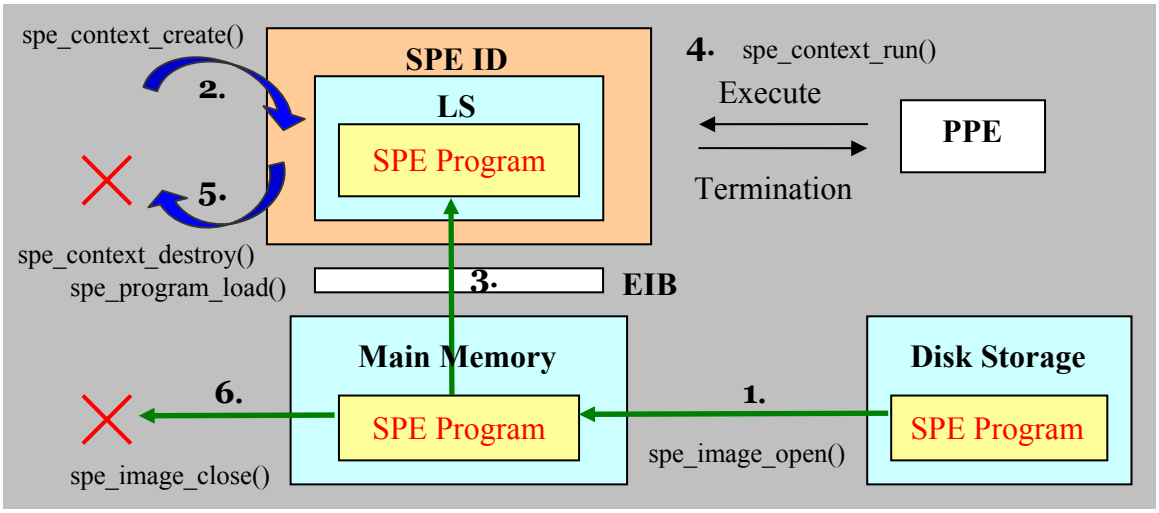


圖 2. SPE 程式的執行

ppe.c

```
spe_program_handle_t  *program;
spe_context_ptr_t speid;
uint32_t entry = SPE_DEFAULT_ENTRY;

program  = spe_image_open( "  spe "  );
speid = spe_context_create( 0 , NULL );
spe_program_load( speid , program );
spe_context_run( speid , &entry , 0 , NULL , NULL , NULL );
spe_context_destroy( speid );
spe_image_close( program );
```

傳入參數

圖 3. 執行 SPE 程式的片段程式碼

2.3 DMA 傳輸 (Direct Memory Access transfer)

DMA 是一種特殊的硬體結構，它允許系統直接讀寫記憶體，而不必透過 CPU，這種直接由特殊硬體完成資料傳輸的方法，節省了許多程式的執行時間。這邊所指的 DMA 傳輸，是以主記憶體與 SPE 上的區域記憶體以及 SPE 區域記憶體之間的資料傳輸為討論主軸，此外，後面還特別介紹雙重緩衝技術的應用以及 DMA 傳輸最常發生的錯誤 -- 匯流排錯誤 (Bus Error)。

主記憶體以及 SPE 區域記憶體是位於相同的記憶體空間上，如圖 4 所示，因此，只要取得位於記憶體空間的絕對位址，就可以知道資料位於 PPE 端或 SPE 端上的確

切位置，但是，PPE 端的絕對位址和 SPE 端區域記憶體上的絕對位址有些不同，每個 SPE 的區域記憶體可以想成是記憶體空間的某段記憶體，所以要得到區域記憶體上的絕對位址，必須知道該段記憶體的起始位址，以及相對於起始位址之位址偏移 (offset)，有了起始位址和位址偏移，便可以計算出區域記憶體上的絕對位址，有了絕對位址，不論是 PPE 與 SPE 之間或是 SPE 之間的資料傳輸，均可以用 DMA 傳輸到絕對位址作存取的動作。

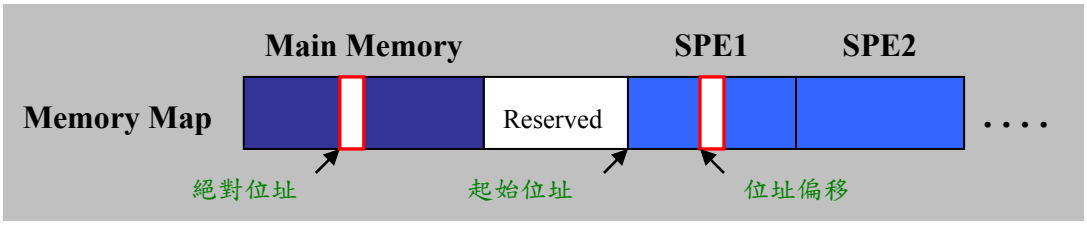


圖 4. 記憶體空間示意圖

2.3.1 主記憶體和 SPE 區域記憶體之間的資料傳輸

每個 SPE 都有 256KB 大小的區域記憶體，同時也是唯一 SPE 有權限進行讀寫的記憶體，因此，如果想和 PPE 共享位於主記憶體的資料時，必須先知道資料位於主記憶體的哪一個位置，並將資料從主記憶體搬運至自己的區域記憶體中，所以得到 PPE 端的資料位址為首要步驟。

得到 PPE 端的資料位址的方法有兩種：一、儲存在啟動 SPE 程式時的傳入參數，但只能在執行的一開始傳送一次，如圖 3 圈起來的部分，二、 SPE 程式執行期間，可透過 Mailbox 來傳送絕對位址的資訊，Mailbox 的詳細說明將會在 2.4.1 提到。

當取得絕對位址之後，便可利用 API 函數並透過 DMA 命令，將資料從主記憶體讀入至區域記憶體，或是從區域記憶體寫出至主記憶體中。詳細的硬體及軟體架構流程如圖 5 所示，流程以讀取主記憶體上的資料為例子。

1. 已知位址資料之後，需先在 SPE 程式內，宣告一個資料結構，其大小、型態要和主記憶體欲傳送過來的資料完全相同，接著，使用 **mfc_get()** 函數，告知函數要從主記憶體的哪個位址，傳多少資料到 SPE 的區域記憶體內，此函數會發出 DMA 指令，通過 channel 到 MFC (Memory Flow Controller) 裡的 DMA controller 下達指令。

2. 下完 DMA 指令之後，MFC 便會開始執行主記憶體和區域記憶體之間資料的調動，然後便開始接收 PPE 端送來的資料至 SPE 端已建立好的資料結構中。

3. 最後，使用 **mfc_write_tag_mask()** 和 **mfc_read_tag_status_all()** 函數等待資料傳送完成，圖 6 為程式的實現。

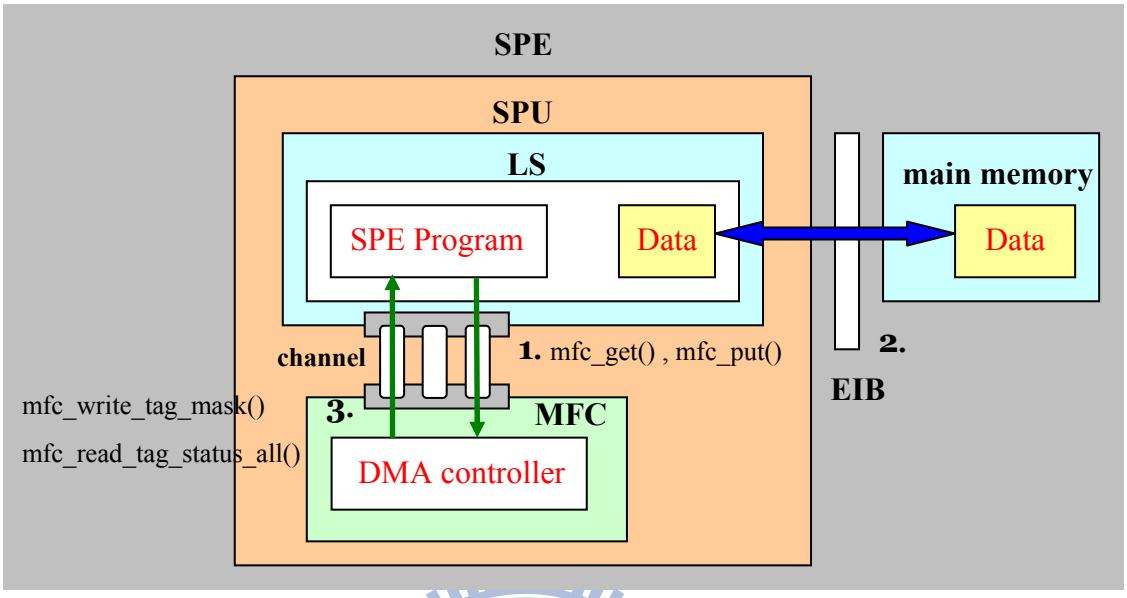


圖 5. DMA 傳輸

spe.c

```
uint32_t tag_id ;
ctx_t ctx __attribute__ ((aligned(128))) ; // 任意的一個資料結構變數

tag_id = mfc_tag_reserve();
                    位於 PPE 端上的資料之絕對位址
mfc_get( &ctx ,addr, sizeof(ctx) , tag_id , 0 , 0 );
mfc_write_tag_mask( 1<<tag_id );
mfc_read_tag_status_all();
```

圖 6. SPE 端 DMA 傳輸的片段程式碼

2.3.2 SPE 區域記憶體之間的資料傳輸

SPE 之間的資料傳輸，相當於 SPE 的區域記憶體之間的記憶存取，舉例來說，當 SPE1 需要取得位於 SPE2 的區域記憶體上的資料時，SPE1 必須先知道資料位於 SPE2 區域記憶體的哪一個位置，但是，要取得 SPE2 區域記憶體上的資料之絕對位址，需要取得兩個位址參數，第一個位址參數為 SPE2 的區域記憶體之起始位址，第二個位址參數是資料相對於起始位址的位址偏移 (offset)，將這兩個位址參數相加之後才是真正 SPE2 區域記憶體的資料位址，也就是說，當要跟某人拿東西時，你要知道要到

哪一層樓的哪一間房間，樓層就相當於起始位址，房間號碼就相當於偏移，而整棟樓就相當於整個記憶體空間。

SPE1 要取得 SPE2 的區域記憶體上的資料之絕對位址，所需的位址參數之取得流程如下：

1. 起始位址： SPE2 的區域記憶體之起始位址是由 PPE 利用系統函數獲得，接著透過 Mailbox 將起始位址傳送給 SPE1。

2. 資料的相對位址偏移：先由 SPE2 本身透過 Mailbox 方式將資料的相對位址偏移送回 PPE ，再從 PPE 端利用 Mailbox 傳送給 SPE1。

3. 最終位址：將 **1.** 得到的起始位址，加上 **2.** 的相對偏移， SPE1 就得到了 SPE2 資料位於區域記憶體上的絕對位址，利用計算出來的位址，就可使用 DMA 傳輸資料。

2.3.3 雙重緩衝技術 (Double Buffering)

SPE 程式運作中，當需要讀取大量的資料，並對部分資料進行運算的動作時，若使用雙重緩衝技術，將會大幅提升 SPE 程式運作的效能，因此，這邊將對雙重緩衝技術作詳細的說明，以及為何能達到提升效能的原因。以下為雙重緩衝的必要過程，其中輸出和輸入的緩衝區均有兩個，**input_1**、**input_2** 代表輸入的緩衝區，而 **output_1**、**output_2** 則代表輸出的緩衝區。

1. 首先，對 **input_1** 和 **input_2** 同時下達 DMA get 指令。

2. 等待 **input_1** 接收完成，對此資料進行運算，完成後存至 **output_1**，並下達 DMA put 指令將 **output_1** 寫出，同時，再次對 **input_1** 下達 DMA get 指令 (因為 **input_1**，已經完成運算的動作，所以可以再次接收新的一筆資料，等待下一次的運算)。

3. 此時切換到 **input_2**，等待接收完成 => 進行運算並存至 **output_2** => 下達 DMA put 指令將 **output_2** 寫出 => 再次對 **input_2** 下達 DMA get 指令。

4. 再一次切換回 **input_1**，這時，要注意的是，除了等待 **input_1** 接收完成，還須多等待 **output_1** 寫出結束，才可進行運算，否則會覆蓋到未寫完的資料，而造成結果錯誤，後面的步驟就跟 **2.** 一樣，進行運算並存至 **output_1** => 下達 DMA put 指令將 **output_1** 寫出 => 再次對 **input_1** 下達 DMA get 指令。

5. 將 **4.** 的所有步驟切換成緩衝區 2。

6. 重複切換 **4.** 和 **5.**。

7. 等待最後 **output_1** 和 **output_2** 寫出結束。

使用雙重緩衝技術之所以能夠提高效率，是因為 DMA 在進行資料傳輸時，不用透過 CPU，藉由此種特殊的硬體特性，可以在等待 DMA 傳輸完成的時間內，插入程序運算的步驟。最簡單的 DMA 流程如圖 7 所示，這種方法浪費大量時間等待 DMA 傳輸完成，因此，為了達到運算與傳輸同步進行，就需要分配兩個緩衝區，之所以輸出及輸入均須使用兩個緩衝區，原因在於等待 DMA 傳輸以及程式運算作用在不同的緩衝區，以達到資料間不會造成覆寫的情況。

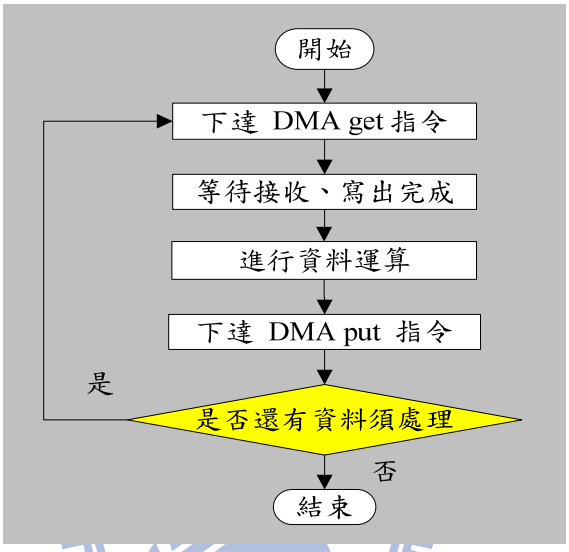


圖 7. DMA 流程圖

2.3.4 匯流排錯誤 (Bus Error)

匯流排錯誤是撰寫 SPE 程式的 DMA 傳輸中，最常見、也是最麻煩的問題，當 SPE 的區域記憶體要讀取主記憶體上的資料，取得資料的位址為首要步驟，但是，若讀取的主記憶體位址和存放的區域記憶體位址，沒有同時達到位址的對齊 (alignment)，此時就會發生匯流排錯誤，圖 8 為 16-byte 對齊與沒有對齊的例子。DMA 傳輸時，除了位址對齊的問題之外，還須考慮 DMA 傳輸大小，因此接下來將針對這兩大問題作詳細的討論與說明。

◆ 1. DMA 傳輸大小

每次的 DMA 傳輸資料大小必須為 1、2、4、8 或 16 的倍數，且每次的傳輸上限為 16 K，單位為 byte 數。

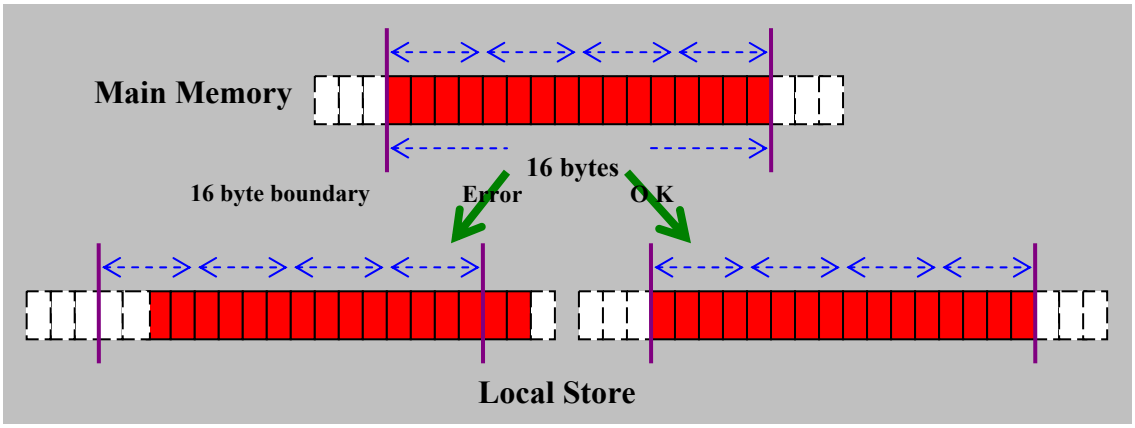


圖8. 匯流排錯誤 (16-byte)

◆ 2. 位址對齊

傳輸資料大小為 16 的倍數時，只須注意位址是否對齊在 16 位元組的邊界上 (byte boundary)，若小於 16 位元組時，傳送端和接收端要同時符合兩個限制條件，才可以避免造成匯流排錯誤，圖 9 是以傳輸大小為 4-bytes 的錯誤與正確範例。

(1) 限制一：傳輸大小與位址對齊大小相符，換句話說，當傳輸大小為 2-bytes 時，位址就要在 2-byte boundary 上，圖中的 Error_1 就是不符合此限制條件。

(2) 限制二：以 16-byte boundary 為基準點，傳送端與接收端位址之相對位置也要完全相同，如圖中的 Error_2，雖然有達到傳輸大小和位址對齊限制相符，但相對位置卻不相同。

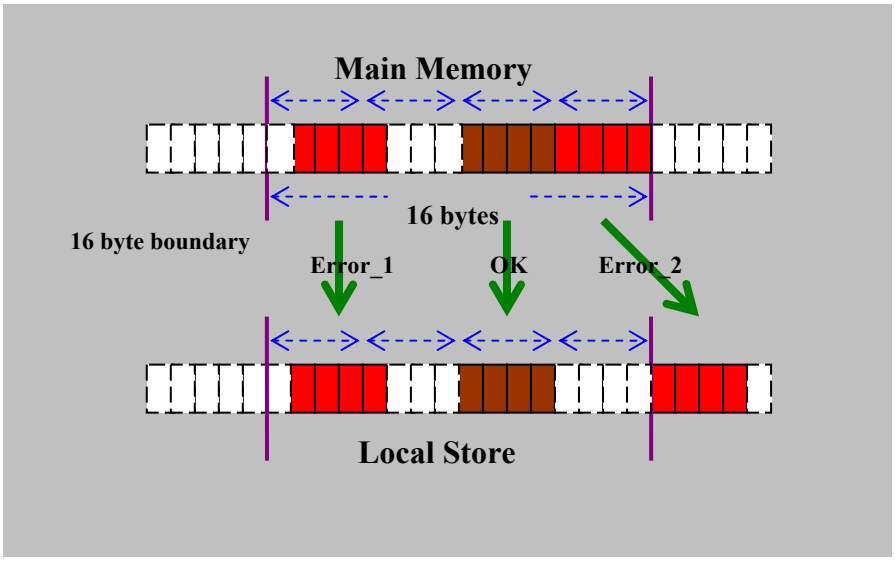


圖 9. 匯流排錯誤 (less than 16-byte)

2.4 Mailboxs 和 Signal Notification Registers 之通訊機制

在多核心系統的架構下，因為 SPE 的區域記憶體並不是共享的，因此須透過通訊機制來達到交換資料、同步等問題，除了上一節的 DMA 傳輸之外，還有 Mailboxs 和 Signal Notification Registers 這兩種通訊機制，這三種基本的通訊機制都是由 SPE 的 MFC 所控制，所以本節將對 Mailboxs 以及 Signal Notification Registers 來做詳細的說明。

2.4.1 Mailboxs

每個 SPE 均有三種不同功能的 Mailboxs，分別是輸出信箱 (Outbound Mailbox)、輸出中斷信箱 (Outbound Interrupt Mailbox) 以及輸入信箱 (Inbound Mailbox)，詳細的硬體及軟體架構流程如圖 10 所示，其中輸入信箱有四個暫存區，輸出及輸出中斷信箱各有一個暫存區，每個暫存區大小為 32-bit，且輸入信箱的四個暫存區之信息存取為先進先出 (First In First Out : FIFO)。

由於 Mailbox 只能傳送 32 bit 大小的資料，所以主要可分為兩大用途：一、當 PPE 端要通知 SPE 開始執行程式，或是 SPE 端運算結束後，要讓 PPE 得知運算結束的訊息時，Mailbox 就可以用來作為 PPE 和 SPE 之間的通訊，來傳送這些開始或結束的

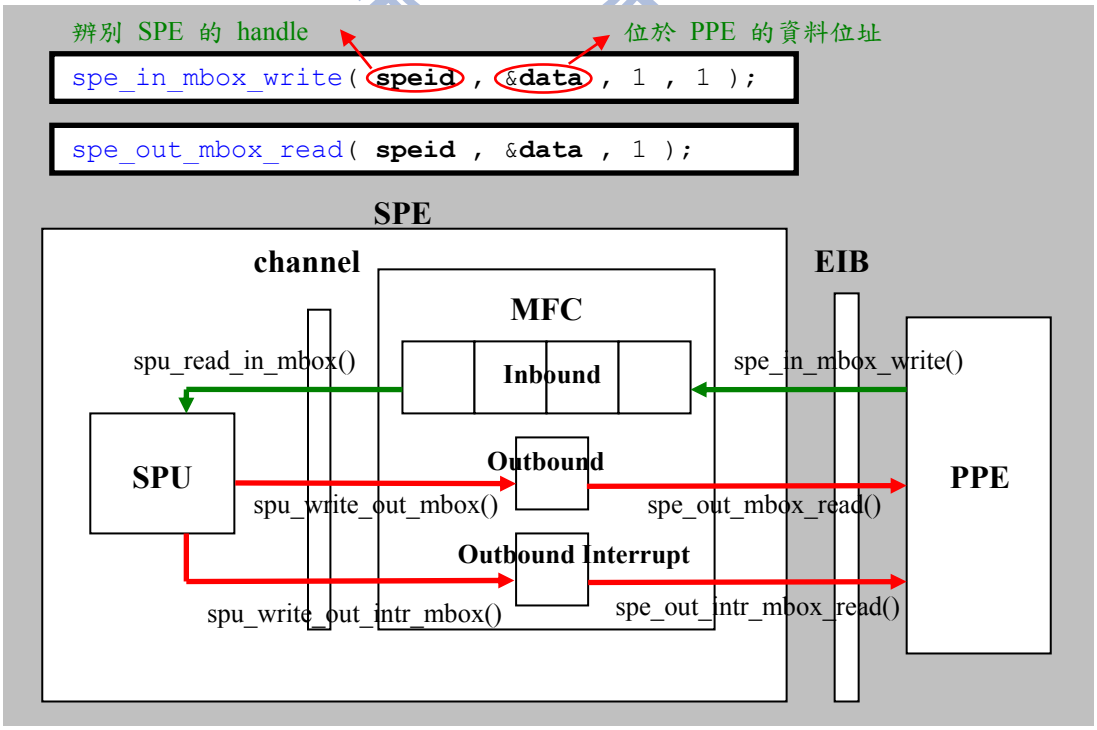


圖 10. Mailboxs

狀態報告。二、當 SPE 程式的運作過程中，需要到主記憶體或其他 SPE 端的區域記憶體抓取資料時，必須有那些資料的所在位址，這時，就可透過 Mailbox 來傳送位址給其他 SPE，或是接收其它 SPE 的所在位址。

輸出信箱和輸入信箱的使用上，最主要的不同在於，SPE 不論是讀取輸入信箱、寫入輸出信箱，均是阻塞式的 (BLOCKING)，而 PPE 則為非阻塞式的。以 SPE 讀取輸入信箱為例，當輸入信箱為空時，SPE 將會一直處於阻塞狀態，直到 PPE 或其他 SPE 寫入資料至輸入信箱。

