1. JPEG 簡介

聯合影像專家小組（Joint Photographic Experts Group：JPEG）組織於 1986 年成立，其目的為制定靜態影像之數位壓縮與編碼標準。JPEG 標準是聯合影像專家小組（Joint Photographic Experts Group）於 1994 年所制定完成的標準，其正式名稱為 ISO/IEC 10918-1 | ITU-T Rec. T.81 "Information Technology ─ Digital compression and coding of continuous-tone still images"。基本上 JPEG 標準共有四種不同的工作模式：序向式以離散餘弦轉換為基礎之模式（Baseline Sequential DCT-based Coding）、漸進式以離散餘弦轉換為基礎之模式（Progressive DCT-based Coding）、序向式無失真之模式（Lossless Coding）、層次式（Hierarchical Coding）。一般俗稱的 JPEG 壓縮模式指的即是序向式以離散餘弦轉換為基礎之模式（Baseline Sequential DCT-based Coding）。此次實驗所實現的 JPEG 編解碼系統即是實現序向式以離散餘弦轉換為基礎之模式（Baseline Sequential DCT-based Coding）之 JPEG 編解碼系統。

1.1 色彩座標轉換

由於常見的影像位元流是採用 RGB 的色彩座標，然而，在影像處理中使用的色彩模型為 YC_bC_r 此種色彩模型。因此，需要作色彩座標的轉換，其轉換公式如下：

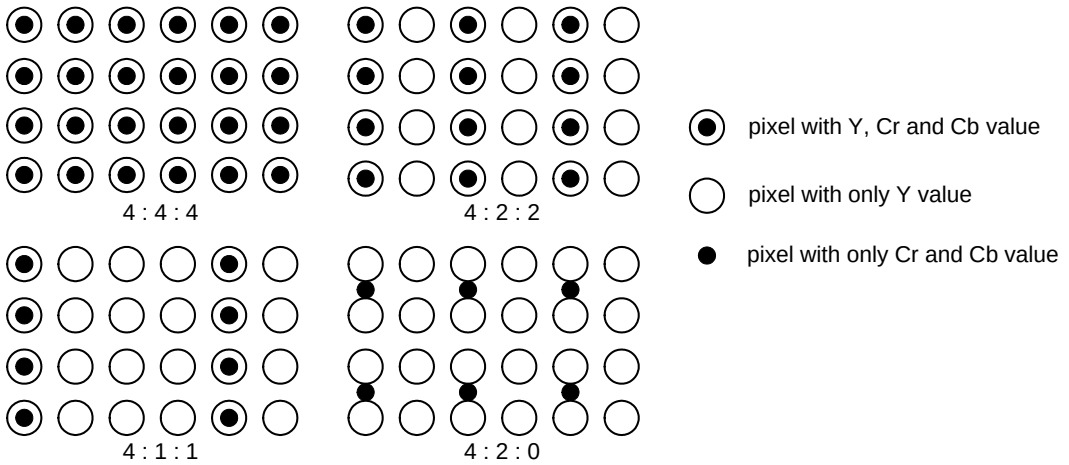
$$\begin{cases} Y = 0.299R + 0.587G + 0.114B \\ C_b = -0.168R - 0.331G + 0.499B \\ C_r = 0.5R - 0.419G - 0.081B \end{cases} \quad \cdots \quad (1)$$

透過此轉換公式，能夠將 RGB 的色彩座標轉換成 JPEG 影像處理中採用的 YC_bC_r 色彩座標。要作此色彩座標的轉換目的在於，人類的視覺對於亮度（luminance）比較敏感，而 YC_bC_r 色彩座標中的 Y 分量就是代表了亮度；而對於彩度（chrominance）則比較不敏感（YC_bC_r 色彩座標中的 C_b 以及 C_r 分量即是彩度）。