而對於非阻塞式的 PPE 信箱操作來說，可能會發生兩種錯誤，一、當 PPE 在 SPE 程式讀取輸入信箱之前，PPE 已經連續寫入五次，此時第五次的資料將會遺失，二、當 PPE 讀取輸出信箱時，其實 SPE 還未寫入信息，此時 PPE 將會讀到錯誤的值。因此，為了解決以上兩種問題，必須使用特殊的 API 函數，來了解信箱內的狀態。

◆ 1. 錯誤一

當 PPE 要寫入信息到輸入信箱之前，都要先利用 **spe_in_mbox_status()** 函數，取得輸入信箱中，尚可寫入幾個信息，當狀態為零時，代表是滿的，此時就要等待狀態不為零，才可以再寫入信息。

◆ 2. 錯誤二

當 PPE 要從輸出信箱讀取出信息之前，先使用 **spe_out_mbox_status()** 函數，取得輸出信箱中，尚有幾個信息未讀出，當狀態為零時，代表輸出信箱是空的，因此要等狀態不為零，即表示有資料進來了，此時才可去讀取信息，圖 11 分別為解決錯誤一、二的片段程式碼。

錯誤一

```
while( spe_in_mbox_status(speid)==0 );  // 0 代表滿的，就停住
spe_in_mbox_write( speid , &data , 1 , 1 );
```

錯誤二

```
while( spe_out_mbox_status(speid)==0 ); // 0 代表空的，就停住
spe_out_mbox_read( speid , &data , 1 );
```

圖 11. Mailbox 錯誤的解決方法

之前有提到 SPE 程式執行期間，可以透過 Mailbox 取得主記憶體上的資料之位址，圖 12 為片段程式碼內容，目的是將 PPE 端陣列的起始位址傳送給 SPE 端，由於 PPE 端的位址大小是 64-bits，但 Mailbox 一次只能傳送 32-bits 的信息，所以須分兩次傳送，第一次先傳送位址之較高位元的 32-bits 資料，第二次再傳送較低位元的 32-bit資料。

ppe.c

```
addr = (uint64_t)&block[0];
spe_in_mbox_write( speid , (uint32_t *)&addr , 2 , 1 );
```

一次傳送兩個 mail 信息

spe.c

```
ea_h = spu_read_in_mbox();
ea_l = spu_read_in_mbox();
ea = mfc_hl2ea(ea_h,ea_l); // 將高位及低位重新組成 64-bits 的位址
```

圖 12. 傳送 PPE 端資料位址的片段程式

2.4.2 Signal Notification Registers

每個 SPE 均有兩個信號通知暫存器 (Signal Notification Register)，分別為 SNR1 和 SNR2，詳細的硬體及軟體架構流程如圖 13 所示，PPE 利用 **spe_signal_write()** 函數寫出信號至 SPE2，而 SPE1 則利用 **mfc_sndsig()** 函數寫出信號至 SPE2，但是

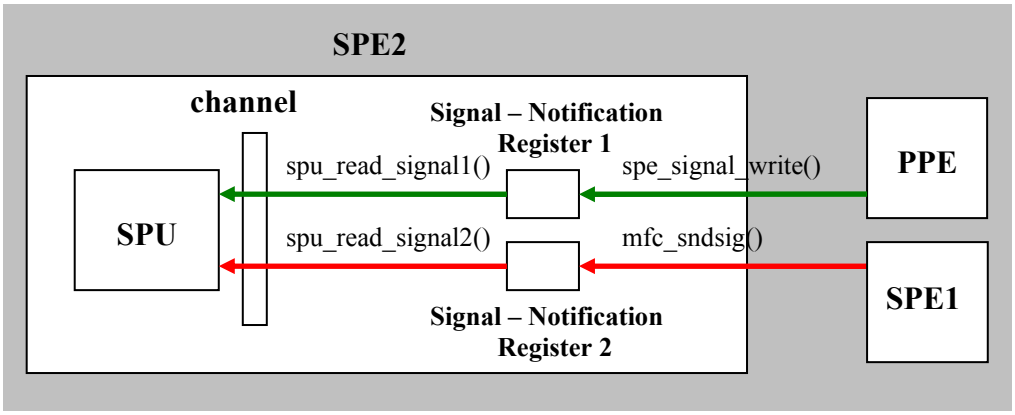


圖 13. Signal Notification Registers

，要讓 SPE1 傳送信號到 SPE2 的 SNR2 時，必須讓 SPE1 知道 SPE2 的 SNR2 之位址，因此，要先在 PPE 端取得 SPE2 的 SNR2 之絕對位址，而取得步驟如下：一、

先使用系統函數 **spe_ps_area_get()** 取得 SNR2 的起始位址，二、從起始位址位移 12-byte 之後即為 SNR2 的確切位址，再將此確切位址利用 Mailbox 傳送給 SPE1，接著，SPE1 就可以利用此位址傳送信號給 SPE2。圖 14 為計算 SPE2 的 SNR2 之絕對位址的程式碼以及記憶體的示意圖。

ppe.c

```
spe_sig_notify_2_area_t  *ea_area2;
uint64_t ea_sig2;

ea_area2  = (spe_sig_notify_2_area_t*)
            spe_ps_area_get(speid, SPE_SIG_NOTIFY_2_AREA);
ea_sig2 = (uint32_t)&(ea_area2->SPU_Sig_Notify_2);
```

SPE2 的 handle

SPE2

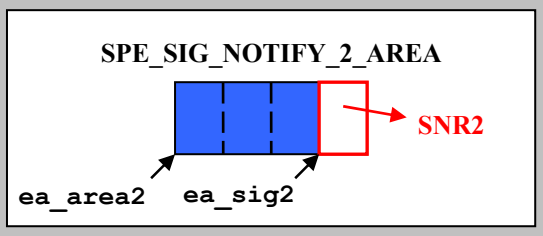


圖 14. PPE 端計算 SNR2 位址之程式碼以及示意圖

圖 15 為圖 13 的程式實現，由上述可知，信號通知暫存器是 12-byte 位址對齊，所以 SPE 之間的信號傳輸要如圖 14 中 spe1.c 方塊框起來的程式寫法，才能正確傳送信號，而不會造成匯流排錯誤。

ppe.c

```
spe_signal_write( speid , SPE_SIG_NOTIFY_REG_1 , data );
```

spe1.c

```
uint32_t signal[4] __attribute__ ((aligned(16)));

signal[3] = data;
mfc_sndsig( &signal[3] , ea_sig2 , tag_id , 0 , 0 );
mfc_write_tag_mask( 1<<tag_id );
```

SPE2 的 SNR2 之位址

圖 15. 對照圖 13.的片段程式碼

Signal 和 Mailbox 相同的是，暫存器大小都是 32-bit，不同的是，Signal 對於

Register 的寫入模式可設定成覆寫模式 (Overwrite mode) 或是邏輯 OR 模式 (logical-OR mode)，且 Register 每被讀取一次，內部所有位元資料將全被清為零。

◆ 1. 覆蓋模式

此模式的功能和 Mailbox 的輸入信箱大致上相同，只是由四個暫存區變成一個暫存區，而且對於 PPE 來說，信號的寫入也是屬於非阻塞式的，所以也會發生如 Mailboxs 提到的錯誤一，但是 Signal 和 Mailbox 產生的錯誤卻有些不同。當 SPE 程式讀取信號通知暫存器之前，PPE 已經寫入兩次以上時，不像 Mailbox 的輸入信箱，遺失的資料不是最後一次寫入的資料，而是前一次的資料會被後一次的寫入所覆蓋，換言之，就是只留下最後一次寫入的資料，因此，該模式比較像是一對一的信號通知。

◆ 2. OR 模式

此寫入模式算是多對一的一種，因為它可針對每個 bit 作個別寫入的動作，換句話說，如果 SPE1 和 SPE2 分別傳送 4 和 8 的 Signal 信號 (相當於第三和第四個 bit 為一的信號) 給 SPE3 時，其接收到的 Signal 信號結果將會變成 12 (=> 4 + 8)，但是，在 SPE3 讀取信號之前， SPE1 和 SPE2 必須已經完成寫入 Signal 信號的動作，否則就只會取得 SPE1 或 SPE2 的信號。

從圖 16 的示意圖可以看出這兩種模式的差異，而這差異在於位元處理的方式，因此，我們可以利用 OR 模式這種特別的通訊原理，來進行資料運算之間的同步作業，舉例來說：當 SPE1 和 SPE2 為兩個不同運算步驟的程式，每個程式均要處理六筆資料，且每筆資料要先等待 SPE1 運算完之後，才可由 SPE2 接著下一步驟的運算作業，因此，在 SPE1 端，每計算完一筆資料，就傳送代表著該筆資料的 bit 代號，換句話說，當 SPE1 的第一筆資料運算結束後，就向 SPE2 發出 000001 的信號，第二筆資料運算結束後，發出 000010 的信號，而最後一筆運算結束後，則是發出 100000 的信號，然而，在 SPE2 端，則不定時的去讀取信號通知暫存器，如果讀取到的信號為 000111，則代表目前 SPE1 已經運算完前三筆資料，此時 SPE2 就會先對這三筆資料作下一步驟的運算，運算結束之後，再次去查看信號通知暫存器，直到六筆資料都處理完成。

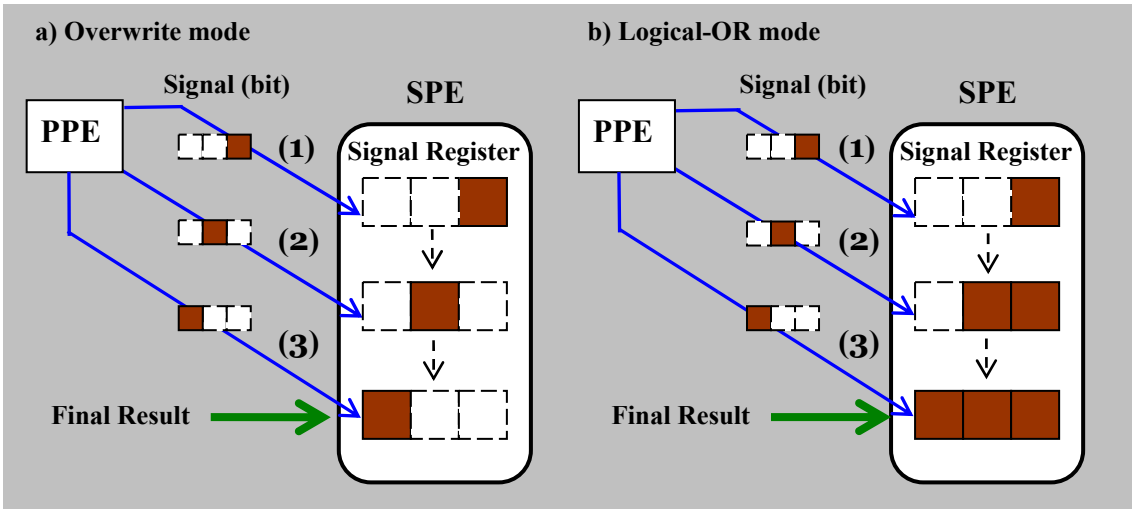


圖 16. 覆寫模式和邏輯 OR 模式

2.4.3 通訊機制的差異

Signal 和 Mailbox 的通訊機制最大差異在於 PPE 端與 SPE 端之間的通訊，Signal 屬於單向的溝通，而 Mailbox 是雙向的。對 Signal 來說，它只能從 SPE 端接收 PPE 端的信號，而不能傳送信號給 PPE 端，但是， Mailbox 還能讓 SPE 端寫入信息到輸出信箱，讓 PPE 端來讀取，因此，若考慮 SPE 和 PPE 之間的通訊問題，使用 Mailbox 較為恰當。

2.5 SPE 程式的嵌入 (Embedded SPE Program)

PPE 程式執行 SPE 程式時，是讀取編譯好的 SPE 程式之 ELF (Executable and Linkable Format) 可執行檔，其詳細流程已在之前已提過了，但是，一般大型程式的建立，最後的 ELF 可執行檔，均是由許多個不同的子程式編譯 (Compiler) 成各個 object 檔，最後再將這些物件檔結合 (Link) 起來形成 ELF 執行檔，因此，除了直接讀取 SPE 程式之 ELF 執行檔之外，SPE 程式還可以直接嵌入到 PPE 程式之 ELF 可執行檔中。

簡單來說，就是透過一些特別的 API 函數來將 SPE 執行檔轉換成 PPE 的 object 檔，圖 17 為編譯時的整個程序，藉由了解其詳細過程，可以對於 PPE 和 SPE 程式之間的關係有更加深入的了解。

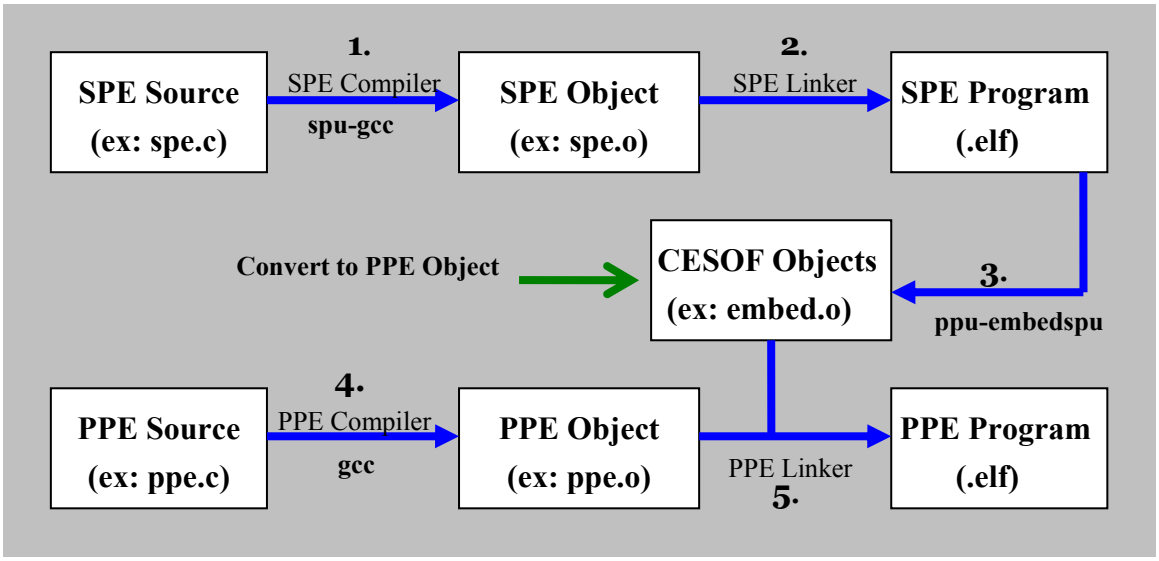


圖 17. 編譯程序

1. 先透過 **spu-gcc** 指令將 SPE 原始碼編譯成 object 檔。

2. 連結 object 檔形成 ELF 可執行檔。

3. 接著利用 **ppu-embedspu** 指令將 SPE 程式之 ELF 執行檔轉換成 CESOF (CBEA Embedded SPE Object Format) object 檔，此 object 檔為特殊的 PPE object 檔。

4. 再使用 **gcc** 指令將 PPE 原始碼編譯成 object 檔。

5. 最後將 **3.** 產生的特殊 PPE object 檔一併連結成 PPE 程式之 ELF 執行檔。

第三章 XVID CODEC

由於最後實驗的軸心是將 Xvid Codec 部分程式碼 porting 到 SPE 上運作，所以此章為細部討論影像解碼時，各個解碼元件的程式分析，並透過程式的分析，從中探討 MPEG-4 如何實現影像的壓縮技術。因此，3.1 節為簡單介紹整個解碼流程，以幫助對於程式的分析有初步的了解；對於 MPEG-4 編碼之後的資料來說，畫面之間會有相依性，因此，每解出一張畫面，便須將畫面以及一些必要的資訊儲存在資料結構中，3.2 節將針對這些資料結構做詳細的分析；3.3 至 3.5 節討論 I-Frame、P-Frame 以及 B-Frame 的程式解碼流程，並分析程式的架構。

3.1 解碼資料流程 (Decoder Data Flow)

藉由研究 Xvid Codec 的原始碼 (Open Source)，深入了解影像解碼的部分，因此，必須先對影片的結構有一定的了解。一段未壓縮的影片經過 Xvid Codec 編碼之後的影片結構如圖 18 所示，每個 Video Bitstream 是由許多 Sequence 構成的，每個 Sequence

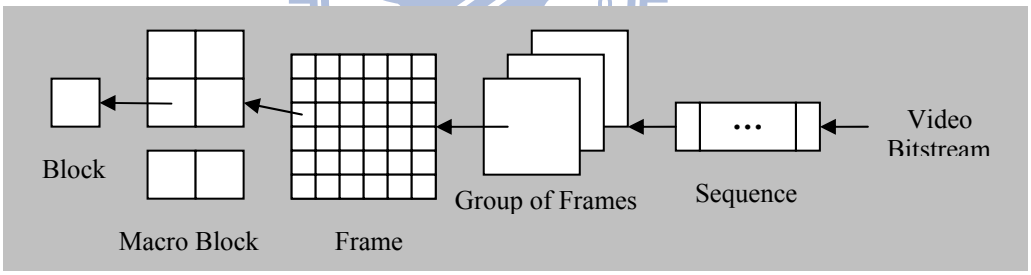


圖18. Video Stream 結構圖

又是由一堆 Frame 所組成，通常是一個 I-Frame 以及多個 P-Frame 和 B-Frame，每張 Frame 是由許多 MacroBlocks (MBs) 所組成，最後每個 MB 是 4y1u1v 六個 blocks 所組成，而標準的 I-Frame 或 P-Frame 的 MB 解碼資料流程如圖 19 所示：

1. 首先，對於壓縮影像的位元流進行變動長度解碼 (VLD)，內容包括計算所有 MB 的係數、移動向量以及相關的標頭。

2. 接著將解好的係數經過一些適當的重新排列，並進行反量化 (IQ)、反離散餘弦轉換 (IDCT) 等運算得到資料 D，若目前是進行 I-Frame 的解碼，D 即為結果 (稱為 intra MB)。

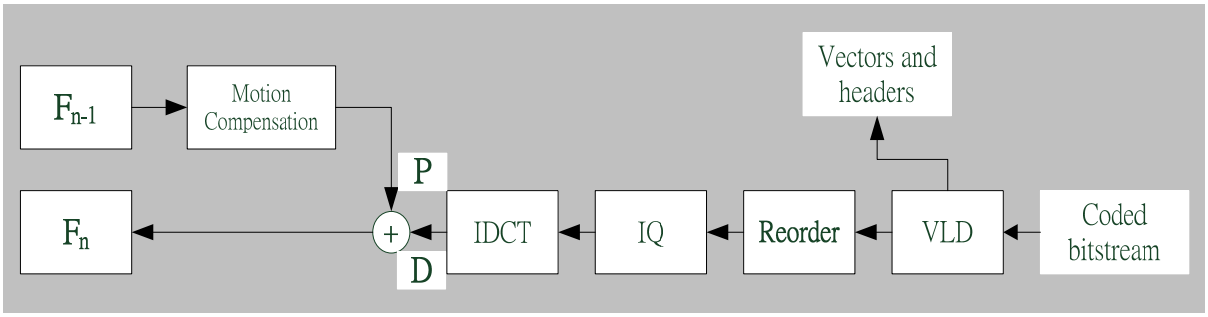


圖 19. Decoder Data Flow

3. 若目前是進行 P-Frame 的解碼，得到 D 之後，複製參考畫面中之移動向量所對應的預測區塊 P (也稱為 reference MB)，此步驟稱為移動補償 (MC)，然後將 P 加上 D (這邊的 D 是 residual MB，因為是儲存係數間的差值)，就得到資料 Fn，而 Fn 即為 P-Frame 解碼的最後結果 (稱為 inter MB)。

3.2 資料結構的分析

由於畫面間具有相依性，因此，解碼之後的部分影像資料必須保留，而儲存這些資料的結構為 DECODER，表 1 列出較為重要的結構成員，而表 2 為儲存整張畫面的 IMAGE 以及儲存 motion vector 的 VECTOR 結構內容，IMAGE 中的 *y、*u 以及 *v 分別為指向記憶體中存放 Y、U、V 畫面的起始位址。

表 1. DECODER 的結構成員 --- 變數宣告為 *dec

type	Name	description
uint32_t	mb_width	寬度有幾個 MBs
uin32_t	mb_height	高度有幾個 MBs
int	quarterpel	是否使用 1/4 畫素的旗標
int	interlacing	是否使用交錯式處理的旗標
int	time_bp	目前 B 畫面和先前參考畫面撥出時間差
int	time_pp	未來參考畫面和先前參考畫面撥出時間差
VECTOR	p_fmv	順向移動向量之預測值
VECTOR	p_bmv	逆向移動向量之預測值
IMAGE	cur	目前畫面
IMAGE	refn[2]	參考畫面
MACROBLOCK	*mbs	畫面中全部 MBs 的必要資訊
MACROBLOCK	*last_mbs	未來畫面中全部 MBs 的必要資訊

表 2. IMAGE 和 VECTOR 的結構成員

資料結構名稱	type	name	資料結構名稱	type	name
IMAGE	uint8_t	*y	VECTOR	int	x
	uint8_t	*u		int	y
	uint8_t	*v			

◆ **mb_width、mb_height**

為了解出一張完整的 Frame，我們總共需要解出 mb_height × mb_width 個 MB，mb_height 代表 y 方向有幾個 MB，而 mb_width 則代表 x 方向。

◆ **quarterpel**

在作移動補償時，一般的移動向量均落在整數畫素上，為了使移動補償達到更好的壓縮率，有時移動向量會落在 1/2 畫素或 1/4 畫素上。整數畫素之間的中間點即稱為 1/2 畫素，所以 1/2 畫素之間的中間點即為 1/4 畫素。

◆ **interlacing**

當畫面使用交錯式處理時，代表著該畫面的奇數橫列和偶數橫列，分別儲存著不同時間點的影像資料，使用交錯式處理與否，將影響解碼時，該如何將正確影像資料還原回來。

◆ **time_bp、time_pp**

解 B-Frame 時，如果為直接模式 (direct mode)，順向、逆向移動向量均是從參考向量運算出來，而參考的移動向量為未來畫面中位於相同的 MB 所在位置上之向量。欲運算出向量，除了需要上述的參考向量，還需要 time_bp 和 time_pp 參數，而 time_bp 為目前 B 畫面的撥出時間減去先前參考畫面的撥出時間，相對的，time_pp 為未來參考畫面的撥出時間減去先前參考畫面的撥出時間，至於更完整的運算過程，會在 3.5 節中做說明。

◆ **p_fmv、p_bmv**

移動向量的取得為向量差值加上向量預測值，因此，解 B-Frame 時，若不為直接模式，p_fmv、p_bmv 為儲存順向、逆向移動向量之預測值的參數。

◆ **cur、refn[0]、refn[1]**

cur 為存放目前解碼後的整張畫面之結構變數，而 refn[0] 則是存放未來參考畫面之結構變數、refn[1] 為先前參考畫面之結構變數。

◆ ***mbs**

解碼過程中，不只是畫面之間資料有相依性，同一個畫面中相鄰的 MBs 之間也具有相依性，因此，每個 MB 運算完之後，均要將相關的資料留下，讓下一個 MB 利用這些資料進行運算。*mbs 為指向整張畫面的所有 MBs 相關資料的起始位址指標。MB 相關的資料結構為 MACROBLOCK，較為重要的結構成員如表 3 所示，且儲存所有 MBs 相關資料的記憶體大小為 sizeof (MACROBLOCK) × mb_width × mb_height。

◆ ***last_mbs**

進行 B-Frame 的解碼，且為直接模式時，運算向量需要儲存未來畫面中的參考向量，而參考向量就儲存在 last_mbs->mvs[4] 中，其中，*last_mbs 為指向未來參考畫面之 MBs 相關資料的起始位址指標。

表 3. MACROBLOCK 的結構成員

type	name	description
short int	pred_values[6][15]	係數的預測值
int	acpred_directions[6]	AC 係數的預測方向
int	mode	此 MB 的模式 (0～4)
int	quant	量化參數
int	cbp	4y1u1v 六個 block 的位元資訊
VECTOR	mvs[4]	Y 的四個 block 之移動向量
VECTOR	b_mvs[4]	Y 的四個 block 之逆向移動向量