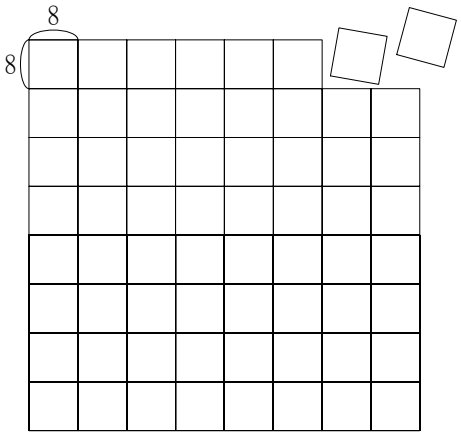
1.2 JPEG 編解碼系統

經過色彩座標轉換成為 YC_bC_r 色彩模型後，接下來就是作像素的取樣，常見的取樣的方法有 4:4:4、4:2:2、4:2:0 等等幾種取樣方法，如圖一所示。由於 JPEG 影像編解碼是利用區塊編碼（Block Coding）的概念完成的，因此，經過取樣後，將影像切割為一個個 8x8 大小的區塊，如圖二。接下來就是整個 JPEG 編碼的流程，如圖三所示。首先將每個 8x8 的區塊作二維離散餘弦轉換（2-D **D**iscrete **C**osine **T**ransform：2-D DCT），將像素值轉換為頻率域的值。由於，人的視覺對於低頻信號靈敏程度遠高於高頻訊號。因此，透過二維離散餘弦轉換之後的量化（**Q**uantization：Q）程序，將高頻值大幅量化為零，盡量只保留低頻的值，此目的是為了在不犧牲影像品質之下，透過資料之值具有大量的零值，可達到資料壓縮的目的。經過量化之後，JPEG 將 DC 值利用差分編碼（**D**ifferential **P**ulse **C**ode **M**odulation：DPCM）的方法編 DC 的值；而 AC 值部分則先利用 Zig-Zag 掃描方式，依序將 AC 值掃描完畢，並利用變動長度編碼（**R**un-**L**ength **C**oding：RLC）依序編碼。最後再經過霍夫曼編碼（Huffman Coding）以及加入 JPEG 標準的檔頭即成為一 JPEG 位元流。

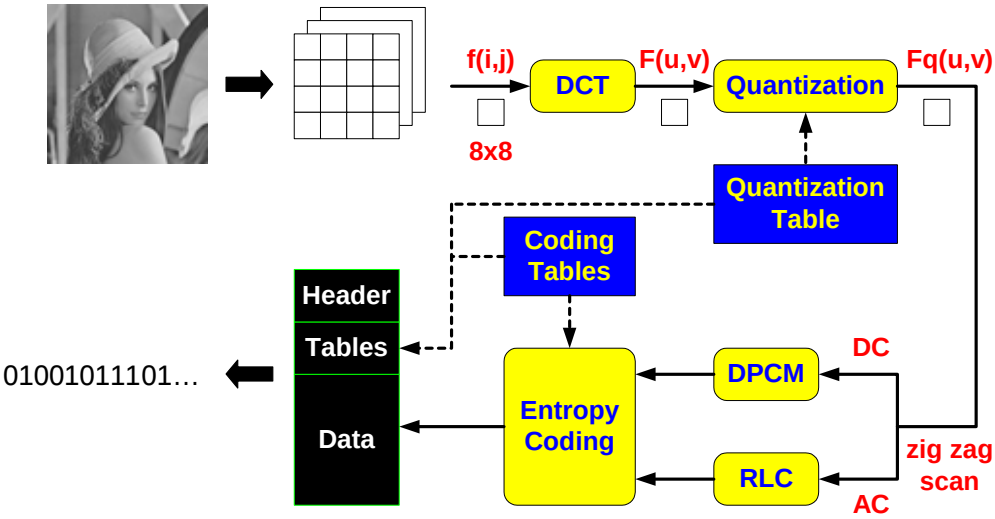
而 JPEG 解碼系統和編碼系統類似，只是整個解碼流程剛好是編碼流程的顛倒。一開始，先處理檔頭，接下來先做完可變長度解碼後（**V**ariable **L**ength **D**ecoding：VLD），再經過反量化（IQ）以及反二維離散餘弦轉換（IDCT），最後再作 YC_bC_r→RGB 的色彩座標轉換即可將影像重建，如圖四所示。



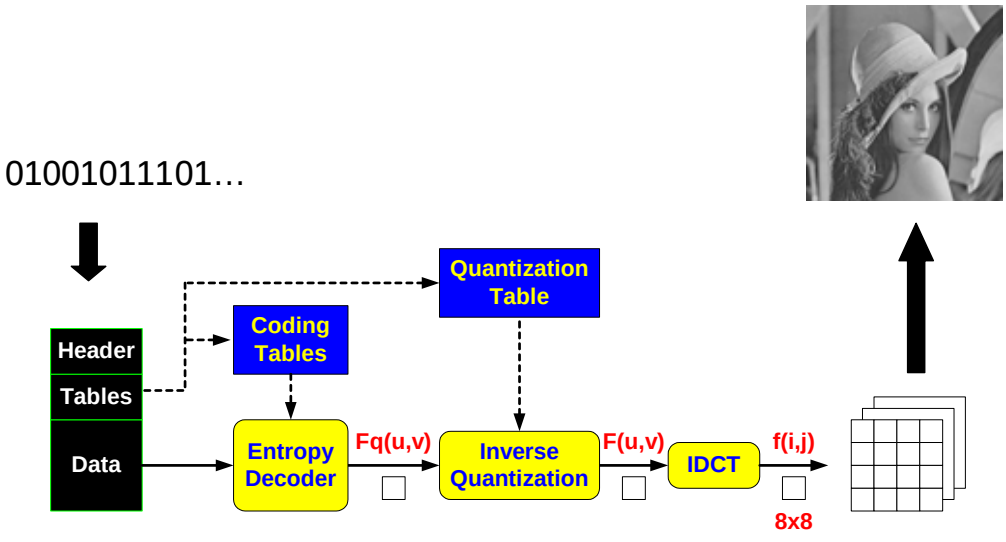
圖一：彩度取樣範例



圖二：影像切割

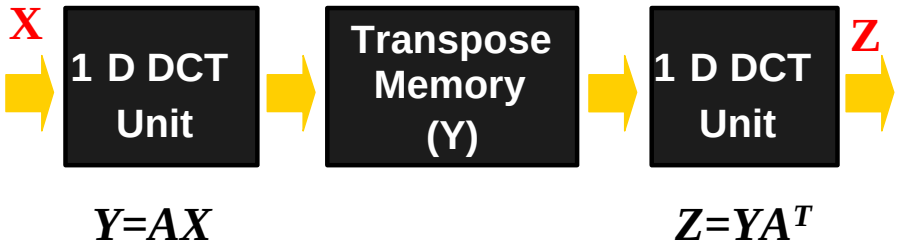


圖三：JPEG 編碼系統方塊圖



圖四：JPEG 解碼系統方塊圖

1.3 二維離散餘弦轉換（2-D Discrete Cosine Transform：2-D DCT）



圖五：利用行列分解法執行二維離散餘弦轉換

離散餘弦轉換就是將空間域數位影像資料轉換成頻率域，此動作又稱為離散餘弦正轉換（Forward Discrete Cosine Transform：FDCT）；反之，將頻率域數位影像資料轉換為空間域則稱為離散餘弦逆轉換（Inverse Discrete Cosine Transform：IDCT）。由於當將數位影像資料作離散餘弦正轉換之時，影像區塊大小為 8×8。因此，在 JPEG 的離散餘弦轉換之運算皆為二維之離散餘弦轉換。JPEG 再處理二維離散餘弦轉換之前會先將所有的像素值減去 128，再利用二維離散餘弦轉換之公式將空間域資料轉換為頻率域資料。同理，逆轉換則是再將資料從頻率域轉換為空間域後加 128。而二維離散餘弦正逆轉換的公式如下：