◆ **pred_values[6][15]、acpred_direction[6]**

因為相鄰的 intra-coded 8× 8 blocks 通常都是有相關性的，所以，與其儲存整個係數值，倒不如儲存相鄰 blocks 係數之間的差值，以達到資料壓縮的目的，而這些差值稱為係數的預測值 (coefficient prediction)。因此，要計算一塊未解碼的 block 時，可從相鄰的 (左、上及左上) block 來計算出 DC 係數及 AC 係數的第一列或第一欄的預測值 (因此預測暫存區總共為 15 個)，加上每個 MB 由六個 blocks 所組成，所以每個 MB 的所有係數之預測值，便儲存在 dec->mbs[**mb_width** × **mb_height**].pred_values[6][15] 的參數中。

當決定好 AC 係數是由相鄰 block 的第一列或第一欄取得後，便存下預測方向在 acpred_direction[6] 參數中，在 3.3.3 節有針對 Intra prediction 作更詳細的說明。

◆ **mode**

對於 MB 的情況，一般可以分為五種，0 為 inter、1 為 inter + Q (Q 表示量化參數須作變動性的調整)、2 為 inter + 4mv、3 為 intra、4 為 intra + Q。

◆ **quant**

還原係數時，須乘上該量化參數以取得原始的係數值，量化參數的範圍為 1～31。

◆ **cbp**

此變數的位元資訊如圖 20 所示 (最右邊為最低位元)，且 cbp 又可以分成 cbpc 和 cbpy，cbpc 為 U、V 的部分，而 cbpy 則是 Y。進行 intra block 解碼時，此資訊決定是否運作 Run-level Coded 係數的解碼，而進行 inter block 解碼時，此資訊決定是否運作 residual block 的解碼。

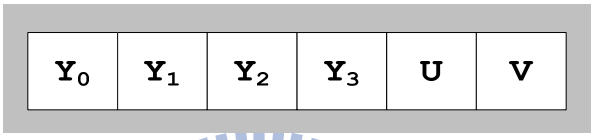


圖 20. cbp 的位元資訊

◆ **mvs[4]、b_mvs[4]**

當 mode 為 inter + 4mv 時，mvs[0] ~ mvs[3] 代表著 Y 的四個 motion vector，若 mode 不為 inter + 4mv 時，Y 只需儲存一個 motion vector，所以只使用 mvs[0]，而 b_mvs 為 B-Frame 解碼時，儲存逆向移動向量之參數。

3.3 Decode for I-Frame (Intra Frame)

I-Frame 的解碼，位於 Xvid Codec 原始碼的 **decoder_iframe()**，圖 21 為程式的流

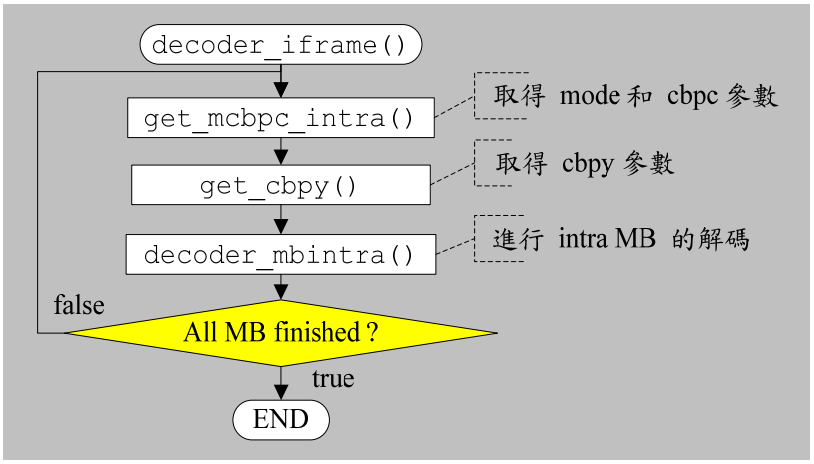


圖 21. decoder_iframe() 流程圖

程圖，圖 22 為實現 I-Frame 解碼的部分程式碼，接下來將詳細說明子程式的運作流程。

```
void decoder_iframe(DECODER *dec , Bitstream *bs ,int quant)
{
    for( y = 0 ; y < mb_height; y++) {
        for (x = 0; x < mb_width; x++) {
            mcbpc = get_mcbpc_intra(bs);
            mb->mode = mcbpc & 7;    // 只取 mcbpc 的最右邊 3 bits
            cbpc = (mcbpc >> 4);
            // 是否採用 prediction 來取得 ac 係數的旗標
            acpred_flag = BitstreamGetBit(bs);
            cbpy = get_cbpy(bs, 1); // 1 表 intra , 0 表 inter
            cbp = (cbpy << 2) | cbpc;
            if ( mb->mode == MODE_INTRA_Q ) {
                quant += dquant_table[ BitstreamGetBits(bs, 2) ];
                if (quant > 31)
                    quant = 31;
                else if (quant < 1)
                    quant = 1;
            }
            mb->quant = quant;          // 取得量化參數(quantizer parameter)
            if (dec->interlacing)       // 是否採用交錯式處理
                mb->field_dct = BitstreamGetBit(bs);
            decoder_mbintra(dec, mb, acpred_flag, cbp, bs, quant);
        }
    }
}
```

圖 22. decoder_iframe() 片段程式碼

3.3.1 get_mcbpc_intra()

此函數是從位元流中取得 mcbpc (mode and coded block pattern for chrominance) 參數值，mcbpc 代表二個資訊，第一個資訊為 MB 的 mode，而第二個資訊為 cbpc。由於目前是 intra mode，所以 mode 只會有 3 和 4 兩種可能，配合 cbpc 有四種情況 (00、01、10、11) ，將 mode 和 cbpc 組合之後，共有八種結果，其中 mode 資訊儲存在 mcbpc 值的右邊四位元，cbpc 資訊則是儲存在右邊數過來的第五、六位元，如表 4 所示。

一般常用的編碼技術為變動長度編碼 (Variable Length Coding : VLC)，而解碼時，我們可以利用 Huffman tree 或是查表的方式來進行解碼，這邊選用查表的方式解出 Huffman Code 所對應到的資訊，接下來，將以 intra mode 為例，分析如何透過 table 達到 Huffman Code 的編解碼。

表4. VLC table for mcbpc

Huffman Code	cbpc (56)	mbtype	mcbpc (cbpc-mode)	Integer for mcbpc
1	00	3	00-0011	3
001	01	3	01-0011	19
010	10	3	10-0011	35
011	11	3	11-0011	51
0001	00	4	00-0100	4
000001	01	4	01-0100	20
000010	10	4	10-0100	36
000011	11	4	11-0100	52

以編碼的角度來看，當 mcbpc 為 01-0011 (cbpc-mode) 時，對照表 4 可以得知 Huffman Code = 001，因為這八種 Huffman Code 長度最長為 6，欲使用查表就必須將長度均固定為 6，於是將 001 後面添加 3-bits 的資訊，而 Huffman Code + 添加的 bits 即為陣列位置，如 001-xxx，而且 xxx 不論資訊為何，均會對應到相同的 mcbpc，如圖 23 所示，圖中的 {19 , 3}，分別表示 mcbpc 和 Huffman Code Length，下一段將以解碼的角度來解釋如何利用 {mcbpc , Huffman Code Length} 達到解碼。

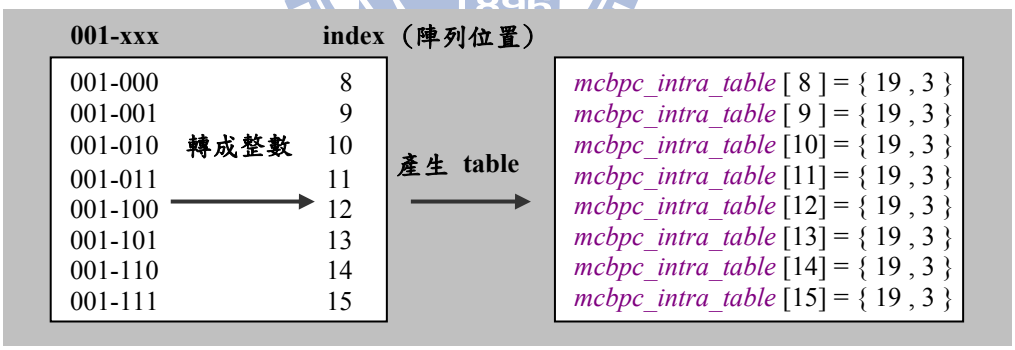


圖 23. 001-xxx 對照 mcbpc之範例

以解碼的角度來看，由圖 24 的程式碼得知，先從位元流中取出 6-bits 換算成陣列位置 index，以 index = 59 (位元表示為 111011) 為例，對照圖 25 的表後，得到 *mcbpc_intra_table* [59] = { 3 , 1 }，又 111011 屬於 1- xxxxx 的情況，所以 Huffman Code = 1、Huffman Code length = 1，因此，元素 {3, 1} 中的第一個資料 3 是待回傳的 mcbpc 值，第二個資料 1 為 Huffman Code length，是用來從位元流中取出 Huffman Code 資訊。

```
get_mcbpc_intra(Bitstream * bs){
    uint32_t index;
    index = BitstreamShowBits(bs, 6);
    BitstreamSkip(bs,mcbpc_intra_table[index].len); // 取出 Huffman Code
    return mcbpc_intra_table[index].code;           // 回傳 mcbpc 值
}
```

圖 24. get_mcbpc_intra 程式碼

```
VLC const mcbpc_intra_table [64] = {
{-1, 0},{20, 6},{36, 6},{52, 6},{4, 4}, {4, 4}, {4, 4}, {4, 4},
{19, 3},{19, 3},{19, 3},{19, 3},{19, 3},{19, 3},{19, 3},{19, 3},
{35, 3},{35, 3},{35, 3},{35, 3},{35, 3},{35, 3},{35, 3},{35, 3},
{51, 3},{51, 3},{51, 3},{51, 3},{51, 3},{51, 3},{51, 3},{51, 3},
{3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1},{3, 1},
{3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1},{3, 1},
{3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1}, {3, 1},{3, 1},
{3, 1}, {3, 1}, {3, 1},{3, 1}, {3, 1}, {3, 1}, {3, 1},{3, 1}
} ; index 代表陣列位置
```

index = 59

圖 25. mcbpc_intra_table

得到回傳值 mcbpc = 3 後，換算成位元為 00-0011 (cbpc - mode)，即得到 mode = 3 且 cbpc = 00 的資訊，然後就結束了 Huffman Code 的解碼。

3.3.2 get_cbpy()

由圖 26 的程式碼得知，先從位元流中讀出 6-bits 換算成陣列位置 index，透過 index 查表取出 cbpy 參數，cbpy 參數是儲存亮度 $y_0 \sim y_3$ 之 block 是否為 1 的資訊，因此，

```
get_cbpy(Bitstream * bs, int intra){
    int cbpy;
    uint32_t index = BitstreamShowBits(bs, 6);
    BitstreamSkip(bs, cbpy_table[index].len);    // 取出 Huffman Code
    cbpy = cbpy_table[index].code;
    if (!intra)   cbpy = 15 - cbpy;              // inter 和 intra 剛好相反
    return cbpy;
}
```

圖 26. get_cbpy 程式碼

cbpy = 0 ~ 15，cbpy = 1111 代表全部為 1，詳細的編解碼技術和取得 mode 和 cbpc 的函數 (**get_mcbpc_intra**) 相同，且從表 5 得知，intra 和 inter 的 cbpy 剛好相反，例如：當 Huffman Code = 11，對 intra 來說，此時 cbpy = 1111，而 inter 時，cbpy = 0000。

表5. VLC table for cbpy

Huffman Code	cbpy(intra-MB) (1234)	cbpy(inter-MB) (1234)
0011	0000	1111
00101	0001	1110
00100	0010	1101
1001	0011	1100
00011	0100	1011
0111	0101	1010
000010	0110	1001
1011	0111	1000
00010	1000	0111
000011	1001	0110
0101	1010	0101
1010	1011	0100
0100	1100	0011
1000	1101	0010
0110	1110	0001
11	1111	0000

取得 cbpy 參數後，將 cbpy 和 cbpc 作 OR 運算合併成 cbp，接著會查看 mode 是否為 intra + Q，如果是，將對量化參數作些微的調整，如圖 22 框起來的地方，而這些調整大小為 {-2 , -1 , 1 , 2}，且量化參數的最小值為 1、最大值為 31。

3.3.3 decoder_mbintra()

此程式為解出 intra block 的核心函數，圖 27 為 **decoder_mbintra()** 的部分程式碼，而圖 28 為程式的流程圖，從流程圖可以看出，先前得到的 cbp 參數，是用來決定是否需要進行 Run-level 解碼的旗標。

```
static void decoder_mbintra(dec, pMB, acpred_flag, cbp, bs,quant)
{
    for (i = 0; i < 6; i++) {
        uint32_t iDcScaler = get_dc_scaler(quant, i<4);
        int16_t predictors[8];  // 係數預測值的暫存區
        predict_acdc( .., predictors ,.. );
        if (!acpred_flag) pMB->acpred_directions[i] = 0;
        if (cbp & (1 << (5 - i)))
            get_intra_block(bs,&data[i*64],direction, .. );
        add_acdc( .., &data[i*64], iDcScaler, predictors, .. );
        dequant_h263_intra(.,&data[i*64],quant,iDcScaler,.);
        idct((short * const)&data[i*64]);
    }
    transfer_16to8copy(pY_Cur , &data[0*64] , stride);
    transfer_16to8copy(pY_Cur + 8 , &data[1*64] , stride);
    transfer_16to8copy(pY_Cur + next_block , &data[2*64] , stride);
    transfer_16to8copy(pY_Cur + next_block + 8 , &data[3*64] , stride);
    transfer_16to8copy(pU_Cur , &data[4*64] , stride2);
    transfer_16to8copy(pV_Cur , &data[5*64] , stride2);
}
```

圖 27. decoder_mbintra() 片段程式碼

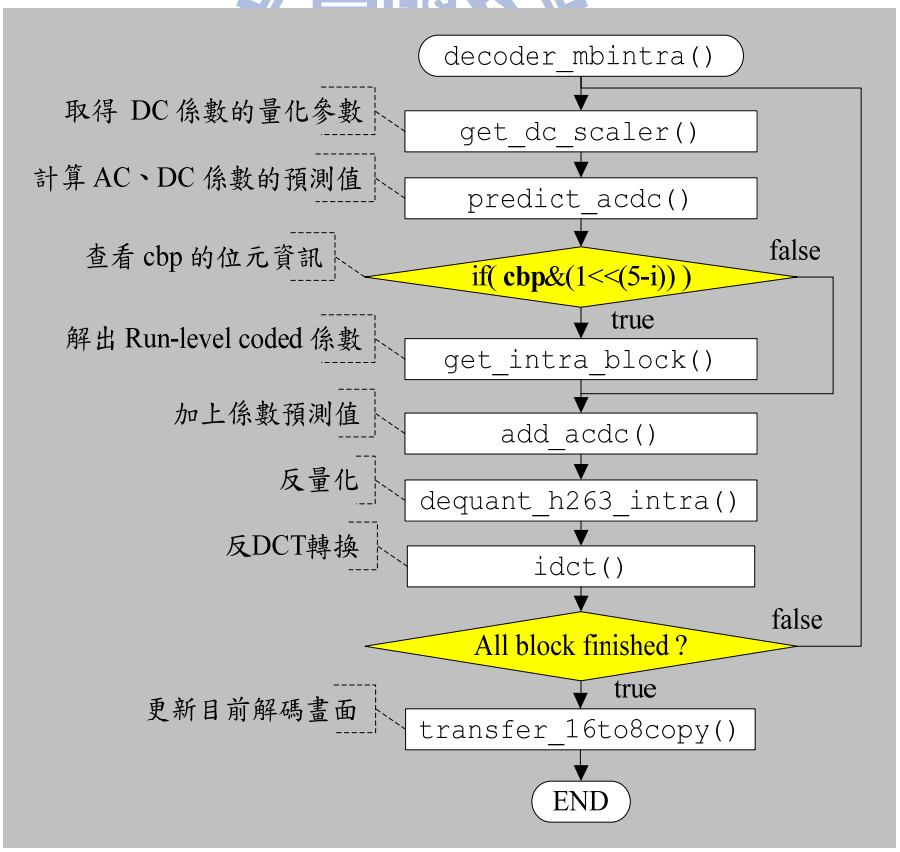


圖 28. decoder_mbintra() 流程圖

◆ 1.get_dc_scaler()

傳入值 quant 為量化參數 (Quantizer scale parameter：QP)，而 i＝0～3 時，為 Luminance，i＝4～5 時，為 Chrominance，接著由表 6 取得所對應的 DC 係數之量化參數 iDcScaler。

表 6. QP ⇔ dc_scaler 轉換表

Block type	QP ≤ 4	5 ≤ QP ≤ 8	9 ≤ QP ≤ 24	25 ≤ QP
Luma	8	2× QP	QP + 8	(2× QP)-16
Chroma	8	(QP + 13)/2	(QP + 13)/2	QP-6

◆ 2. predict_acdc()

此函數用來計算 DC 係數以及部分 AC 係數之預測值。首先，先針對左、上及左上區塊的係數預測值做初始化，其默認值 (default values) 為 DC 係數 1024，其餘的 14 個 AC 係數設為 0，然後開始判斷左、上及左上的 block 此時是否位於邊界上，若位於邊界，其相鄰 block 的係數預測值則使用默認值，若不在邊界上，則使用相鄰 block 的係數預測值，由於亮度 (Luma) 是由四個 block 所組成，所以選擇的相鄰 block 之係數預測值，特別要經過一些指標上的偏移，以圖 29 為例，當想要取得 Y_2 的相鄰 block 之係數預測值時，左邊 block 的係數預測值位於 left MB 的 Y_3 位置，上面 block 的係數預測值則為

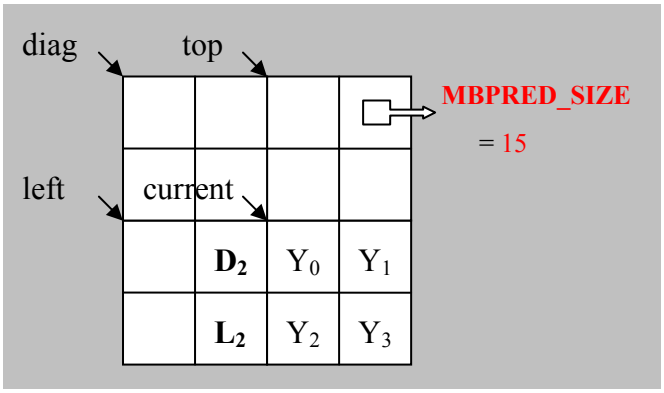


圖 29. 相鄰 block 的示意圖

current MB 的 Y_0 位置，而程式碼的實現如圖 30 的 case2 所示，而 case 5 的程式碼，為取得 V 的相鄰 block 之係數預測值。若想取得 current V 的相鄰 block 之係數預測值時，由於 V 只有一個 block，且每一個 MB 的係數預測值是依照 Y_0、Y_1、Y_2、Y_3、U、V 的

```
case 2:          若左邊是邊界，此時 left 是 NULL
    if(left){
        pLeft = left + 3 * MBPRED_SIZE; // 如圖 28 的 L2
        pDiag = left + MBPRED_SIZE;     // 如圖 28 的 D2
    }
    pTop = current;
    top_quant = current_quant;
    break;
. . . . . .
case 5:
    if(left)    pLeft = left + 5 * MBPRED_SIZE; // 跳過五個 block
    if(top)     pTop  = top  + 5 * MBPRED_SIZE;
    if(diag)    pDiag = diag + 5 * MBPRED_SIZE;
    break;
```

圖 30. predict_acdc() 片段程式碼

順序所排列，因此從程式碼可以看出，均是固定跳過五個 block 的指標偏移來取得 V 的相鄰 block 之係數預測值。

取得相鄰的左、上及左上 block 的係數預測值之後，如圖 31 左邊的 pseudo code，經由選擇較小的 DC 差，來決定預測方向，如果左邊和左上 block 的 DC 係數預測值相

```
if |DCA - DCB| < |DCB - DCC|
      predict from block C
else
      predict from block A
```

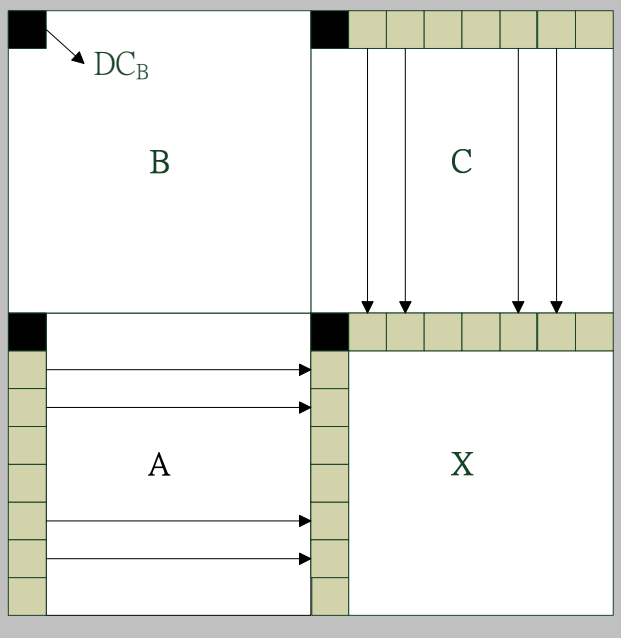


圖 31. DC、AC 係數的預測

減較小，預測方向為垂直方向 (dec->mbs[**x** + **y** * **mb_width**].acpred_directions[i] = 1，其中 i 代表 4y1u1v 的 block 旗標) ，此時 block X 將選擇 block C 為預測目標，表示

block X 的 DC 以及 AC 的第一橫列之係數預測值將由 block C 取得，反之，當預測方向為水平方向時 (dec->mbs[**x** + **y** * **mb_width**].acpred_direction[i] = 2)，block X 的 DC 以及 AC 的第一直欄之係數預測值則是從 block A 取得，因此，此函數的目的，是取得八個係數預測值 (一個 DC、七個 AC)。

◆ 3. get_intra_block()

經由查看 cbp 參數之後，決定是否要做 Run-level Decode (RLD)，解碼的一開始，會先決定存放 block 係數的掃描順序，選擇掃描方式的依據在於何種方式可以達到較佳的壓縮效率，而掃描方式這邊分為三種，Z 字形 (Zig-Zag)、交錯式水平 (Alternate-Horizontal) 及交錯式垂直 (Alternate-Vertical) 掃描，一般都是選用 Z 字形掃描 (dec->mbs[**x** + **y** * **mb_width**].acpred_directions[i] = 0)，因為 DCT 之後的係數大小會按照頻率的高低來作排列，而且，越是高頻訊號，值為零的機率越高，因此，並可以利用此特性配合 Z 字形掃描使得出現零值的訊號連續出現，接著使用變動長度編碼 (Variable-Length Coding : VLC) 來達到最高的壓縮率，但是，如果在計算預測值的函數中 (**predict_acdc**)，預測方向為水平時 (dec->mbs[**x** + **y** * **mb_width**].acpred_directions[i] = 1)，此時的掃描方式就為交錯式垂直掃描，圖 32 為不同掃描方式的示意圖。

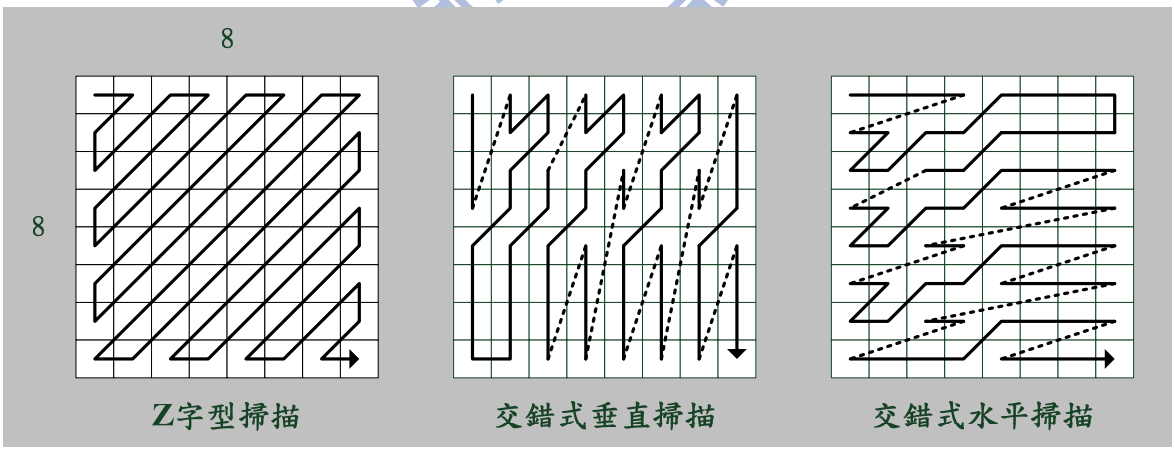


圖 32. 係數掃描方式

決定好掃描方式之後，便開始從位元流中取出 RLD 所需要的參數，係數的解碼利用查表來實現，其相關的表為 *DCT3D* [2] [4096]，且表所儲存的資料結構為 REVERSE_EVENT，其結構成員如表 7，從結構成員可以看出，有四個最主要的參數，last 代表是否為最後一個不為零的係數，0 代表不是、1 代表是，run 的值代表該係數的

前面有幾個零，level 就是此係數真正的值，而 len 值代表該 VLC 的 Huffman Code 碼長，len 的用途在於更新位元流的位置，藉由這四個參數就可以解出整個 block 的所有係數值。*DCT3D* 陣列前面的 2 代表 intra 或 inter，後面的 4096 儲存了 0～2^{12} - 1 的所有值，因為最長的 VLC 長度為 12-bits，所以最長的 VLC 為最大限制，且每個 VLC 均外加 1-bit 儲存正負號。

表 7. EVENT 和 REVERSE_EVENT 的結構成員

資料結構名稱	type	name	資料結構名稱	type	name
EVENT	uint8_t	last	REVERSE_EVENT	uint8_t	len
	uint8_t	run		EVENT	event
	int8_t	level			

和係數相關的 (run , level , last) 共有 102 種組合，這些組合儲存在 coeff_tab 陣列中，圖 33 為 coeff_tab 的部分陣列元素，因此，*DCT3D* 透過這 102 種組合對陣列作初

```
VLC_TABLE const coeff_tab[2][102] =
{    // intra = 0
                VLC          EVENT
             code len    last run level
    {    { { 2,  2}, { 0,   0,   1} },
         { {15,  4}, { 0,   0,   2} },
         { {21,  6}, { 0,   0,   3} }, . .  },
    // intra = 1
    {    { { 2,  2}, { 0,   0,   1} },
         { {15,  4}, { 0,   0,   3} },
         { {21,  6}, { 0,   0,   6} }, . . },
};
```

```
typedef struct {
    VLC vlc;
    EVENT event;
} VLC_TABLE;
```

```
typedef struct {
    uint32_t code;
    uint8_t len;
} VLC;
```

圖 33. coeff_tab 部分陣列元素

始化，初始化的程式碼如圖 34 所示，下一段將詳細說明如何利用 102 種組合來建立 *DCT3D* 陣列的所有元素，而 *DCT3D* 為 RLD 解碼所需要的 VLC table。

以圖 33 圈起來的部分為例，此時 intra = 0，code = 2 (即 Huffman Code = 10)，len = 2 (Huffman Code length) 時，已知最長的 Huffman Code 碼長為 12，但是目前碼長為 2，因此剩下的 10-bits 便可任意存放 0 或 1，轉換成整數即為 0～2^{10}-1，因此，*DCT3D* [0] [index] .len = 2 且 *DCT3D* [0] [index] .event = {0 , 0 , 1}，其中 index = 10xxxxxxxxxx，換成整數為 2048 + (0～2^{10}-1)，程式實現部分如圖 34。

```
for(intra = 0 ; intra < 2 ; intra++){
    for(i = 0 ; i < 102 ; i++){
        for(j = 0 ; j < (1<<(12- coeff_tab[intra][i].vlc.len)) ; j++){
            DCT3D[intra][(coeff_tab[intra][i].vlc.code <<
                          (12-coeff_tab[intra][i].vlc.len))| j ].len
            = coeff_tab[intra][i].vlc.len;
                                        index = 10 - xxxxxxxxxx
            DCT3D[intra][(coeff_tab[intra][i].vlc.code <<
                          (12-coeff_tab[intra][i].vlc.len)) | j ].event
            = coeff_tab[intra][i].event;
        }
    }
}
```

圖 34. 建立DCT3D 陣列的部分程式碼

DCT3D 陣列建立好之後，解 RLD 的步驟，如同先前解 mcbpc (**get_mcbpc_intra**) 函數提到的查表法，例如：當目前為 intra，且從位元流中取出的 12-bits 位元資訊為 010101-xxxxxx 時 (xxxxxx 中的 x 表示任意填入 0 或 1)，將 12-bits 轉成陣列位置 index，透過 DCT3D 查表得到 *DCT3D* [1][index].len = 6，*DCT3D* [1][index].event={ 0, 0, 6 }，其中 index = 010101xxxxxx，接著，就可以透過這些參數來進行解碼。

◆ 4. add_acdc()

將先前計算出來的係數預測值 predictors [8]，加上 RLD 之後的第一行或第一欄的係數，並且將加完之後的 DC 係數乘上 iDcScaler 參數，然後把 DC 係數以及 AC 係數的第一行和第一欄 (總共 15 個值)，儲存在 blocks 的 dec->mbs.pred_values[6][15] 結構中，以提供給接下來未解碼的 block，計算係數預測值之用。

◆ 5. dequant_h263_intra()

此函數為進行反量化 (Inverse Quantization : IQ) 運算的部份，至於反量化的原理是將各個原始係數除上量化參數後，取最接近的整數，而此量化後的整數就是我們需要傳

送的資料，而量化的目的是希望透過量化這個步驟，在影像品質能夠接受的情況下，將不重要的訊號大小降低，由於 DC 係數的值遠比 AC 係數大上許多，因此量化時所使用的 QP 也就不同，所以反量化時，DC 係數使用 iDcScaler 參數，而 AC 係數則使用 quant 參數，而詳細的程式實現如圖 35 的 pseudo code 所示，此解碼步驟將 porting 到副核心 SPE 上運作。

```
DC = DC_Q * iDcScaler

if ( quant is odd and AC_Q !=0 )
    |AC| = quant * (2*|AC_Q| + 1 )
else if ( quant is even and AC_Q !=0 )
    |AC| = quant * (2*|AC_Q| + 1 )-1
else
     AC = 0
if ( coefficient > 2047 )        coefficient =   2047
else if ( coefficient < -2048 ) coefficient = - 2048

where DC_Q and AC_Q are quantized DC and AC coefficient
```

圖 35. 反量化的 pseudo code

◆ 6. idct()

最後的一個解碼步驟就是反 DCT (Inverse Discrete Cosine Transform) 轉換，而 2-D DCT 的原理為，64 個係數經過 DCT 轉換之後，會將空間域數位影像資料轉換成頻率域，而 DCT 以及 IDCT 的轉換公式如下：

$$DCT: F(m,n) = \frac{2}{\sqrt{MN}} C(m)C(n) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N} \qquad (1)$$

$$IDCT: f(x,y) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} C(m)C(n)F(m,n) \cos\frac{(2x+1)m\pi}{2M} \cos\frac{(2y+1)n\pi}{2N} \qquad (2)$$

$$where\ C(m),C(n) = \frac{1}{\sqrt{2}}\ for\ m,n = 0\ ,\ and\ \ C(m),C(n) = 1\ otherwise$$

由於 DCT 具有可分離的 (separable) 特性，意思就是說它可以先執行一維離散餘弦轉換的列運算，再執行一維離散餘弦轉換的欄運算，此時的計算架構具有以乘累加的

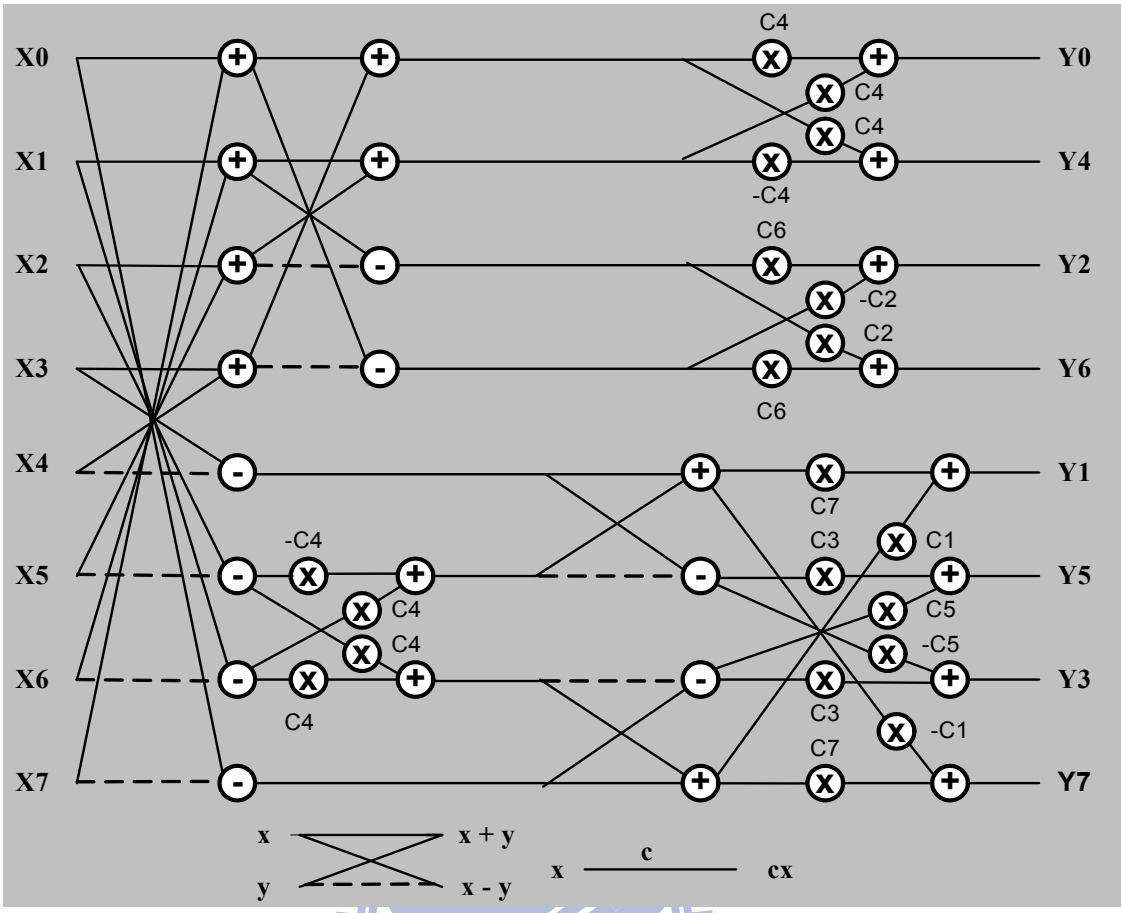


圖 36. FDCT 示意圖

特性，因此，這邊所採用的 DCT 則是利用此特性的 Chen's Algorithm，此演算法是一個非常有效率的快速演算法，圖 36 為 Chen's Algorithm 的示意圖。

由於它的可分離之特性，此解碼步驟則拆成兩部份，porting 在兩顆不同的 SPE 上運作。

◆ 7. transfer_16to8copy()

結束反 DCT 轉換之後，就結束整個完整的 intra block 係數值的解碼，因此，該函數便將已解碼的係數值，更新到目前畫面 (Current Frame) 的暫存區中，但是，更新之前，還須對所有係數值經過一些轉換處理。每個 block 的係數都會從 16-bit 轉換成 8-bit 大小，如果係數大於 255 就取 255，如果係數小於 0 就取 0，其他的值就保持原值，轉換結束之後，便存放該係數到所對應的暫存區，而 Y、U、V 的暫存區分別位於 dec->cur.y , dec->cur.u , dec->cur.v 中，等所有係數都轉換且更新結束之後，整個 I-Frame 的解碼就結束了，因此，該程式為 porting 至 SPE 的最後一個解碼部份。

3.4 Decoder for P-Frame (Predictive Frame)

對於 P-Frame 的解碼，位於 Xvid Codec 原始碼的 **decoder_pframe()**，圖 37 為整個程式的流程圖。

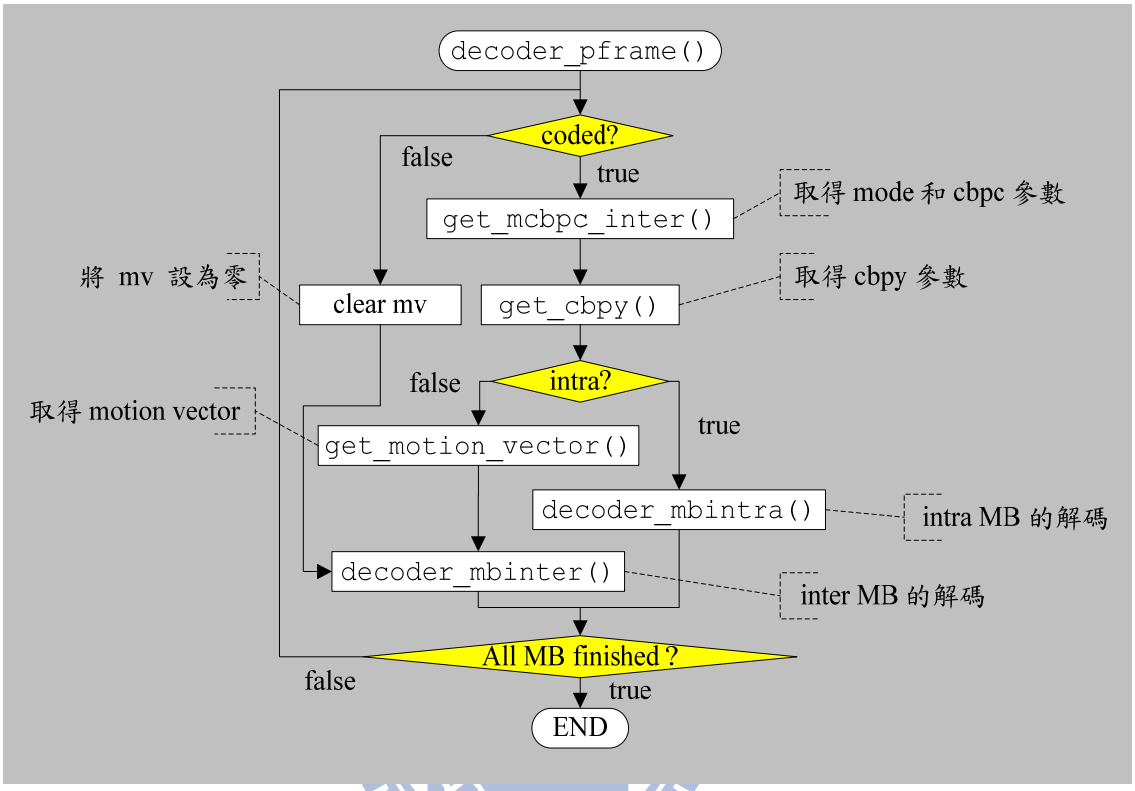


圖 37. decoder_pframe() 流程圖

3.4.1 get_mcbpc_inter()

此函數用來取得 mcbpc 參數，如之前提到的，此參數包含 MB 的 mode 和 U、V 的 block 是否有值的資訊 (即 cbpc)，但是在 P-Frame 中，mode 總共有五種，inter、inter + Q、inter + 4mv、intra 以及 intra + Q，而 U、V 的 block 是否有值也分為四種情況，所以總共有二十種結果組合，因此利用 table 來進行編解碼時，就需要較大的 table 來實現。

3.4.2 get_cbpy()

如 **3.3.2** 所提及的，intra 和 inter 的 cbpy 參數值，恰好是相反的，例如：查表得知 cbpy = 1010 時，intra 模式下，cbpy = 1010，而 inter 模式下，cbpy = 0101，圖 38 為 **decoder_pframe()** 的部分程式碼。

```
void decoder_pframe(){
    for ( y = 0 ; y < mb_height ; y++ ) {
        for ( x = 0; x < mb_width ; x++ ) {
            if (!(BitstreamGetBit(bs))) { // 此 block 已編碼
                mcbpc = get_mcbpc_inter(bs);
                mb->mode = mcbpc & 7;
                cbpc = (mcbpc >> 4);
                if (intra)  acpred_flag = BitstreamGetBit(bs);
                cbpy = get_cbpy(bs, intra);
                cbp =(cbpy << 2)|cbpc;
                if(mb->mode == MODE_INTER_Q || mb->mode == MODE_INTRA_Q){
                    quant += dquant_table[BitstreamGetBits(bs, 2)];
                    if (quant > 31)
                        quant = 31;
                    else if (quant < 1)
                        quant = 1;
                }
                mb->quant = quant;// 取得量化參數
                if ( mb->mode == MODE_INTER || mb->mode == MODE_INTER_Q ){
                    get_motion_vector(dec,bs,x,y,0,&mv,fcode,bound);
                    mb->mvs[0]= mb->mvs[1]= mb->mvs[2]= mb->mvs[3]= mv;
                    decoder_mbinter(dec,mb,x,y,cbp,bs,rounding,0,0);
                } else { // 模式是 MODE_INTRA
                    mb->mvs[0]= mb->mvs[1]= mb->mvs[2]= mb->mvs[3]
                                = {0,0}; // 清空 motion vectors
                    decoder_mbintra(dec, mb, x, y, acpred_flag, cbp, bs,
                                    quant, intra_dc_threshold, bound);
                }
            } else {  // 此 block 沒有被編碼
                mb->mode = MODE_NOT_CODED;
                mb->quant = quant;
                mb->mvs[0]= mb->mvs[1]= mb->mvs[2]= mb->mvs[3]= {0,0};
                decoder_mbinter(dec,mb,x,y,0,bs,rounding,0,0);
            }
        }                          此傳入參數為 cbp，因為沒有編碼，cbp = 0
    }
}
```

圖 38. decoder_pframe() 片段程式碼

3.4.3 get_motion_vector()

此函數的目的在於取得之後要作反移動補償 (Inverse Motion Compensation : IMC) 的 motion vector，而 motion vector 的目的為指向目前 MB 和參考畫面 (reference frame) 中最相似 (best matching) 的 MB 之相對位置，而當 mode 為 intra 時，這些 motion vector 會被預設為零。

由於 Y 是由四個 blocks 所組成，因此，當 mode 為 inter + 4mv 時，代表著 Y 有四個不同的 motion vector，而 mode 不為 inter + 4mv 時，即代表 Y 的四個 blocks，都是使用相同的 motion vector，圖 39 簡單標示出這些 motion vector 所代表的意義，

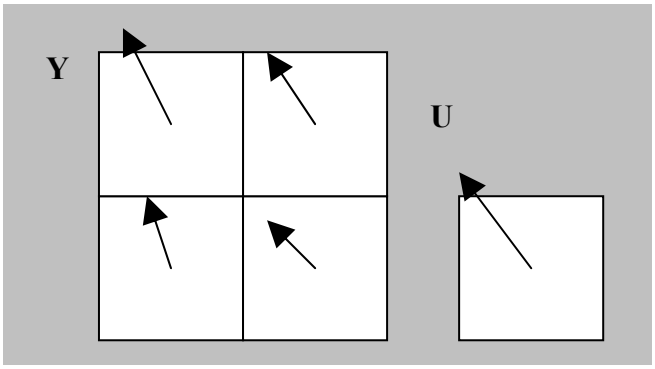


圖 39. motion vector 示意圖

而 U、V block 的 motion vector，是將 Y 的四個 motion vector 取平均。之所以選用 inter + 4mv 模式的目的在於有效率地減少剩餘能量 (residual energy)，但是，卻需要多儲存四個 motion vector 的資料在位元流中，因此，編碼時須比較這兩種模式 (inter + 4mv 或 inter) 的總能量大小來做選擇。

至於取得 motion vectors 的步驟如下：一、先從相鄰 block 的 dec->mbs.mvs 取得移動向量的預測值，二、從位元流中取得實際值與預測值之間的差值，三、將差值與預測值相加還原成原始移動向量值。

取完向量之後，就進到 inter block 解碼的 **decoder_mbinter()** 函數。

3.4.4 decoder_mbinter()

圖 40 為程式的流程圖，從流程圖可以得知，**interpolate16x16_quarterpel()**、**interpolate16x16_switch()**、**interpolate8x8_switch()** 以及 **interpolate8x8_quarterpel()** 這四個移動補償時的內插運算函數的選擇取決於是否為 inter + 4mv 以及是否為 quarter-pixel。

◆ 1. validate_vector()

在解 inter MB 時，motion vector 是非常重要的一個參數，因此，該函數目的就是將 motion vector 限制在有效的畫面範圍內，因為當位元流產生錯誤時，所取得的 motion vector 也是錯的，進而造成移動補償 (MC) 時，抓取到畫面以外的資料，所以算是一種防護機制。

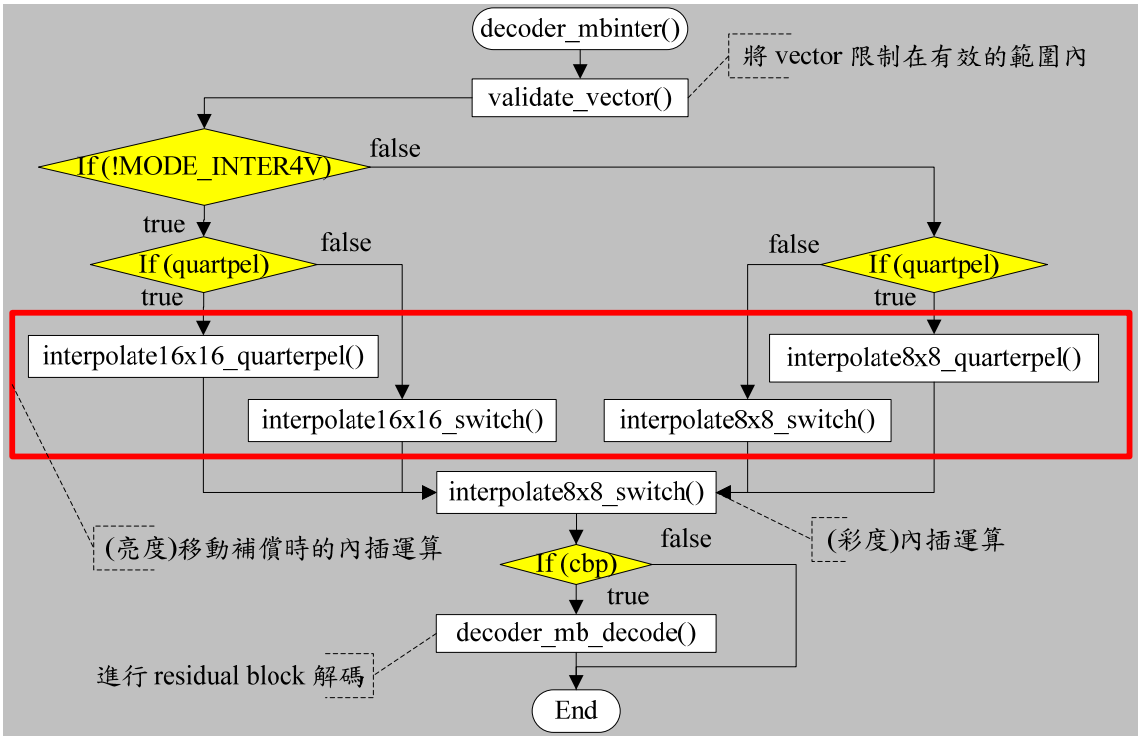


圖 40. decoder_mbinter() 流程圖

◆ 2. interpolate16x16_switch()