$$X_{k_1,k_2} = \frac{2}{N} c(k_1) c(k_2) \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} x_{n_1,n_2} \cdot \cos\frac{(2n_1+1)k_1\pi}{2N} \cos\frac{(2n_2+1)k_2\pi}{2N}$$

$$n_1, n_2, k_1, k_2 = 0,1,\cdots,N-1 \qquad \cdots \quad (2)$$

$$where \quad c(0) = \frac{1}{\sqrt{2}} \quad and \quad c(n) = 1 \quad for\ n \neq 0$$

$$x_{n_1,n_2} = \frac{2}{N} \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} c(k_1) c(k_2) X_{k_1,k_2} \cdot \cos\frac{(2n_1+1)k_1\pi}{2N} \cos\frac{(2n_2+1)k_2\pi}{2N}$$

$$n_1, n_2, k_1, k_2 = 0,1,\cdots,N-1 \qquad \cdots \quad (3)$$

$$where \quad c(0) = \frac{1}{\sqrt{2}} \quad and \quad c(n) = 1 \quad for\ n \neq 0$$

二維離散餘弦轉換的超大型積體電路設計一般而言是採用行列分解法（Row-Column Decomposition）完成。其概念具有下列兩種特性：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行運算；架構是以乘累加的架構特性。根據線性代數的觀念可知：$Z=AXA^T$，因此，當 X 為空間域的數值時，

經過上述運算即可成為頻率域數值。故，一個二維離散餘弦轉換運算可拆解如圖五所示。其中，每個一維的離散餘弦轉換皆為八點之一維離散餘弦轉換（8 point 1-D DCT）。而一維離散餘弦轉換公式如下：

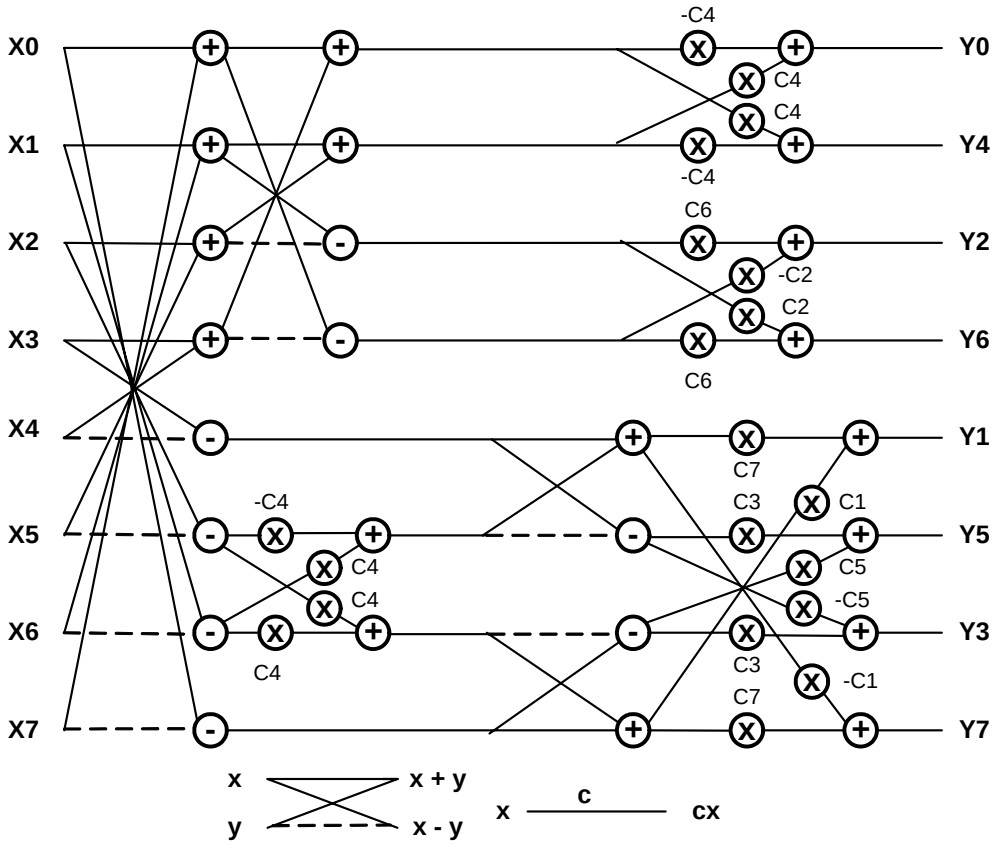
$$F(v) = c(v)\sum_{u=0}^{N-1} f(u) \cdot \cos\frac{(2u+1)v\pi}{2N}$$