內插法可分為 half-pixel 和 quarter-pixel，這邊只針對 half-pixel 來做討論，與 half–pixel 相關的函數為 **interpolate16x16_switch()** 和 **interpolate8x8_switch()**，這兩個函數流程大致上都相同，一、透過 motion vector，到參考畫面中取得 reference block，二、將 block 上的資料經過內插法 (interpolate) 之後，複製到目前待解碼的 block 位置上，而唯一不同的為是否選用四個 motion vector。當 mode 為 inter + 4mv 時，代表著每個 block 各有自己的 motion vector，而程式內容如圖 41 所示，若 mode 為 inter

```
interpolate8x8_switch(dec->cur.y,dec->refn[0].y,16*x_pos,16*y_pos
                      , mv[0].x , mv[0].y , stride,rounding);
interpolate8x8_switch(dec->cur.y,dec->refn[0].y,16*x_pos+8,16*y_pos
                      , mv[1].x , mv[1].y , stride,rounding);
interpolate8x8_switch(dec->cur.y,dec->refn[0].y,16*x_pos,16*y_pos+8
                      , mv[2].x , mv[2].y , stride,rounding);
interpolate8x8_switch(dec->cur.y,dec->refn[0].y,16*x_pos+8,16*y_pos+8
                      , mv[3].x , mv[3].y , stride,rounding);
```

motion vector 的 x 方向　　motion vector 的 y 方向

圖 41. 模式為 inter + 4mv 的部分程式碼

時，表示 Y 的四個 block 均使用相同的 motion vector，此時程式內容如圖 42 所示。

```
void interpolate16x16_switch( cur,refn,x,y,dx,dy,stride,rounding )
{
    interpolate8x8_switch(cur,refn,x,y,dx,dy,stride,rounding);
    interpolate8x8_switch(cur,refn,x+8,y,dx,dy,stride,rounding);
    interpolate8x8_switch(cur,refn,x,y+8,dx,dy,stride,rounding);
    interpolate8x8_switch(cur,refn,x+8,y+8,dx,dy,stride,rounding);
}
```

圖 42. interpolate16x16_switch 部分程式碼

◆ 3. interpolate8x8_switch()

此函數為實際運作 half-pixel 內插運算的程式，half-pixel 的內插方式可以分為四種情形，如圖 43 所示，一、當 motion vector 指在整數位置上時，不需使用內插法，直接將 block 上的資料從參考畫面複製到目前畫面，二、當 motion vector 指向位置如圖 43 的 case 1 時，每一點像素值為上下兩個整數像素值相加取平均，三、當位置如圖的 case 2 時，像素值為左右兩個整數像素值相加取平均，四、當位置如圖的 case 3 時，像素值為相鄰的左上、右上、左下以及右下四個整數像素值相加取平均，而程式的實現如圖 44 所示。

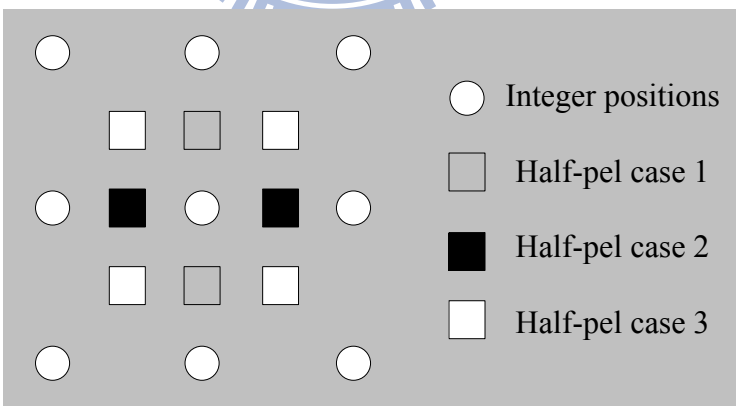


圖 43. half-pixel 內插示意圖

而 Y 的亮度內插運算做完之後，再針對 U、V 的 block 做彩度內插運算(**interpolate8x8_switch**)，接著判斷 cbp 參數來決定是否需要解 residual block (**decoder_mb_decode**)，若不執行就結束了 inter block 的解碼。

```
void interpolate8x8_switch( cur,refn,x,y,dx,dy,stride,rounding )
{
    uint8_t *src = refn + (int)((y + (dy>>1))*stride + x + (dx>>1));
    uint8_t *dst = cur + (int)(y*stride + x);
    switch(((dx&1)<<1)+(dy&1)){
        case 0:
            transfer8x8_copy(dst,src,stride);
            break;
        case 1:
            interpolate8x8_halfpel_v(dst,src,stride,rounding);
            break;
        case 2:
            interpolate8x8_halfpel_h(dst,src,stride,rounding);
            break;
        default:
            interpolate8x8_halfpel_hv(dst,src,stride,rounding);
            break;
    }
}
```

圖 44. interpolate8x8_switch 程式碼

◆ 4. decoder_mb_decode()

此程式為解出 residual block 的核心函數，圖 45 為部分程式碼，而圖 46 為程式的流程圖，residual block 指的是原本 block 的資料減去 reference block 之後所剩餘的係數資料，所以資料大小會遠比 **decoder_mbintra()** 所解的 intra block 小上許多，且從圖 45 的程式碼可以看出，想解出 residual block 的資料只需要三個函數。

```
static void decoder_mb_decode (dec, cbp, bs, pY_Cur, pU_Cur,
                               pV_Cur, pMB)
{
    for (i = 0; i < 6; i++) {
        if (cbp & (1 << (5 - i))) {
            memset(&data[0], 0, 64*sizeof(int16_t)); // 清空內容
            get_inter_block(bs, &data[0], direction, . . .);
            idct((short * const)&data[0]);
            transfer_16to8add(dst[i], &data[0], strides[i]);
        }
    }
}
```

圖 45. decoder_mb_decode() 部分程式碼

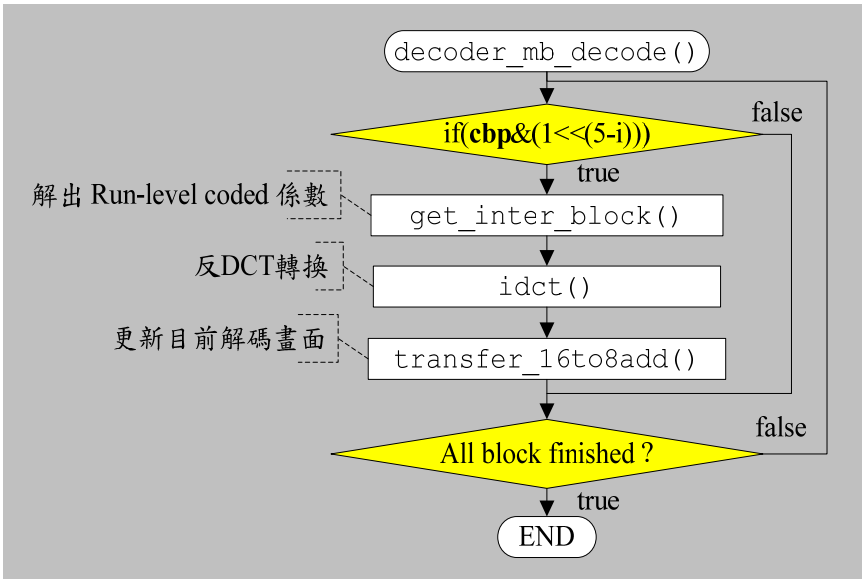


圖 46. decoder_mb_decode() 流程圖

(1) get_inter_block()

此函數的用途為 RLD 加上反量化，解碼的詳細步驟相當於解 intra block 的 **get_intra_block()** (RLD) 加上 **dequant_h263_intra()** (反量化)，唯一的不同只有在反量化的地方。反量化時，不論是 DC 係數還是 AC 係數，均是使用 quant 參數，而不須用到 iDcScaler 參數，且反量化的部份將 porting 到副核心 SPE 上運作。

(2) idct()

做完 VLD 和 IQ 之後，接下來就是進行 IDCT 運算，此解碼元件也是分成兩步驟在不同的 SPE 上執行。

(3) transfer_16to8add()

此函數會先取出位於 dec->cur 上已做好移動補償時的內插運算之 block 資料，再加上剛剛解好的 residual block，將相加之後的值從 16-bit 轉換成 8-bit 大小，如果係數大於 255 就取 255，如果係數小於 0 就取 0，其他的值就保持原值，轉換結束之後，存放該係數資料回 dec->cur 上的正確位置，等所有係數都轉換且儲存結束之後，整個 P-Frame 的解碼就完成了，圖 47 為示意圖。

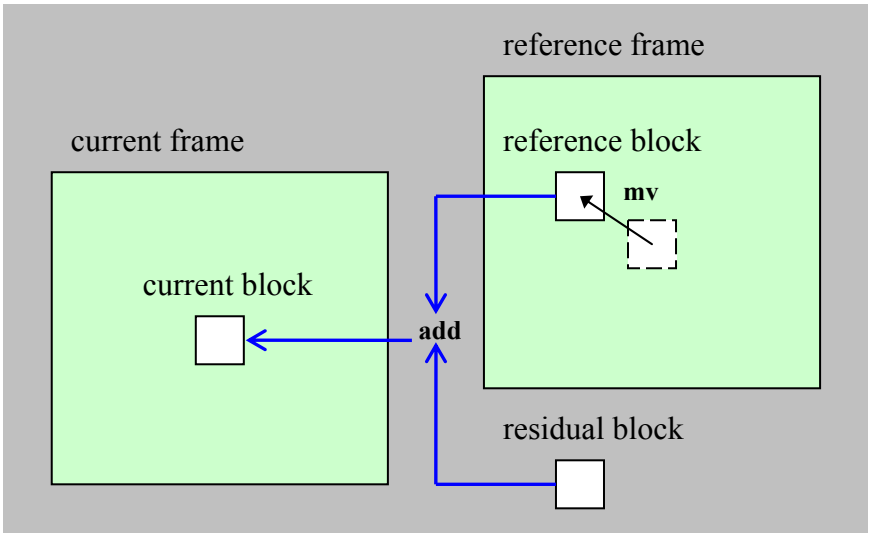


圖 47. 反移動補償 (Inverse Motion Compensation : IMC) 示意圖

3.5 Decoder for B-Frame (Bidirectionally predictive Frame)

對於 B-Frame 的解碼，位於 Xvid Codec 原始碼的 **decoder_bframe()**，圖 48 為整個程式的流程圖，其中，P-Frame 和 B-Frame 最大的不同在於，B-Frame 使用兩張參考畫面來預測出目前畫面，藉由雙向預測改善移動補償的壓縮效率。

程式的一開始，會先判斷 last_mb->mode (未來畫面的 MB mode) 是否有經過編碼，若沒有編碼，該 MB 的係數資料將直接從先前參考畫面之相同位置的 MB 取得，並完成該次 inter MB 的解碼，換句話說，位元流中完全不需要儲存該 MB 的任何資訊，此為特殊情況，故需要先判斷。若 last_mb->mode 為已編碼時，接著會讀入一個位元，來判斷 MB 解碼模式代號為 4 還是 0～3，表 8 為各個情況所對應 mode 數字代號。

表 8. B-Frame 預測方式之 mode 對照表

mb->mode	mode 數字代號	說明
MODE_DIRECT	0	雙向直接預測 (with delta mv)
MODE_INTERPOLATE	1	雙向內插預測
MODE_BACKWARD	2	逆向預測
MODE_FORWARD	3	順向預測
MODE_DIRECT_NONE_MV	4	雙向直接預測

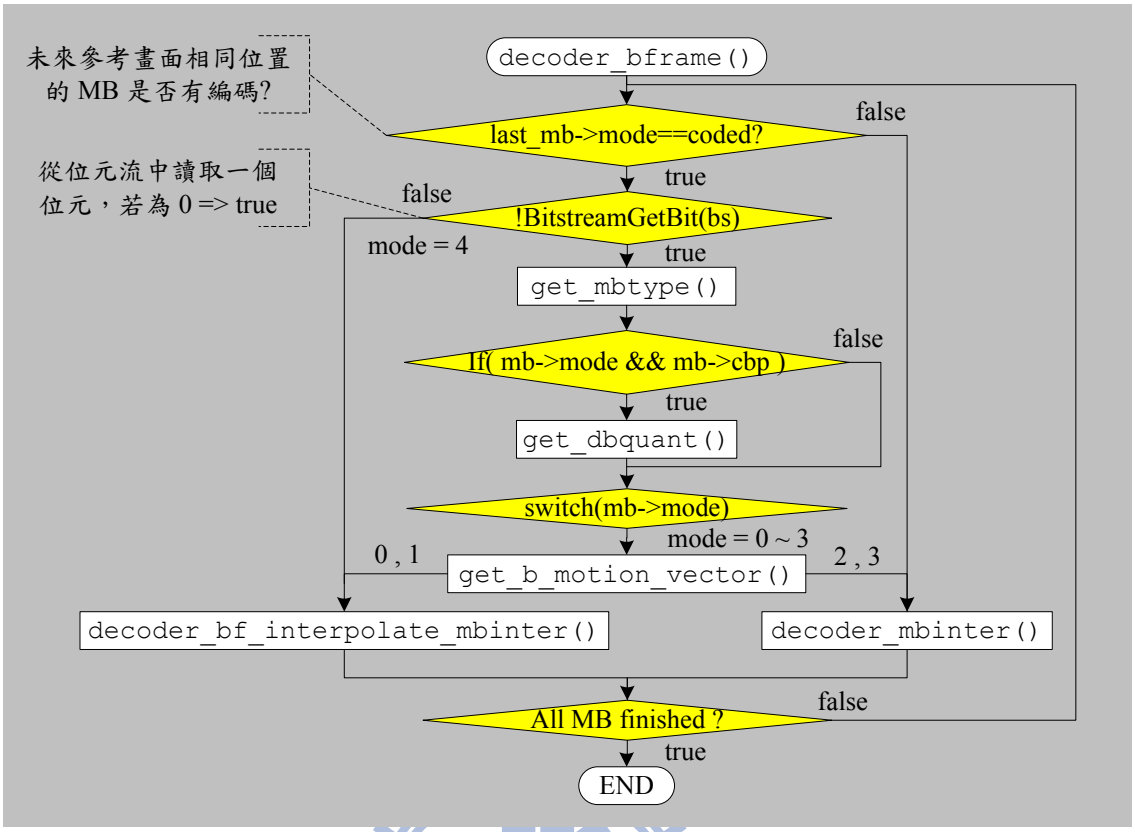


圖 48. decoder_bframe() 流程圖

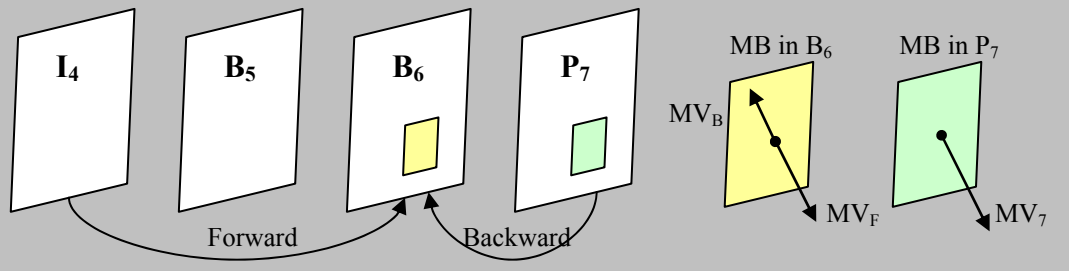
B-Frame 的 MB 解碼可以分為四種模式，圖 49 為程式碼的實現，此四種模式的解碼過程如下：一、順向預測 (Forward prediction)：透過先前畫面的 motion vector 抓取 reference block，並加上計算後的 residual block，二、逆向預測 (Backward prediction)：透過未來畫面的 motion vector 抓取 reference block，並加上計算後的 residual block，三、雙向內插預測 (Bidirectional interpolated prediction)：先由先前畫面的 motion vector 抓取 reference block，再從未來畫面的 motion vector 抓取 reference block，接著將這兩個 reference blocks 相加取平均，最後加上計算後的 residual block，四、雙向直接預測 (Bidirectional direct prediction)：此預測方式也是需要抓取兩個 reference blocks 取平均，但是，此預測方法的順向、逆向移動向量是由未來畫面之相同的 MB 所在位置上之 motion vector 推算得知，至於詳細的推算過程，如圖 50 的例子所示。

關於雙向直接預測，又可分為是否透過‘delta motion vector’來修正運算之後的向量，圖 49 中的 mv 變數即為 delta motion vector。

```
switch (mb->mode) {
    case MODE_DIRECT :
        get_b_motion_vector( & mv , zeromv );
    case MODE_DIRECT_NONE_MV :
        mb->mvs.x = last_mb->mvs.x * time_bp / time_pp + mv.x;
        mb->mvs.y = last_mb->mvs.y * time_bp / time_pp + mv.y;
        mb->b_mvs.x = (mv.x) ?  mb->mvs.x - last_mb->mvs.x
                    : last_mb->mvs.x * (time_bp - time_pp) / time_pp;
        mb->b_mvs.y = (mv.y) ?  mb->mvs.y - last_mb->mvs.y
                    : last_mb->mvs.y * (time_bp - time_pp) / time_pp;
        decoder_bf_interpolate_mbinter();
        break;
    case MODE_INTERPOLATE :
        get_b_motion_vector( & mb->mvs[0] , dec->p_fmv );
        get_b_motion_vector( & mb->mvs[0] , dec->p_bmv );
        decoder_bf_interpolate_mbinter();
        break;
    case MODE_BACKWARD :
        get_b_motion_vector( & mb->mvs[0] , dec->p_bmv );
        decoder_mbinter();
        break;
    case MODE_FORWARD :
        get_b_motion_vector( & mb->mvs[0] , dec->p_fmv );
        decoder_mbinter();
        break;
}
```

向量的預測值

圖 49. 預測 B-Frame 之 block 的部分程式碼



先前參考畫面 : **I_4** , display_time = 2　　目前 B 畫面 : **B_6** , display_time = 6

未來參考畫面 : **P_7** , display_time = 7

MV_7 = last_mb -> mvs , **MV_F** = mb -> mvs , **MV_B** = mb -> b_mvs

time_pp = display_time(P7) – display_time(I4) = 7 – 2 = 5

time_bp = display_time(B6) – display_time(I4) = 6 – 2 = 4

MV_D = mv (delta vector)

$MV_F = (time\_bp / time\_pp) \times MV_7 + MV_D$

if (**MV_D** == 0)

　　$MV_B = [(time\_bp - time\_pp) / time\_pp] \times MV_7$

else

　　$MV_B = MV_F - MV_7$

圖 50. MODE_DIRECT 範例以及相關的 pseudo code

3.5.1 get_mbtype()

從圖 48 的流程圖可以發現，當執行此函數時，即表示要從位元流中獲得該 MB 為何種解碼模式的資訊，且模式只可能為 0～3，而所讀取的位元資訊與模式的對照表如表 9 所示。

表 9. B-Frame mb_type 的解碼

位元流資訊	mode 數字代號	位元流資訊	mode 數字代號
0	0	001	2
01	1	0001	3

3.5.2 get_dbquant()

此函數用來調整量化參數，和 I-Frame 以及 P-Frame 不同的是，調整大小由 {-2,-1,1,2} 變成 {-2,0,2}。

3.5.3 get_b_motion_vector()

此函數用來取得 motion vectors，且為 B-Frame 專用，其求得移動向量的步驟如下：一、從傳入參數取得移動向量之預測值，如圖 49 圈起來的部份，可分為 zeromv、dec->p_fmv、dec->p_bmv 三種，二、從位元流中取得實際值與預測值之間的差值，三、將差值與預測值相加還原成原始移動向量值。

若傳入參數為 dec->p_fmv 或 dec->p_bmv 時，算出原始移動向量值後，還須將原始移動向量值存回 dec->p_fmv 或 dec->p_bmv (若傳入參數為 dec->p_fmv，就存回 dec->p_fmv)，以提供之後的 MB 預測移動向量之用。

3.5.4 decoder_bf_interpolate_mbinter()

圖 51 為程式的流程圖，此程式流程與 inter block 解碼函數 (**decoder_mbinter**) 大致相同，不同之處在於 B-Frame 使用兩張參考畫面進行移動補償時的內插運算，以及一些判斷條件的差異，因此，需多加 **interpolate8x8_add_switch()** 這個函數，此函數的用途會在下一段提到。

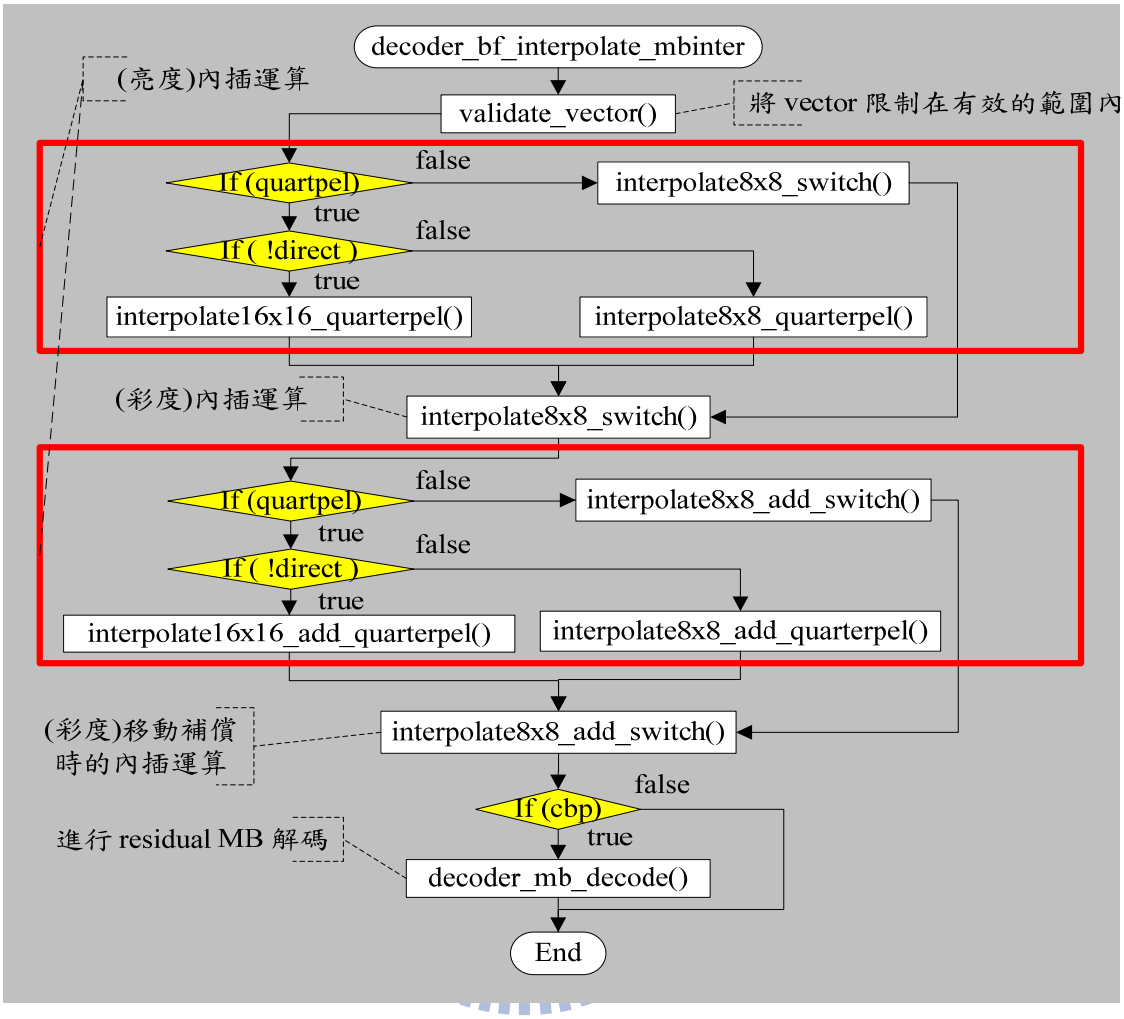


圖 51. decoder_bf_interpolate_mbinter 流程圖

◆ **interpolate8x8_add_switch()**

此函數透過 motion vector 到參考畫面抓取 block，抓取 block 資料時，要考慮 motion vector 指向整數位置還是半像素位置上，以決定選用內插的方式，接著加上目前畫面上的 block 資料後取平均，然後得到 reference block，最後透過 cbp 參數決定是否進入 **decoder_mb_decode()** 解出 residual block，並加至 reference block 計算出 inter block，等全部的 inter MB 都結束之後，整個 B-Frame 的解碼就完成了。

由此可知，此函數和先前的 **interpolate8x8_switch()** 大致相同，只是多加上目前畫面上的 block 資料，圖 52 為程式碼的實現。

```
void interpolate8x8_add_switch( cur,refn,x,y,dx,dy,stride,rounding )
{
    uint8_t *src = refn + (int)((y + (dy>>1))*stride + x + (dx>>1));
    uint8_t *dst = cur + (int)(y*stride+x);
    switch(((dx&1)<<1)+(dy&1)){
        case 0:
            interpolate8x8_halfpel_add(dst,src,stride,rounding);
            break;
        case 1:
            interpolate8x8_halfpel_v_add(dst,src,stride,rounding);
            break;
        case 2:
            interpolate8x8_halfpel_h_add(dst,src,stride,rounding);
            break;
        default:
            interpolate8x8_halfpel_hv_add(dst,src,stride,rounding);
            break;
    }
}
```

圖 52. interpolate8x8_add_switch 程式碼

第四章 MPEG-4 解碼的實現

此章將結合第二章以及第三章的內容，將單核心用的解碼程式，改寫成運作在多核心架構下的程式。實作上，雖然 PS3 有六顆 SPE 可供使用，這邊只使用了四顆來進行平行運算，而為了使 PPE 和其他四顆 SPE 進行平行解碼，必須讓 PPE 和 SPE 之間的資料可以互相傳遞，所以，SPE 要知道資料位於 PPE 的哪個位置或位於另一顆 SPE 的哪個位置，而這些位置就是記憶體空間上的絕對位址，此外，除了絕對位址的取得， PPE 和 SPE 之間的溝通管道也是一個研究的重心，因此，SPE 還須知道其他 SPE 上存放信號的位址，才可以使用 SPE 之間的通訊機制，為了儲存這些位址，我們便需要自訂一個資料結構來存下這些位址資訊。4.1 節說明哪些解碼元件改在 SPE 上運作、如何實現 pipeline 的設計流程以及解決資料覆蓋的問題；4.2 節則是針對自訂資料結構，以及 PPE 端和 SPE 端所撰寫的程式內容作深入的解說；4.3 節討論 porting 時，所遇到的程式問題。

4.1 多核心程式的設計

多核心的程式要點在於，可平行化的運算部份，透過 PPE 分配運算工作給 SPEs，讓 SPEs 獨立完成作業，藉此使得在相同的時間內，使用分工的技術提供較大的計算量，以達到縮短解碼的時間。

4.1.1 解碼元件的分配

經由第三章的研究，我們針對 intra block 以及 residual block 的解碼，找出可平行化的部分為反量化、反 DCT 以及更新解碼畫面之解碼元件，解 intra block 的程式中 (**decoder_mbintra**)，為 **dequant_h263_intra()**、**idct()** 以及 **transfer_16to8copy()**，而在解 residual block 的程式中 (**decoder_mb_decode**)，為 **get_inter_block()**、**idct()** 以及 **transfer_16to8add()**，接下來將詳細說明各個 SPE 進行哪些解碼步驟。

1. 在 SPE1 進行 **dequant_h263_intra()** 或部份 **get_inter_block()** 的運算內容，如第三章提到的，**get_inter_block()** 內含 RLD + IQ，因此，我們便將 **get_inter_block()** 程式修寫成只剩 RLD 解碼部分的 **vld_inter()** 函數，然後將 **vld_inter()** 留在 PPE，而 IQ 部份就放到 SPE1 上執行，換句話說，SPE1 就是負責反量化的解碼元件，如圖 53 所示。

2. 於 SPE2 和 SPE3 上，均是實現 IDCT 的解碼步驟，也就是 **idct()** 函數，如第三章提到的，IDCT 具有可分離性的特性，因此，為了使每個 SPE 的工作量相當，便將 IDCT 拆成兩步驟，分別由 SPE2 進行 IDCT 的前半段，而 SPE3 進行後半段。

3. 最後的 SPE4 則是執行 **transfer_16to8copy()** 或 **transfer_16to8add()**，此解碼元件的目的是 block 資料的輸出，也就是將完整的係數資料經過一些處理，再回傳至主記憶體的暫存區中，所以需要特別留意 2.3.4 提及的位址對齊之問題

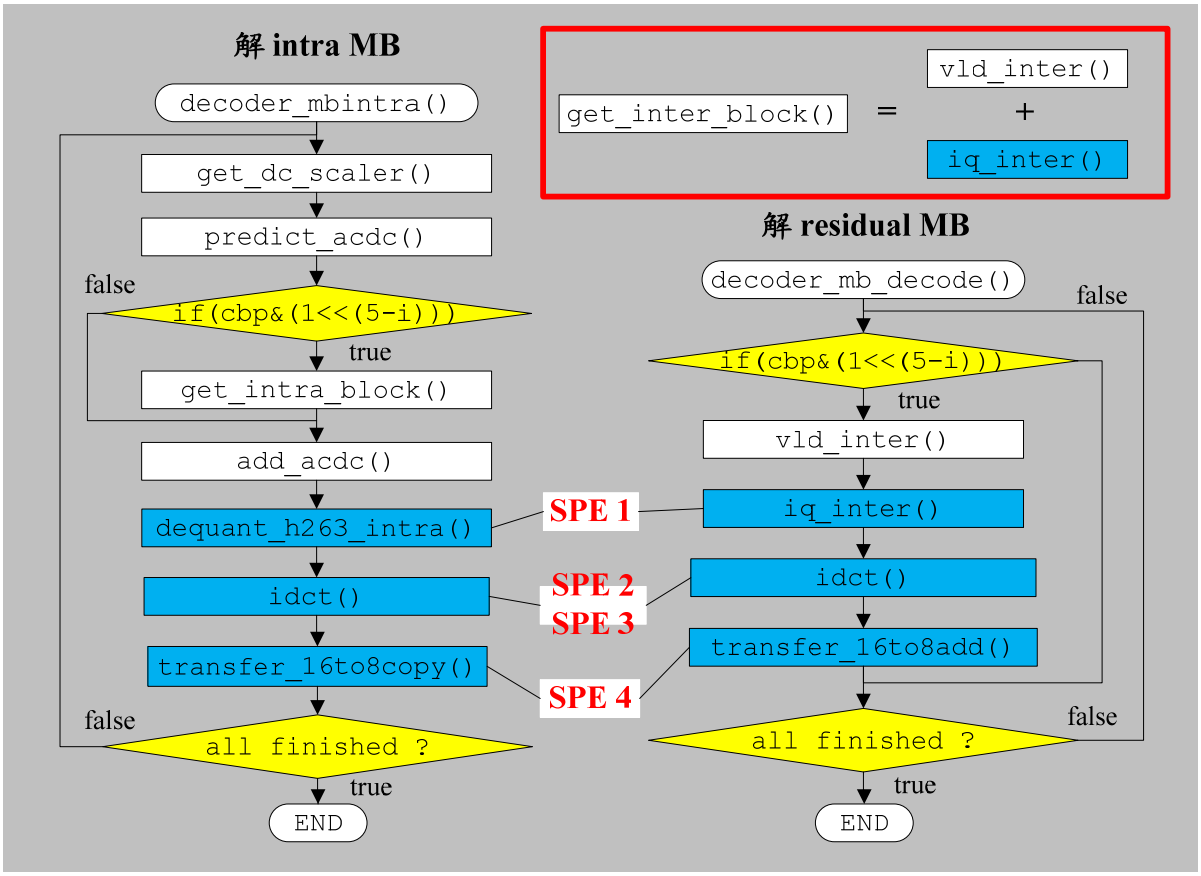


圖 53. SPE 所執行的解碼元件示意圖

4.1.2 pipeline 的流程設計

程式執行時，當 PPE 要指派工作給 SPE1 前，都會先去查看 SPE1 原本的工作完成沒，如果完成，就指派下一筆工作給 SPE1，而查看工作完成的機制，這邊利用 Mailbox 來實現。每當 SPE1 完成自己的工作之後，就會在 SPE1 的輸出信箱留下信息，而 PPE 就會去查看輸出信箱裡面是否留有信息，如果有，就傳送要做哪些 block 的信息到 SPE1 的輸入信箱，因為 SPE1 的讀取輸入信箱這個動作是阻塞式的，所以 SPE1 會一直等待有否信息進到輸入信箱，才會開始繼續動作。

而 SPE1 接收到信息並開始工作時，會先利用 DMA 將 PPE 端的資料複製到 LS，然後進行 SPE1 的運算，完成之後，一樣會先詢問 SPE2 工作狀況，而這邊改使用 Signal 來實現，原因是 SPE 之間，Signal 的溝通較為方便。每當 SPE2 完成自己的工作之後，就會寫出 Signal 信號到 SPE1 的 SNR2 (Signal Notification Registersig2)，而 SPE1 要將運算完的資料交給 SPE2 時，也都會先查看自己的 SNR2 是否留有信號，如果有，就告知 SPE2 有哪些 block 資料需要接手，而告知開始的方法，則以寫出 Signal 信號到 SPE2 的 SNR1，而由於讀取信號的動作也是阻塞式的，所以不用先查看信號是否有無資料在信號暫存區中，圖 54 為整個同步的詳細流程圖。

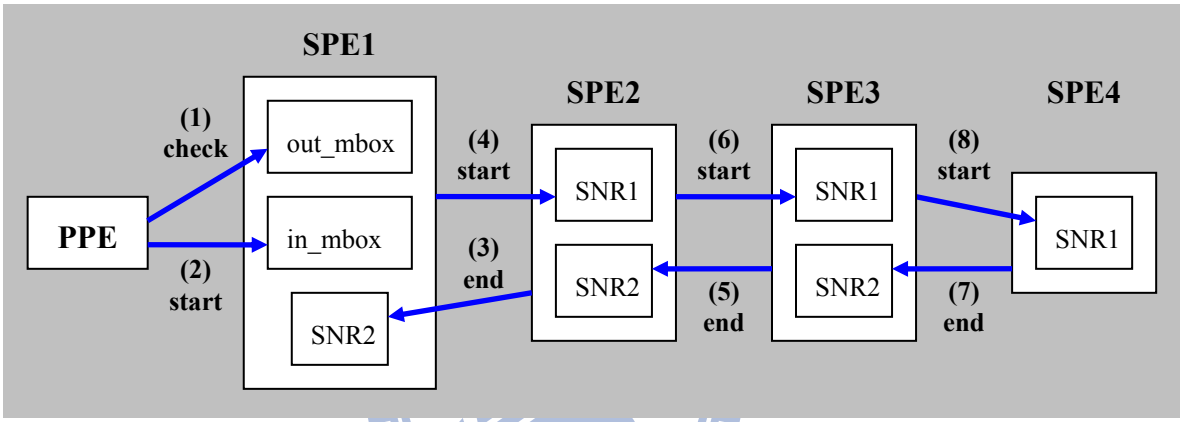


圖 54. 利用通訊機制實現 pipeline 之示意圖

從圖 54 可以知道，每個 SPE 均要知道前一個的 SNR2 位址，以及後一個的 SNR1 位址，簡單來說，SNR1 的信號有兩大功能，一、通知後一個 SPE 開始動作，二、信號中代表著有哪幾個 block 需要運算，而 SNR2 的信號功能，為目前 SPE 結束作業後，告知前一個 SPE 可以開始下一次動作。

在 SPE 端實現 pipeline 時，PPE 以及 SPE 之間的資料傳輸，其所需的資料內容有兩種，一、解碼時，解碼元件所需的參數，而這些參數儲存在自訂資料結構中，二、儲存 MB 的係數陣列，當進行 pipeline 的運作流程時，需要留意資料覆蓋所造成的問題，意思就是說後一層資料可能還未處理完就被前一層的資料所覆蓋掉，而造成錯誤，所以，便在 PPE 以及 SPEs 端使用兩個暫存區，一個是進行資料運算時，所存放的暫存區 A，一個是要交給下一層所存放的暫存區 B，而每當資料運算完之後，要先確定後一層的資料也運算結束，並且已經複製到暫存區 B，才可以將運算完之後的資料由暫存區 A 複製到暫存區 B。

實作上，PPE 以及 SPEs 端，不只是存放係數陣列需要使用兩個暫存區，連儲存必要參數的自訂資料結構也須宣告兩個變數，而自訂的資料結構內容將在下一節做詳細的說明，圖 55 為解決資料覆蓋問題之示意圖，data 為存放係數陣列的暫存區，而 parm 為自訂資料結構之變數暫存區，這邊需要特別留意的是，data_B 資料複製到 data_A 的 DMA 傳輸，均是由接收端主動去傳送端抓取資料，而非傳送端主動。

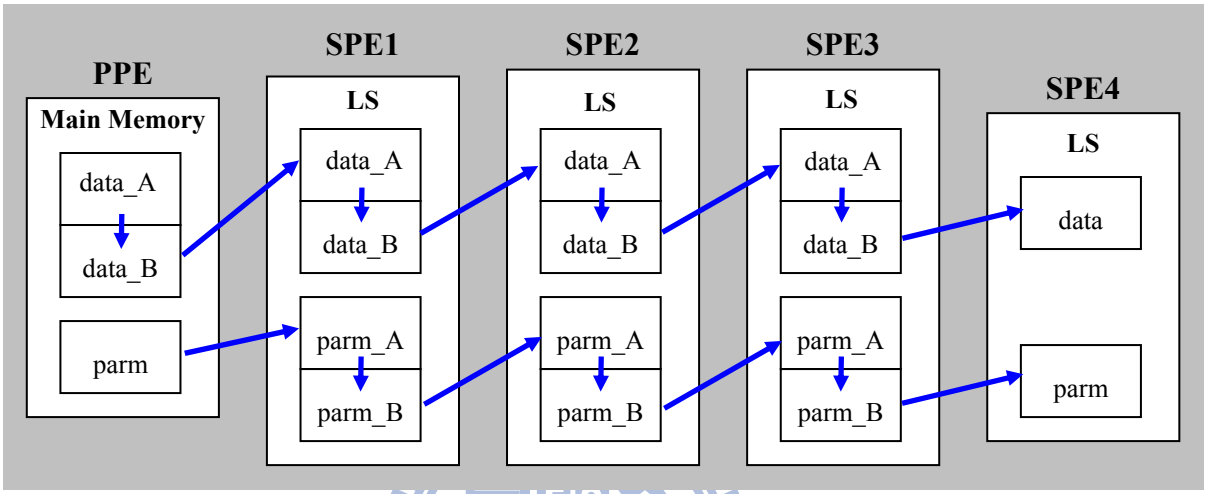


圖 55. 利用多個暫存區解決資料覆蓋之示意圖

4.2 自訂資料結構及程式的撰寫

一般解碼時，大多需要一些必要的參數來進行解碼，例如反量化時，便需要反量化參數，因此，在 SPE 端進行解碼時，必須從 PPE 端傳送必要的參數給 SPE 端，而這些 SPE 所需的必要參數便儲存在自訂的資料結構中。

若 PPE 端要傳送自訂的資料結構給 SPE 端時，所需步驟如下：一、先在 PPE 端建立自訂資料結構的變數，並將所有結構成員賦予值，二、在 SPE 端也建立相同資料結構之變數，此時，必須知道 PPE 端的結構變數之位址，三、使用 DMA 傳輸，並透過位址從 PPE 端把資料結構內容複製至 SPE 端的結構變數中。

關於自訂的資料結構將在 4.2.1 作詳細的討論。而為了實現多核心程式的運作，分別需要在 PPE 端修改部份程式碼，以及在 SPE 端重新撰寫程式碼，圖 56 為 PPE 端的完整解碼程式流程，對於 PPE 端的程式修改包含了 decoder_create()、decoder_destroy() 以及 decoder_iframe()、decoder_pframe、decoder_bframe 中的 decoder_mb_decode() 和 decoder_mbintra()，4.2.2 和 4.2.3 將針對 PPE 端以及 SPE 端所添加的程式碼作詳細地解說。

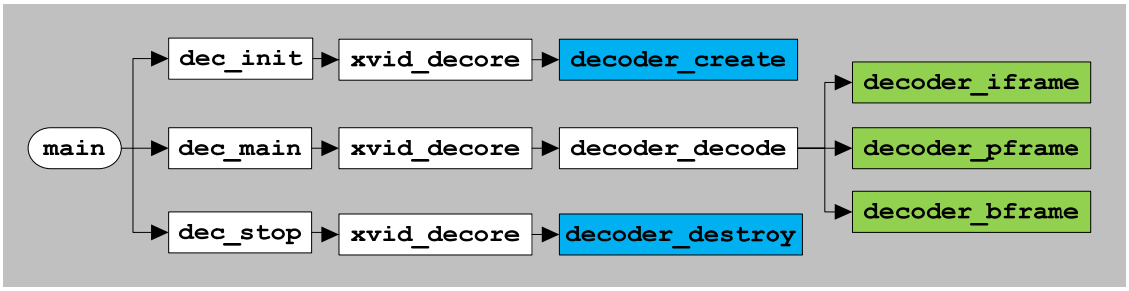


圖 56. PPE 端的完整解碼程式流程

4.2.1 自訂資料結構的分析

要在 SPE 端實現反量化以及更新解碼畫面的解碼步驟，必須從 PPE 取得必要的變數資訊，因此，自訂了 **parm_context** 的資料結構，而反量化只需用到 iQuant 和 mode，其他資訊均為更新解碼畫面所需的結構變數，表 10 為完整的結構成員。

表 10. parm_context 的結構成員

type	name	description
uint64_t	ea_y	Y 畫面的暫存區之確切位置
uint64_t	ea_u	U 畫面的暫存區之確切位置
uint64_t	ea_v	V 畫面的暫存區之確切位置
uint32_t	stride	Y 畫面的寬度大小
uint32_t	stride2	U、V 畫面的寬度大小
uint32_t	next_block	至下一個 block 的位址偏移值
uint32_t	iQuant	反量化參數
uint32_t	mode	判斷 intra 或 inter 的旗標
uint32_t	pad	為達到 16-byte alignment

◆ **ea_y、ea_u、ea_v**

當 SPE 端解完一個 MB 之後，便需要將 Y、U、V block 的係數資料傳回至 PPE 端，因此，此三個結構變數，為 PPE 端的 Y、U、V 畫面的暫存區之 MB 確切位置，圖 57 為計算 MB 確切位置的程式碼，而 x_pos 和 y_pos 分別代表 MB 的 (x , y) 座標值，如 (2 , 1) 表示該 MB 位於第三橫列的第二直欄位置，其中 0 代表第一個位置，且 dec->cur.y 即為存放 Y 畫面的暫存區之起始位址。

```
parm_context parm __attribute__((aligned(128)));
                                    計算出目前 MB 的位置
parm.ea_y = dec->cur.y + (y_pos << 4)* stride    + (x_pos << 4);
parm.ea_u = dec->cur.u + (y_pos << 3)* stride2   + (x_pos << 3);
parm.ea_v = dec->cur.v + (y_pos << 3)* stride2   + (x_pos << 3);
```

圖 57. 於 PPE 端的程式碼

◆ **stride、stride2、next_block**

此三個變數跟 Y、U、V 畫面存放於記憶體的位置有關，**stride** 為影片的寬度(長度單位為 pixel，且為 Y 畫面的寬度)，而 **stride2** 為 U、V 畫面的寬度，所以是 Y 畫面寬度的一半，且 **next_block** 為 8×**stride**。若為交錯式處理時，**stride2** 不變，但是，**stride** 為 Y 畫面寬度的兩倍，且 **next_block** 為 Y 畫面的一倍寬度，原因如圖 58 所示。

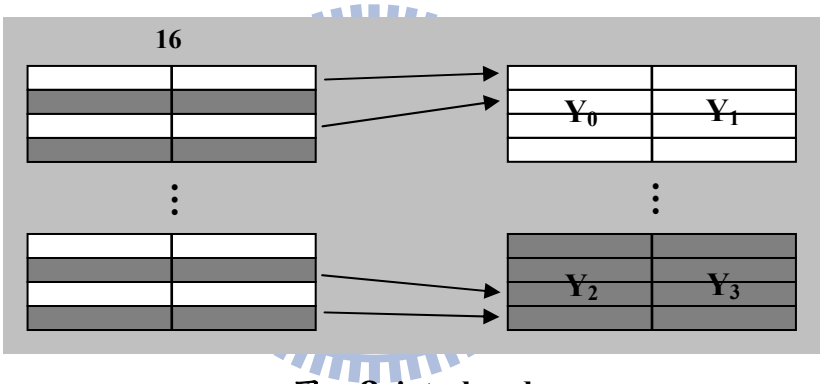


圖 58. interlaced

◆ **iQuant**

由於取得 DC 量化參數的程式 (**get_dc_scaler**) 也移植到 SPE1 上，因此，只需要傳送 AC 係數的反量化參數即可，此結構變數便是儲存反量化參數。

◆ **mode**

因為運作反量化以及更新解碼畫面時，須針對 intra 和 inter block 而有些許不同的解碼流程，此結構變數儲存該 MB 為 intra 或 inter 的訊息，其中，1 表示為 intra，0 表示為 inter。

如先前 4.1.2 提到的，多核心程式執行時，核心間的資料傳輸包含自訂結構變數以及 MBs 係數陣列，也就是必須了解資料位於核心內的正確位址，而通訊機制主要透過 Signal，所以，需要知道 SNR1 和 SNR2 的位址，因此，須自訂 **parm_addr** 的資料

結構儲存這些位址資訊，表 11 為完整的結構內容。

表 11. parm_addr 的結構成員

type	name	description
uint64_t	ea_parm	**parm_context** 結構變數的位址
uint64_t	ea_base	data_B 或 LS 起始位址
uint64_t	ea_sig1	SPE 的 SNR1 位址
uint64_t	ea_sig2	SPE 的 SNR2 位址

◆ **ea_parm**

此變數只有 SPE1 需要用到，它儲存位於 PPE 端的 **parm_context** 結構變數的位址，使得 SPE1 可透過此位址到 PPE 端拿取 **parm_context** 資料結構的所有內容。

◆ **ea_base**

參考圖 55 的示意圖，對於 SPE1 來說，此結構變數存下 PPE 端的 data_B 之陣列位址，而其他 SPE 均存下前一個 SPE 的 LS 起始位址，以提供之後計算出變數的正確位址之用。

◆ **ea_sig1、ea_sig2**

參考圖 54 示意圖，除了 SPE4 之外，每個 SPE 均須取得後一個 SPE 的 SNR1 位址，且除了 SPE1 之外，每個 SPE 均須取得前一個 SPE 的 SNR2 位址。

4.2.2 PPE 端程式的修改

要在原程式中使用 SPE 的子程式，並達到平行處理程式的效果，就必須使用執行緒 (Thread)，簡言之，當一個 PPE 的主程式執行時，可以建立多個 PPE 的執行緒，而每個執行緒又對應到一個 SPE 的執行緒，如圖 59 所示。由於 SPE 的執行緒是作