$$u, v = 0,1,\cdots,N-1$$

$$f(u) = \sum_{v=0}^{N-1} c(v)F(v) \cdot \cos\frac{(2u+1)v\pi}{2N} \qquad \cdots \quad (4)$$

$$where \quad c(0) = \sqrt{\frac{1}{N}} \quad and \quad c(v) = \sqrt{\frac{2}{N}} \quad for\ v \neq 0$$

此外，為了加速一維離散餘弦轉換的運算速度，許多快速演算法已在文獻中討論。Chen's Algorithm 是一個非常有效率的快速演算法。執行一個 8 point 1-D DCT 的 Chen's Algorithm 只需要用到 16 個乘法器以及 26 個加法器，同時具有規則以及架構簡單的特性。圖六為 Chen's Algorithm 之示意圖。



圖六：Chen's Algorithm[1]

1.4 量化與量化表

量化的目的是希望透過量化的手段，在影像品質能夠接受的情況下，將不重要的訊號于與忽略。因此，一般而言，對於影像處理而言，高頻訊號基本上會透過量化的方法將之去除。JPEG 標準中對於亮度與彩度各有一個量化表，分別如圖七與圖八。仔細觀察此二量化表可以發現，由於高頻訊號明顯比低頻訊號不重要，因此，高頻部分的量化參數遠高於低頻部分。

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	67	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

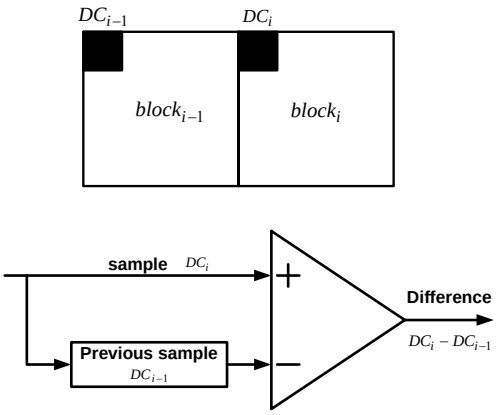
圖七：亮度量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

圖八：彩度量化表

1.5 差分編碼（Differential Pulse Code Modulation：DPCM）

數位影像處理經過離散餘弦轉換以及量化之後，可將信號分為 DC 值與 AC 值兩類。所謂 DC 值即為每個 8x8 區塊中最左上角的值。JPEG 標準中，對於每個 8x8 區塊的 DC 值採用差分編碼。利用差分編碼的目的在於，由於每個 8x8 區塊的 DC 值基本上都很接近，因此，與其儲存各 DC 值，倒不如儲存前後兩兩 DC 值間的差值。如此可進一步達到資料壓縮的目的。而差分編碼的硬體實現也是非常容易完成，如圖九所示。