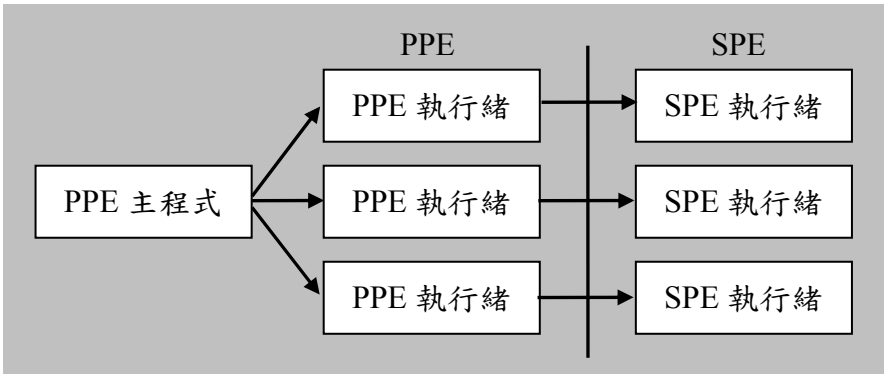


圖 59. Cell BE 執行緒示意圖

用在獨立的 CPU 上，CPU 之間的運作效能不會互相影響到，所以稱之為平行處理，且一個 SPE 上只能運作一個執行緒。

為了達到多核心的運作，啟動解碼程式時，不只要建立好執行緒，還需要將資料以及通訊相關的位址資訊先設定好，所以，我們在解碼初始化函數 **decoder_create()** 中加入建立 PPE、SPE 執行緒以及與 SPE 相關的位址初始化設定之程式碼，並在負責結束解碼的 **decoder_destroy()** 函數中加入銷毀執行緒以及 SPE 程式的程式碼。

◆ 1. 執行緒的建立

圖 60 為建立執行緒之程式碼內容，使用 **pthread_create()** 函數便可建立 PPE 執

decoder.c

```
typedef struct{
    spe_context_ptr_t speid;      // 如 2.2 提到的，此為辨別 SPE 的 handle
    pthread_t pthread;            // 建立 PPE 執行緒代碼
    void *argp;                   // 儲存 parm_addr 結構變數的位址
}spu_data_t;

spu_data_t spu[4];
parm_addr ctx[4] __attribute__((aligned(128)));

void *spu_pthread(void *arg){
    spu_data_t *input = (spu_data_t *)arg;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    spe_context_run(input->speid,&entry,0,input->argp,NULL,NULL);
    pthread_exit(NULL);
}

int decoder_create(xvid_dec_create_t * create)
{
    . . . . . .
    spu[i].argp = (void*)&ctx[i];
    pthread_create(&spu[i].pthread,NULL,&spu_pthread,&spu[i]);
    . . . . . .
}
```

辨別 SPE 的變數

儲存 parm_addr 結構變數的位址

函數指標

傳入參數

圖 60. 建立執行緒的部分程式碼 (於 PPE 端)

行緒，此執行緒會透過函數指標執行 **spu_pthread** 程式，並夾帶著傳入參數 **spu** [i]，進到 **spu_pthread** 程式後，使用 **spe_context_run()** 函數便啟動 SPE 執行緒，SPE 執行緒的完整建立流程在 2.2 節。

啟動 SPE 執行緒時，它會依照 **speid** 變數執行所對應的 SPE 程式內容，同時傳入 **argp** 的參數至 SPE 程式，如圖 61 中所選取的前兩個參數，此為啟動 SPE 執行

緒時的 **speid** 以及 **argp** 變數，有了 **argp** 所儲存的絕對位址，並透過 DMA 傳輸將 PPE 端的 **ctx** [0] 內的位址資訊複製至 SPE1 端 (**ctx** [1] ~ [3] 就是儲存 SPE2 ~ SPE4 相關的位址資訊)，有了這些位址資訊，多核心間的資料傳輸以及通訊問題就解決了。

spe1.c

```
parm_addr ctx __attribute__((aligned(128)));

int main( speid , argp , envp ){
    . . . . . .
    mfc_get(&ctx,argp,sizeof(ctx),tag_id[0],0,0);
    . . . . . .
}
```

圖 61. 利用傳入參數取的 parm_addr 的結構資料

◆ **2.** 位址的初始設定 (**parm_addr** 資料結構內的成員設定)

在解碼初始化函數 **decoder_create()** 中，還包含了位址的初始設定之程式碼，且位址資訊的初始化可分為兩類，SPE1 以及 SPE2 ~ SPE4，而詳細的 parm_addr 自訂結構變數設定如表 12 所示，由表中可知，只有 SPE1 完整知道 PPE 端存放係數陣列以及自訂結構變數的位址，而 SPE2 ~ SPE4 必須從前一個 SPE 的 LS 起始位址，加上 2.3.2 提到的資料相對位址偏移，方可獲得前一個 SPE 的係數陣列以及自訂結構變數之絕對位址，所以 SPE2 ~ SPE4 只存下前一個 SPE 的 LS 起始位址，而圖 62 是以 SPE2 要取得 SPE1 之資料的絕對位址為例的程式內容，下一段將對這些流程作詳細說明。

表 12. 位址的設定

SPE ID	member	description
SPE1	ea_parm	PPE 端的 **parm_context** 結構變數位址
	ea_base	PPE 端的 data_B 之陣列位址
	ea_sig1	SPE2 的 SNR1 位址
	ea_sig2	沒用處
SPE2 ~ SPE4	ea_parm	沒用處
	ea_base	前一個 SPE 的 LS 起始位址
	ea_sig1	後一個 SPE 的 SNR1 位址
	ea_sig2	前一個 SPE 的 SNR2 位址 (SPE4 除外)

decoder.c

```
parm_addr ctx[4] __attribute__((aligned(128)));   // 儲存 SPE1 ~ SPE4 相關的位址資訊
spe_context_ptr_t speid[2]; // [0] 代表 SPE1, [1] 代表 SPE2
uint64_t ea_ls_base;
uint32_t ls_offset;

ea_ls_base = (uint32_t)spe_ls_area_get(speid[0]); // (1) SPE1 的 LS 起始位址
ctx[1].ea_base = ea_ls_base;                      // (2)
. . . . . .
while( spe_out_mbox_status(speid[0])==0 );        // (4)
spe_out_mbox_read(speid[0],&ls_offset,1);    //  SPE1 係數陣列的 offset
spe_in_mbox_write(speid[1],&ls_offset,1,1); //  傳給 SPE2
while( spe_out_mbox_status(speid[0])==0 );
spe_out_mbox_read(speid[0],&ls_offset,1);    //  SPE1 自訂結構變數的 offset
spe_in_mbox_write(speid[1],&ls_offset,1,1); //  傳給 SPE2
```

spe1.c

```
short int data_A[6][64] , data_B[6][64];
parm_context parm_A, parm_B;

spu_write_out_mbox((uint32_t)data_B);    //  SPE1 係數陣列的 offset   (3)
spu_write_out_mbox((uint32_t)&parm_B);   //  SPE1 自訂結構變數的 offset
```

spe2.c

```
parm_addr ctx __attribute__((aligned(128)));
uint32_t ea_offset ;
uint64_t ea_data , ea_parm ;

. . . . . .
ea_offset = spu_read_in_mbox();      //  取得 SPE1 係數陣列的 offset   (5)
ea_data = ctx.ea_base + ea_offset;   //  計算 SPE1 係數陣列的絕對位址
ea_offset = spu_read_in_mbox();      //  取得 SPE1 自訂結構變數的 offset
ea_parm = ctx.ea_base + ea_offset;   //  計算 SPE1 自訂結構變數的絕對位址
```

圖 62. SPE2 ~ SPE4 取得位址之部份程式碼 (以 SPE1 和 SPE2 為例)

從圖中 PPE 端的第一行程式碼可以看到，由於 SPE 共有四個，所以需要宣告四個資料結構為 **parm_addr** 的變數 **ctx**，而 **ctx**[0] 負責儲存與 SPE1 相關的位址內容，而 **ctx**[1] ~ **ctx**[3] 為儲存 SPE2 ~ SPE4 之位址內容，接下來將以 SPE2 取得 SPE1 變數之絕對位址為範例，詳細解釋程式中如何利用 LS 起始位址計算出資料的絕對位址，其步驟如下：

(1) 先從 PPE 端使用系統函數得到 SPE1 的 LS 起始位址。

(2) 將 SPE1 的 LS 起始位址存至 PPE 端的 **ctx**[1]. ea_base 結構變數中，之後 SPE2 利用如圖 61 的方法，將資料結構內容從 PPE 端複製至 SPE2 端。

(3) 從 SPE1 利用 Mailbox 傳送相對位址偏移 (offset) 至 PPE 端。

(4) 利用 Mailbox 從 PPE 端接收 SPE1 所傳送之係數矩陣或自訂結構變數的相對位址偏移後，再傳送給 SPE2。

(5) SPE2 從 Mailbox 中得到相對位址偏移後，將 LS 的起始位址加上變數的位址偏移，就得到在 SPE1 上的係數矩陣以及自訂結構變數之絕對位址。有了此絕對位址，就可以透過 DMA 傳輸，從 SPE2 端到 SPE1 端存取資料。

SPE 執行緒和位址初始化設定好之後，便進入反覆的解碼過程，接下來將對於 intra block 解碼以及 residual block 解碼的程式修改作詳細說明。

◆ 3. decoder_mbintra() 程式的修改

圖 63 為修改後的程式碼，解碼步驟如下：

(1) 讓所有 block 先運算完反量化之前的解碼程序。

(2) 接著透過 Mailbox 得知 SPE 是否可接手之後的解碼步驟，若 Mailbox 的輸

```
static void decoder_mbintra(dec, pMB, acpred_flag, cbp, bs,quant){
    for (i = 0 ; i < 6 ; i++) {                                    (1)
        uint32_t iDcScaler = get_dc_scaler(quant, i<4);
        predict_acdc( .., predictors ,.. );
        if (cbp & (1 << (5-i)) )
            get_intra_block(bs,&data_A[i*64],direction, .. );
        add_acdc( .., &data_A[i*64], iDcScaler, predictors, .. );
    }
    // 當 SPE1 的輸出信箱不是空的，代表已經做完之前的工作，就指派 SPE1 去做
    if( spe_out_mbox_status(spu_data[0].ctx)!=0 ){
(2)     spe_out_mbox_read(spu_data[0].ctx,&status,1);
        parm.xxx  = xxx; // 儲存一些必要的參數，結構如表 8 所示
        memcpy( data_B,data_A,6*128 );                             (3)
        status = 63;      // 表示六個 block 均要進行運算
        spe_in_mbox_write(spu_data[0].ctx,&status,1,1);
    }else{
        for(i = 0 ; i < 6 ; i++){
            iDcScaler = get_dc_scaler(quant, i<4);
            dequant_h263_intra(.,&data_A[i*64],quant,iDcScaler,.);
(3)         idct((short * const)&data_A[i*64]);
        }
PPE     transfer_16to8copy(pY_Cur ,&data_A[0*64] ,stride); // 呼叫六次
        . . . . . .
    }
}
```

圖 63. decoder_mbintra() 修改後的部份程式碼

出信箱有信息，代表 SPE 有空閒，如果沒信息，表示 SPE 尚在作業中。

(3) 當 SPE 尚未處理完上一筆資料時，就由 PPE 自行完成之後的解碼步驟。若 SPE 已完成作業，就先將輸出信箱中的信息讀出，並將解碼用的必要參數設定好，然後再把係數資料由緩衝區 A 複製到緩衝區 B，以避免資料覆蓋的問題，最後告知 SPE1 要處理哪幾個 block (此旗標儲存在 **status** 參數中)，而 PPE 就可繼續運作下一個 MB 的解碼，不需要等待 SPE 完成之後的解碼步驟。

◆ 4. decoder_mb_decode() 程式的修改

圖 64 為修改後的程式碼，解碼步驟如下：

(1) 接著透過 Mailbox 得知 SPE 是否可接手之後的解碼步驟，若 Mailbox 的輸出信箱有信息，代表 SPE 有空閒，如果沒信息，表示 SPE 尚在作業中。

(2) 當 SPE 尚在作業時，就由 PPE 自行完成之後的解碼步驟。若 SPE 已完成作業，就先將輸出信箱中的信息讀出，並根據 **cbp** 的資訊讓該 block 運算完 RLD，然後再把係數資料由緩衝區 A 複製到緩衝區 B，等到 **cbp** 的資訊都取完後，再將解碼用的必要參數設定好，然後告知 SPE1 要處理哪幾個 block (此時旗標儲存在 **cbp** 參數中)，而 PPE 就可繼續運作下一個 MB 的解碼，不需要等待 SPE 結束。

```
static void decoder_mb_decode (dec, cbp, bs, pY_Cur, pU_Cur, pV_Cur, pMB)
{
    if(spe_out_mbox_status(spu_data[0].ctx)!=0){
(1)     spe_out_mbox_read(spu_data[0].ctx,&status,1);
        for(i = 0 ; i < 6 ; i++){
            if (cbp & (1 << (5-i)) ){
                vld_inter(bs, &data_A[i][0], direction, . . .);
                memcpy(data_B[i],data_A[i],128);
            }
        }
        parm.xxx  = xxx; // 儲存一些必要的參數，結構如表 8 所示
        spe_in_mbox_write(spu_data[0].ctx,&cbp,1,1);
    }else{
        for(i = 0 ; i < 6 ; i++){
            if (cbp & (1 << (5-i)) ){
(2)             get_inter_block(bs,&data_A[i][0],direction,...);
PPE             idct((short * const)&data_A[i][0]);
                transfer_16to8add(dst[i],&data_A[i][0],strides[i]);
            }
        }
    }
}
```

(2)

圖 64. decoder_mb_decode() 修改後的部份程式碼

4.2.3 SPE 端程式的修改

如 **4.1.1** 提到的，移植到 SPE 上執行的解碼部分為反量化、反 DCT 以及更新解碼畫面等三步驟，接下來將詳細說明在 SPE 的程式碼裡，添加了哪些通訊以及 DMA 傳輸相關的程式碼，以達到完整的實現。

◆ 1. 反量化

圖 65 為 SPE1 解碼元件的程式碼內容，為了使 SPE1 持續作業，於是使用 while 迴圈來實現，SPE1 的解碼流程步驟如下：

```
#define mfc_wait(x) mfc_write_tag_mask(1<<x);
                    mfc_read_tag_status_all();
int main(){
    . . . . . .                          // 一些初始化設定
    spu_write_out_mbox(1);               // 讓 PPE 知道 SPE1 可以開始動作
    while(1){
        cbp = spu_read_in_mbox();        // 接收 PPE 的 cbp 信息
        if(cbp == DATA_DONE ){           // 判斷 cbp 是否為結束信息
            sig[3] = DATA_DONE;
            mfc_sndsig(&sig[3],(uint32_t)ctx.ea_sig1,tag_id[0],0,0);
            mfc_wait(tag_id[0]);
            break;
        }
        mfc_get(&parm_A,ctx.ea_parm,sizeof(parm_A),tag_id[0],0,0);
        mfc_get(&data_A[0][0],ctx.ea_base,6*128,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
        quant = parm_A.iQuant;
        for( i = 0 ; i < 6 ; i++ ){
            if(cbp & (1<<(5-i))){
                if(parm_A.mode)
                    . . . . . .          // IQ – intra
                else
                    . . . . . .          // IQ – inter
            }
        }
        bell = spu_read_signal2();       // 等待 SPE2 結束上一筆工作的信號
        memcpy(data_B,data_A,6*128);
        parm_B = parm_A;
        sig[3] = cbp;                    // 告訴 SPE2 需要做哪幾個 blocks
        mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig1,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
        spu_write_out_mbox(1);           // 讓 PPE 知道 SPE1 的工作已結束
    }
    return 0;
}
```

(1) (2) (3) (4)

圖 65. IQ 的部份程式碼 (於 SPE1 端)

(1) 迴圈的一開始，先判斷接收信息是否為結束訊息，若為結束訊息，則傳送結束信號給下一個 SPE 並跳出迴圈，然後結束 SPE 程式。

(2) 若不為結束訊息，便利用 DMA 傳輸從 PPE 端取得係數陣列以及自訂結構變數的資料。

(3) 接著針對係數陣列作反量化運算。

(4) 等待收到 SPE2 的上一筆結束工作訊息，收到之後，便將係數陣列和結構變數存至另一個暫存區 B，然後傳送 **cbp** 訊息給 SPE2，叫 SPE2 開始下一步的解碼步驟，並讓 PPE 知道 SPE1 已結束工作。

```
int main(){
    . . . . . .              // 一些初始化設定
    bell = 1;                // 讓 SPE1 知道 SPE2 可以開始動作
    sig[3] = bell;
    mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig2,tag_id[0],0,0);
    mfc_wait(tag_id[0]);
    while(1){
        cbp = spu_read_signal1();    // 接收 SPE1 的 cbp 信息
        if(cbp == DATA_DONE ){       // 判斷 cbp 是否為結束信息
            sig[3] = DATA_DONE;
            mfc_sndsig(&sig[3],(uint32_t)ctx.ea_sig1,tag_id[0],0,0);
            mfc_wait(tag_id[0]);
            break;
        }
        mfc_get(&parm_A, ea_parm ,sizeof(parm_A),tag_id[0],0,0);
        mfc_get(&data_A[0][0], ea_data ,6*128,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
        for( i = 0 ; i < 6 ; i++ ){
            if(cbp & (1<<(5-i))){
                . . . . . .          //  IDCT – part1 (or IDCT – part2)
            }
        }
        bell = spu_read_signal2();  // 等待 SPE3 結束上一筆工作的信號
        memcpy(data_B,data_A,6*128);
        parm_B = parm_A;
        sig[3] = cbp;                // 告訴 SPE3 需要做哪幾個 blocks
        mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig1,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
        sig[3] = bell;               // 讓 SPE1 知道 SPE2 的工作已結束
        mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig2,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
    }
    return 0;
}
```

圖 66. IDCT_1 的部份程式碼 (於 SPE2 端)

◆ 2. 反 DCT

上頁的圖 66 為 SPE2 解碼元件的程式碼內容，大部分流程與 SPE1 相同，其中，紅色框起來部分改用 Signal 進行通訊，而 DMA 傳輸圈起來的絕對位址，也有所不同，如圖 62 的取得方法，因為 SPE3 和 SPE2 為 IDCT 的兩部份，因此整個程式碼內容皆相同，只有中間的 IDCT – part1 須改成 IDCT – part2。

◆ 3. 更新解碼畫面

圖 67 為 SPE4 解碼元件的程式碼內容，此 SPE 也是以 Signal 進行通訊，由於該 SPE 為最後一個步驟，因此，少了通知下一個 SPE 開始動作的訊息，且該程式除了從 SPE3 取得係數資料，還須負責將係數資料放回 PPE 端。

```
int main(){
    . . . . . .                          // 一些初始化設定
    sig[3] = bell = 1;                   // 讓 SPE3 知道 SPE4 可以開始動作
    mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig2,tag_id[0],0,0);
    mfc_wait(tag_id[0]);
    while(1){
        cbp = spu_read_signal1();        // 接收 SPE3 的 cbp 信息
        if(cbp == DATA_DONE)  break;// 若 cbp 為結束信息，跳出迴圈
        mfc_get(&parm_A, ea_parm ,sizeof(parm_A),tag_id[0],0,0);
        mfc_get(&data_A[0][0], ea_data ,6*128,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
        xxx = parm_A.xxx;                // 取出必要參數
        for( i = 0 ; i < 6 ; i++ ){
            if(cbp & (1<<(5-i))){
                if(parm_A.mode)
                    . . . . . .          // MEM_OUT – intra
                else
                    . . . . . .          // MEM_OUT – inter
            }
        }
        sig[3] = bell;                   // 讓 SPE3 知道 SPE4 的工作已結束
        mfc_sndsig( &sig[3],(uint32_t)ctx.ea_sig2,tag_id[0],0,0);
        mfc_wait(tag_id[0]);
    }
    return 0;
}
```

圖 67. 更新解碼畫面的部份程式碼 (於 SPE4 端)

要將係數資料存放回 PPE 端，必須先依照位址對齊的方式於 SPE 端擺放好，再透過 DMA 傳輸複製至 PPE 端的 Y、U、V 畫面之暫存區中的正確位置，如圖 68 所

示，這些正確位置則是存放在 **parm_context** 結構變數中的 ea_y、ea_u 以及 ea_v，圖中為 Y 畫面的記憶體之擺放示意圖。

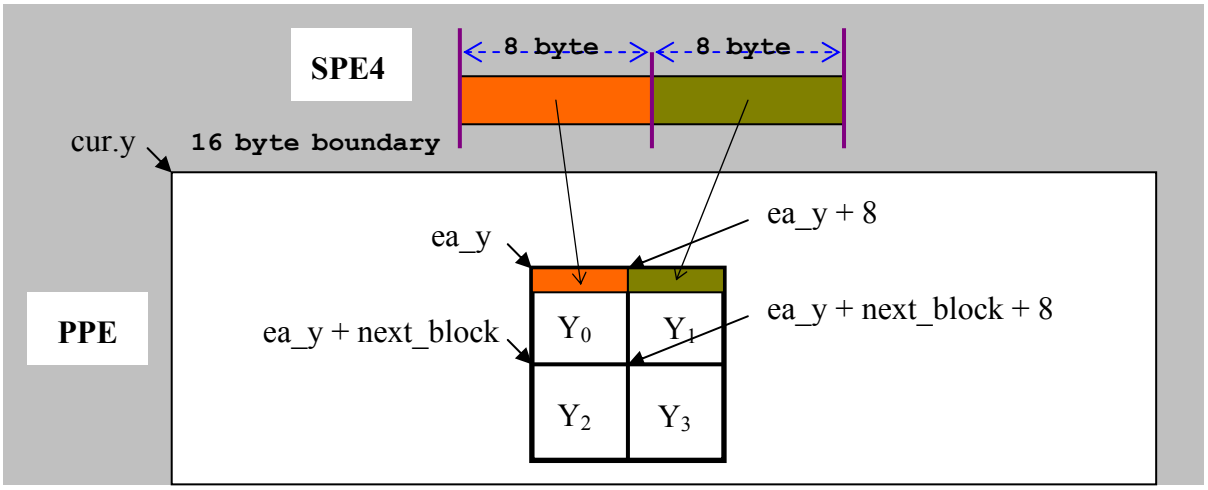


圖 68. 存放係數資料與位址對齊之示意圖

4.3 問題與討論

進行程式的 porting 時，除了 4.1 節提到的三個解碼元件之外，還嘗試過將計算預測值 (**predict_acdc**)、RLD (**get_intra_block**) 以及內插運算 (**interpolate8x8_switch**) 放至 SPE 上執行，實作中，遇到了記憶體不足以及資料相依的問題，因此沒有將這些解碼元件放到 SPE 端上運作，接下來將針對這些問題作分析與討論，以提供最佳化時，可從這些討論點開始著手。

4.3.1 資料相依性

計算預測值和 RLD 函數較難平行化的主要原因是，它們均具有資料相依性，對於計算預測值來說，它在運算過程中，需要用到大量的資料結構資料才得以解碼，而 RLD 方面，則是需要回傳解完後的位元流長度，以提供更新位元流位置的訊息，下面將更深入討論這兩個函數所遇到的問題。

◆ predict_acdc()

要計算出目前 block 之係數預測值，需用到相鄰 block 之係數預測值，也就是 dec->mbs[**mb_width** × **mb_height**] 內的資料結構變數，有了該結構變數資料，才可運算出目前 block 的係數預測值，因此，若想將此程式移植到 SPE 端上執行時，SPE 端

便需耗費 sizeof (MACROBLOCK) × mb_width × mb_height **bytes** 大小的記憶體，但是，SPE 端的區域記憶體只有 256 K bytes 的空間大小，所以無法實現，另一個考量點在於，此函數所佔的運算時間很短，除非 SPE 端的記憶體空間很大，否則其交換代價將沒辦法達到正比。

◆ **get_intra_block()**

要用 RLD 解出係數資料，必須從位元流中獲取資料，解碼步驟如下： SPE 端先從 PPE 端讀取一定大小的位元流資訊，進行係數資料的解碼，然後回傳已用去幾 bits 長度大小給 PPE 端，好讓 PPE 端可以對位元流位置做更新，以繼續之後的解碼步驟，換句話說，PPE 端雖然指派 SPE 端解碼，卻還是需要等 SPE 端回傳訊息，這樣一來，便沒有達到資料平行處理的觀念，且反而會降低解碼速度。

4.3.2 記憶體不足

由於 SPE 的區域記憶體大小只有 256 KB，且該記憶體還須包含程式碼大小，所以區域記憶體不可能存下整張畫面的資料，因此，移動補償時的內插運算便無法在 SPE 端上實現。

◆ **interpolate8x8_switch()**

移動補償時的內插運算需要抓取參考畫面上的資料，如果此參考畫面位於 PPE 端時，會因為反覆地從 PPE 端的主記憶體與 SPE 端的區域記憶體之間作資料搬移的動作，多花費了許多等待傳輸完成的時間，因此，將參考畫面儲存在 SPE 端，方可達到較高的解碼效率，但是，SPE 端的記憶體有限，故參考畫面的所有係數資料無法暫存至 SPE 端。

從第五章的實作中發現，此函數在 PPE 端的運作時間所佔的比例非常高，因此，若能找出方法解決記憶體不足的問題，並配合 residual block 的解碼，將整個反移動補償 (IMC) 的解碼步驟運作在 SPE 端後，所縮短的解碼時間，將會大大的提升。

第五章 實驗數據分析

此章著重在分析實驗數據，從中得知所縮短的解碼時間，並深入了解未來可以繼續改善的部份。5.1 節為分析解碼的實驗數據，得知當使用 SPE 核心協助解碼時，所縮短的解碼時間，以及 SPE 和 PPE 之間所佔的 MB 解碼數目比例；5.2 節則是在探討其它解碼元件在 PPE 端所佔解碼的時間比例，以及 DMA 傳輸大小和傳輸速度之間的影響，以幫助未來作最佳化的討論。

5.1 數據分析與比較

實驗中，利用 Xvid Codec 的解碼程式範例，配合 Mplayer 的影像輸出，每次解 150 張畫面，並取得平均解碼時間，關於實驗環境的建立如附錄所示，表 13 為多次測試後的平均數據。

表 13. 解 150 張畫面的平均解碼時間

Description	總解碼時間	反量化、反 DCT 以及更新解碼畫面	移動補償時的內插運算	其他解碼時間
Time (ms)	14.343	3.423	7.648	3.272
加入 SPE 後	13.005	2.085	7.648	3.272

表中的 14.343 ms 為反量化、反 DCT、更新解碼畫面、內插運算以及其他解碼步驟的總解碼時間，而加入 SPE 幫助解碼後，平均每張畫面解碼的時間從 14.343 縮短為 13.005，也就是少了 1.338 ms，如果只針對 porting 的解碼元件所改善的時間來做比較，相當於加快了 1.64 倍，也就是節省了 40 % 的解碼時間，且該部份大約佔了總解碼時間的 24 %，因此，若還想提升解碼效率，必須將移動補償時的內插運算，運作於另外兩顆尚未用到的 SPE 上。

從表中可以發現，移動補償時的內插運算佔了總解碼時間將近一半的份量，原因在於內插運算時需對每一點像素值作內插運算，且 Y、U、V 畫面位於記憶體中的儲存位置也影響了內插運算的複雜度，圖 69 為記憶體的儲存方式之示意圖，從圖可以發現，要對一個 block 作內插運算時，由於 block 中的每一橫列儲存於記憶體中是分散的，

因此沒辦法一次取出整個 block 的資料來進行內插運算，此為內插運算需要消耗較多解碼時間的原因。

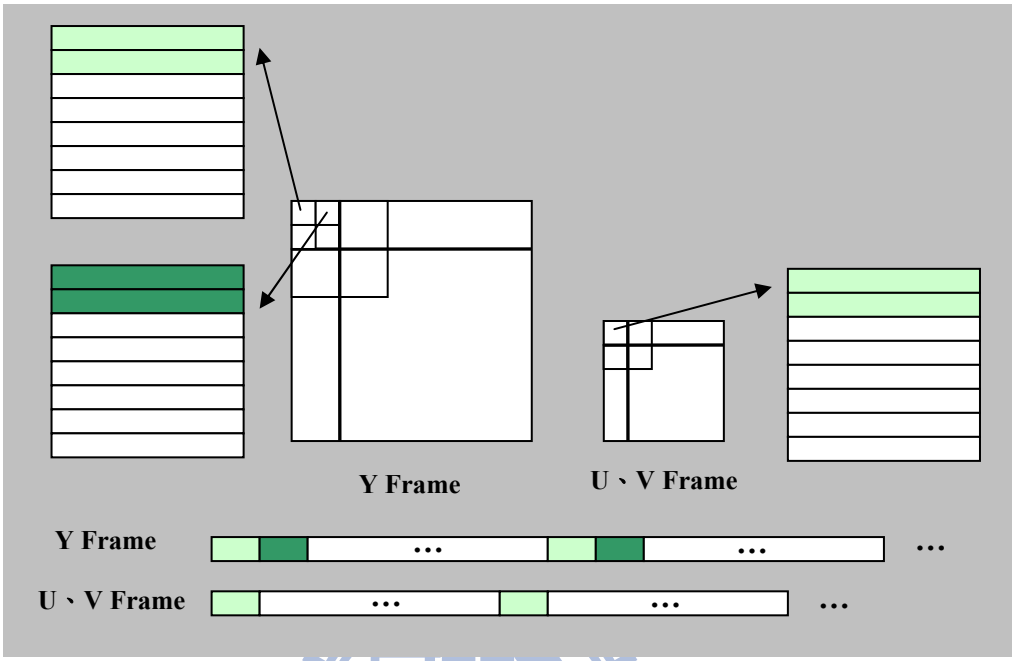


圖 69. Y、U、V 畫面位於記憶體中的儲存方式

而剩餘的其它解碼時間，包括了 RLD、係數預測值運算、 motion vector 的運算、Header 的讀取以及一些解碼程序的條件判斷等，接下來將針對 porting 的解碼元件，作深入的探討及分析。

解碼時，為了可以清楚看出一張畫面中，哪幾個 MB 是 PPE 處理的，哪幾個 MB 是 SPE 處理的，我們於寫出畫面時將 SPE 處理的 MB 的像素質設為 255，使得出現的 MB 圖像內容為全白以便區分，由於整張畫面大小為 640 × 368，所以共有 40 × 23 個 MB (也就是共有 920 個 MB)。

解 I 畫面時，如圖 70 所示，由於每個 MB 均是 intra MB，因此，一定會使用解 intra MB 函數 **(decoder_mbintra)**，所以共執行了 920 次該函數，其中 SPE 被指派了 424 次，也就是 PPE 與 SPE 處理 MB 的比例約為 1.17 : 1，相當於 SPE 幫 PPE 處理了將近一半的資料量。

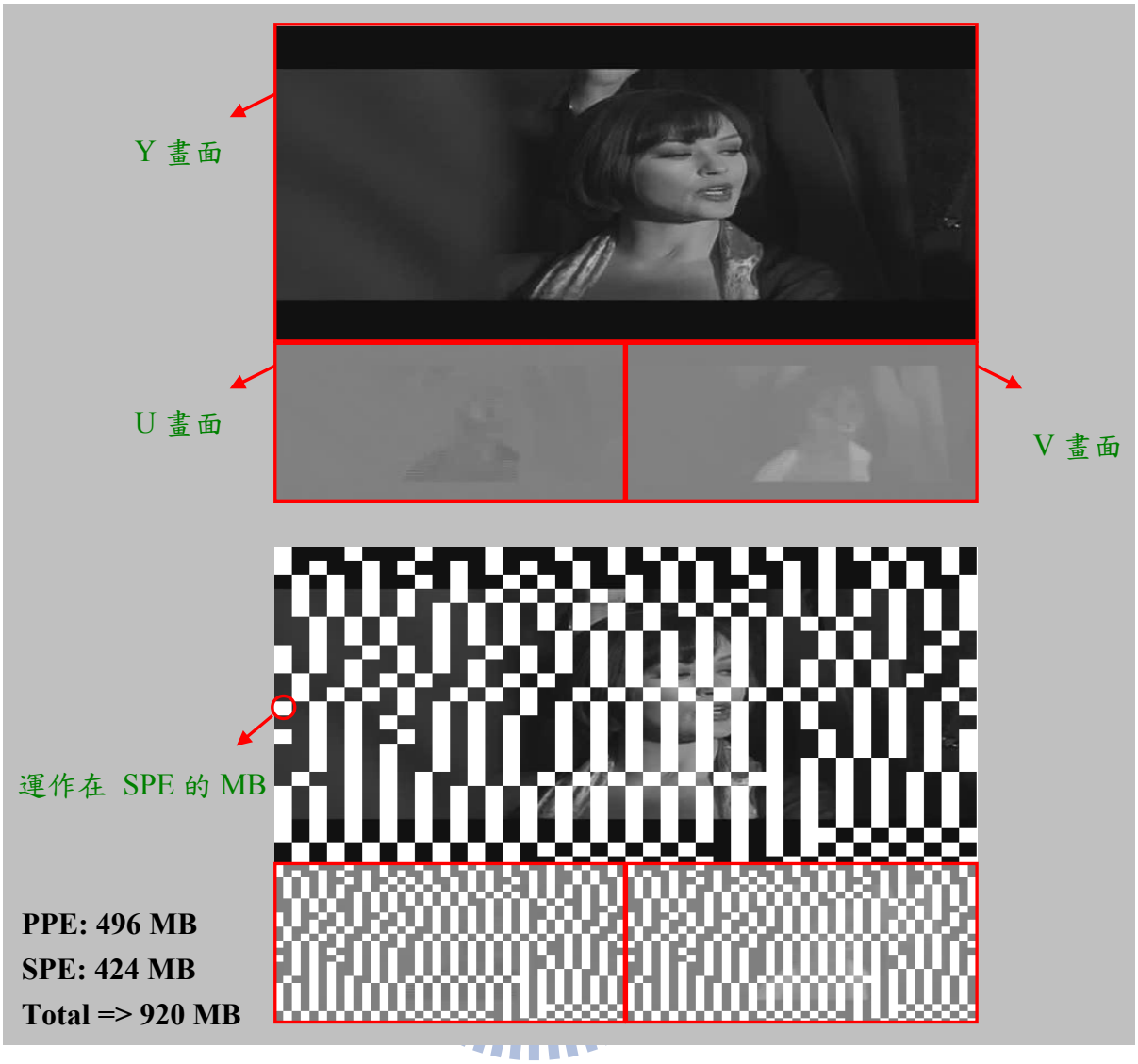


圖 70. SPE 運作在 I-Frame 之YUV 示意圖

解 P 畫面時，如圖 71 所示，但對於 P 畫面來說，部分 MB 是 intra MB、部分是 inter MB，且進行 inter MB 的解碼中，如果 residual MB 全為零時，也就不需使用 **decoder_mb_decode()** 函數，相當於不用 SPE 來協助解碼。

從圖 71 可知，解此 P 畫面時，920 個 MB 中，有 21 個 MB 是 intra MB，且 899 個 MB 是 inter MB，而且 899 個 MB 中，只有 415 個 MB 需要計算 residual MB，而 PPE 與 SPE 的處理比例約為 0.74 : 1，由於 SPE 幫忙解碼的 MB 不多，因此，所改善的解碼時間有限，下一節將針對這些實驗結果作討論。

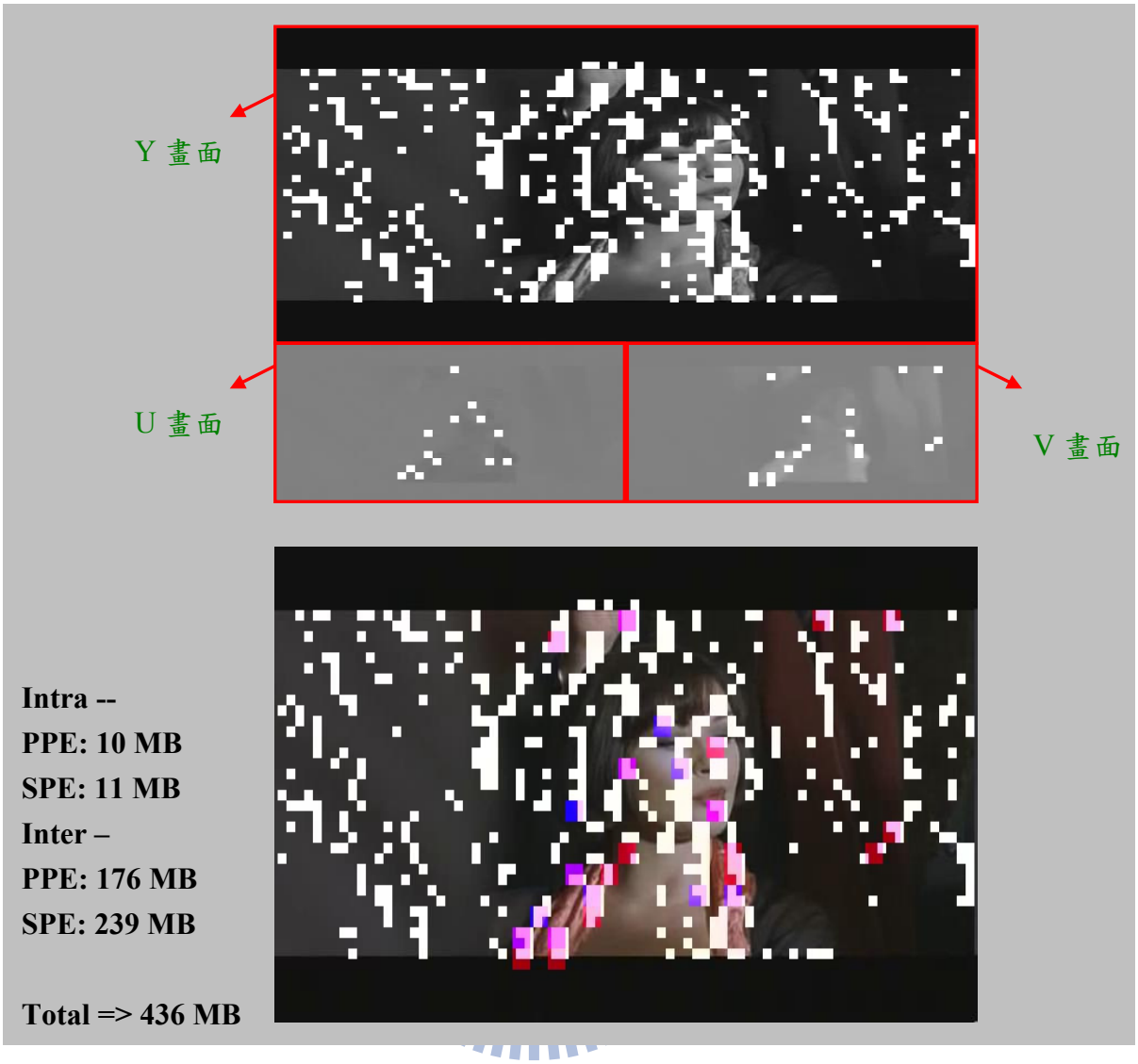


圖 71. SPE 運作在 P-Frame 之YUV 和 RGB 的示意圖

5.2 解碼元件的問題論述

由上節得知，解 I 畫面時，SPE 被指派解碼的比例較多，但是，對於一個 MPEG-4 的影片來說，畫面多數為 P 和 B 畫面，因此，所呈現出來的改善效率有限，所以，若想要縮短更多的解碼時間，必須對於整個解碼的流程作更詳細的了解，找出更多可平行運算的解碼元件，讓 SPE 可以多幫忙平行處理影像資料，並解決如 4.3 所提及的問題，除此之外，解碼元件所佔的解碼時間比例，也是一個值得參考的研究方向。

從表 13 的數據中發現，移動補償時的內插運算在 PPE 端的運作時間所佔的比例相當高，因此，若能克服 4.3.2 所提及的 DMA 傳輸問題，並移植至尚未使用的另外兩顆 SPE 上實現，一定能大幅縮短解碼的時間。

運作多核心程式時，核心間的資料傳輸是相當頻繁的一個動作，因此，為了使得核

心間的資料傳輸達到最高的傳輸效率，必須深入了解其傳輸大小所影響的傳輸時間。從其他文獻得知，由於 PS3 的 cache line 長度為 128-bytes，所以進行資料傳輸時，其傳輸大小最好是 128-bytes 的倍數，且位址對齊也為 128-byte alignment ，這樣才可使得傳輸速度達到最佳的效率。

關於 DMA 傳輸大小和傳輸速度之間的影響，以係數陣列的資料傳輸為例，每個 MB 內含六個 block，每個 block 又有 64 個係數，每個係數形態為 short int (大小為 2 bytes)，因此，一個 MB 的係數資料大小為 6 × 64 × 2 bytes，因此，我們可以選擇一次傳送 768 bytes 大小的資料，還是分六次分別傳送 128 bytes 大小的資料，表 14 為多次的傳輸時間測試，由此可知，有效率地使用 DMA 傳輸，也會影響其改善時間。

表 14. 測試傳輸時間

Description	Transfer Size (byte)	Time Cost (ms)
一次傳送	768	0.026090
六次傳送	6 × 128	0.049248
一次傳送	128	0.023785

第六章 結論與未來發展

6.1 結論

整篇論文是以 MPEG-4 的視訊解碼如何運作在多核心系統上為研究重心，二、三章深入討論多核心處理器的軟體架構以及視訊壓縮演算法的解碼流程，並在第四章說明如何實現多核心程式的運作來縮短解碼時間。

第二章介紹 PS3 中的多核心軟體架構，內容包含 SPE 程式的建立、核心間的資料傳輸以及通訊機制的使用方法等，研究內容主要著重在軟體的使用流程，並藉由所遭遇到的問題了解多核心的運作流程。第三章為分析 MPEG-4 的視訊解碼流程，說明解 I、P、B 畫面時，如何一一解出畫面中的各個 MB，以重建出原本的整張畫面，並藉由剖析流程找出可平行化的運算部分。

熟知 MPEG-4 的解碼原理後，將可平行運算的解碼步驟移至 SPE 上協同運算，而關於如何移至 SPE 上便在第四章作詳細的說明。當部份解碼步驟改在 SPE 上執行時，利用這些節省下來的時間，讓 PPE 端可以繼續下一步的解碼作業，簡單來說，就是運用副核心來協助主核心來進行解碼，經由第五章的實驗數據可知，多核心運作確實將每張畫面的解碼時間縮短約十分之一。

6.2 未來發展

在第五章實驗可觀察到，當 SPE 協助平行解碼時，確實縮短了解碼時間，不過為了讓效能達到最佳化，仍有幾個可以改善的地方：

◆ 將移動補償時的內插運算加到 SPE 上進行運算

若能解決 4.3.2 之記憶體不足的問題，將解碼時間比例較多的解碼元件移至 SPE 上執行，應該可以大幅縮短解碼時間，達到更好的效能。

◆ 讓每個 SPE 的解碼內容達到 Load Balance

Porting 解碼元件時，將各個解碼元件所佔的運算時間比例，平均落在各顆 SPE 上，簡言之，就是讓每個 SPE 的工作量相當。

◆ **使用大量的資料傳輸時，配合雙重緩衝技術**

當移動補償時的內插運算在 SPE 上執行時，便需要在主記憶體和區域記憶體之間作大量的資料傳輸，為了降低資料傳輸所造成的延遲，SPE 在傳遞資料應配合雙重緩衝技術來達到最佳效率。雙重緩衝技術如 2.3.3 中提及的，此技術可在 SPE 上發揮計算與傳輸平行化的優點。

參考文獻

[1] 林海波 謝海波 邵凌 王運洪 等編著, Cell BE 處理器編成指南, 電子工業出版社, 97 年 7 月

[2] Iain E. G. Richardson , H.264 and MPEG-4 VIDEO COMPRESSION , John Wiley & Sons , December 2003

[3] http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingPrimer.html

[4] D. A. Bader and S. Patel, "High performance MPEG-2 software decoder on the cell broadband engine," in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, 2008, pp. 1-10.

[5] http://publib.boulder.ibm.com/infocenter/systems/scope/syssw/index.jsp?topic=/eiccc/eiccckickoff.html&Open&S_TACT=105AGX16&S_CMP=LP

[6] http://book.51cto.com/art/200902/111495.htm

[7] http://twins.ee.nctu.edu.tw/courses/soclab_04/lab_hw_pdf/proj1_jpeg_introduction.pdf

[8] http://www.xvid.org/

[9] 笠野英松 著, MPEG-4 最新動態影像壓縮標準、引領數位視訊全方位運用, 李于青譯, 博碩文化股份有限公司, 2001 年 6 月

[10] Cell broadband engine programming tutorial, IBM ,version 2.1, 2007

[11] Cell Broadband Engine Programming Handbook, IBM, version 1.11, May 2008

[12] Cell broadband engine SDK libraries overview and users guide, IBM, version 2.1, 2007

[13] SPE runtime management library, IBM, version 2.1, 2007

[14] C/C++ language extensions for Cell Broadband Engine architecture, IBM, version 2.4, 2007

[15] Cell Broadband Engine architecture, IBM ,version 1.01, 2006

附錄一：實驗環境的建立

我們在編譯 Xvid Codec 的 Open Source 時，會在 Linux 系統之下將所有的程式碼都編譯成 object 檔，接著再將 object 檔合併成靜態 (Static) 連結檔及動態 (Dynamic) 連結檔，靜態連結檔為 libxxx.a 而動態連結檔則為 libxxx.so，最後將 Xvid 的標頭檔 (Header) 以及剛剛所生成的動態及靜態連結檔，放至 Linux 讀取 Codec 的路徑下，詳細的步驟如下：

1. 至 http://www.xvid.org/ 下載 Xvid Codec 的原始碼，這邊使用 1.2.1 版
2. 到 /xvidcore-1.2.1/build/generic/ 下依序輸入

   # make

   # make install

連結檔建立好之後，接著在原始碼中有一個解碼的程式範例，其用途是讀入 MPEG-4 壓縮格式的影片，透過上述建立好的靜態及動態連結檔來進行解碼，最後會得到 I420 (即 4Y1U1V) 的圖像資料，將一張張的 frame 輸出成 tga 或 pgm 的圖片格式，但是，我們需要的輸出方式是連續的影像輸出，而不是以寫成圖片的方式輸出，所以我們自行在程式範例中添加了一個功能，這功能可以解好的 I420 圖像資料，依照整張 frame 的 y、u、v 畫面順序，以 pipe 的方式輸出，所添加的 fifo 功能之程式碼內容如下圖所示。

位於 /xvidcore-1.2.1/examples/ 路徑下

```
#include <fcntl.h>

#define DATA_OUT "fifo"

static int write_out(unsigned char *image){
    int ret ,fd ;

    ret = mkfifo(DATA_OUT,0666);
    fd = open(DATA_OUT , O_WRONLY);
    write(fd , image , XDIM*YDIM + XDIM*YDIM/2 );
    return 0;
}
```

3. 添加完程式碼後，便到 /xvidcore-1.2.1/examples/ 下依序輸入

   # make all

   # ./xvid_decraw –i /opt/testfile/short.avi –fifo

第二條指令的 /opt/testfile/short/avi 為影片檔案的所在路徑

最後然後開啟 mplayer 去接收，並設定好長寬、每秒播出的張數，就可以成功秀出影片，關於 Mplayer 於 Linux 下的安裝方式，會在下一個附錄作說明。

4. 其中 Mplayer 安裝的所在位置在 /usr/local/mplayer/bin，所以至路徑下輸入

```
# ./mplayer /xvidcore-1.2.1/examples/fifo –demuxer rawvideo –rawvideo
  w=640:h=360 –fps 25
```

pipe 是處理 process 之間的大量資料傳輸，它是將一個 process 與另一個 process 連接起來的一種方式，其用途就是使 pipe 前面 process 之標準輸出導引至 pipe 後面 process 之標準輸入，換句話說，Xvid 的解碼範例之標準輸出為 y、u、v 畫面的順序，而 mplayer 之標準輸入就是 rawvideo 影片格式。

附錄二：Mplayer 的安裝

自行自網站下載 MPlayer-1.0rc2.tar 的壓縮檔，接著依序輸入

```
# tar -jxvf MPlayer-1.0rc2.tar
# cd    MPlayer-1.0rc2
# ./configure --prefix=/usr/local/mplayer --cc=gcc34 (or --cc=gcc4)
# make
# make install
```