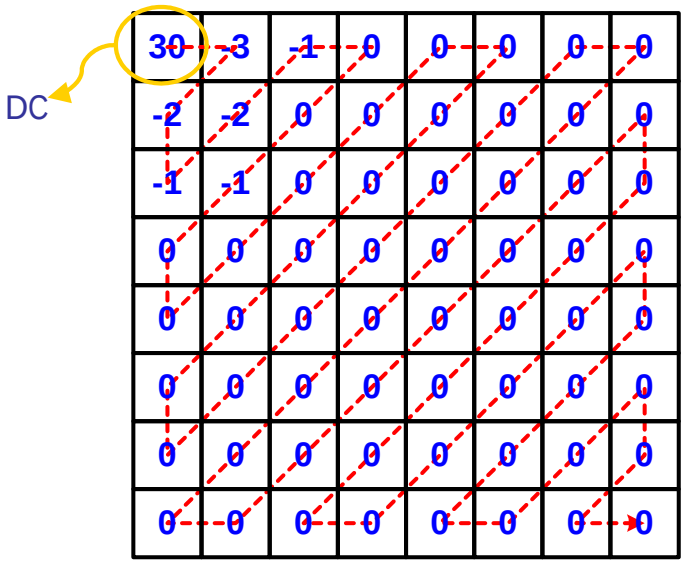


圖九：差分編碼即其硬體實現

1.6 Zig-Zag 掃描與變動長度編碼（Run-Length Coding：RLC）

除了 DC 值利用差分編碼外，其餘的 AC 採用另一個有效的編碼方法，稱為變動長度編碼。在作變動長度編碼之前，由於每個 8x8 區塊具有 63 個 AC 值，何種掃描方式可以達到較佳的壓縮效率，答案是 Zig-Zag 掃描。

Zig-Zag 掃描方式如圖十。為何會有此掃描方式，原因在於此 8x8 區塊是利用二維離散餘弦轉換將訊號從空間域轉頻率域後並經過量化，因此，訊號從左上角至右下角呈現的頻率越來越高。而 Zig-Zag 掃描方式即是將每個 AC 值從低頻訊號逐漸掃描至高頻訊號。況且，吾人已知，越是高頻訊號，值為零的機率越高，因此，透過 Zig-Zag 掃描方式可以讓出現零值的訊號連續出現，讓接下來的變動長度編碼可以達到最高的壓縮率。圖十說明了採用 Zig-Zag 掃描後，變動長度編碼的效率絕對比列掃描（Row Scan）或是行掃描（Column Scan）為佳。



(R,L) => (0,-3)(0,-2)(0,-1)(0,-2)(0,-1)(2,-1)(EOB)

圖十：Zig-Zag 掃描與變動長度編碼

變動長度編碼之格式為（R,L）=（出現 0 的數目,非零之值）。以圖十為範例，DC 值之後接著之 AC 值為-3（非零值），而且-3 之前出現 0 的數目為零，因此可編為（0,-3）；接著-3 後面接著出現-2（非零值），而且-2 之前出現 0 的數目為零，因此可編為（0,-2）；依此類推，接下來依序編出(0,-1) 、（0,-2）、（0,-1）；接下來經過兩個 0 之後，出現的 AC 值為-1，因此可編為（2,-1）；而從此-1 之後至此 8x8 區塊最右下角高頻處之值皆為 0，因此，直接編 EOB 這個碼字，代表，從此-1 之後至區塊最高頻之值皆為 0。

1.7 霍夫曼編碼（Huffman Coding）與霍夫曼表（Huffman Table）

霍夫曼編碼的精神在於符號出現機率越高者，所編的碼字長度越短。透過此種編碼機制，可以讓資料達到壓縮的目的。在 JPEG 編碼中對於 DC 值與 AC 值分別有其相應的霍夫曼表。此外，由於 DC 值不只在亮度有，彩度亦有，因此，亮度與彩度各有其 DC 值的霍夫曼表。故，在編解碼的過程中，透過查表法可以輕易的實現影像訊號的霍夫曼編碼。雖然利用查表法無法讓每個影像達到最佳的壓縮率，然而統一的霍夫曼表可以讓軟硬體的實現變得非常容易實現，且影像品質並不會有顯著的落差。圖十一為亮度與彩度 DC 值之霍夫曼表；圖十二為 AC 值之霍夫曼表。

Size	Length	Codeword
0	2	0
1	3	10
2	3	11
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

圖十一(a)：亮度 DC 值霍夫曼表

Size	Length	Codeword
0	2	0
1	2	1
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

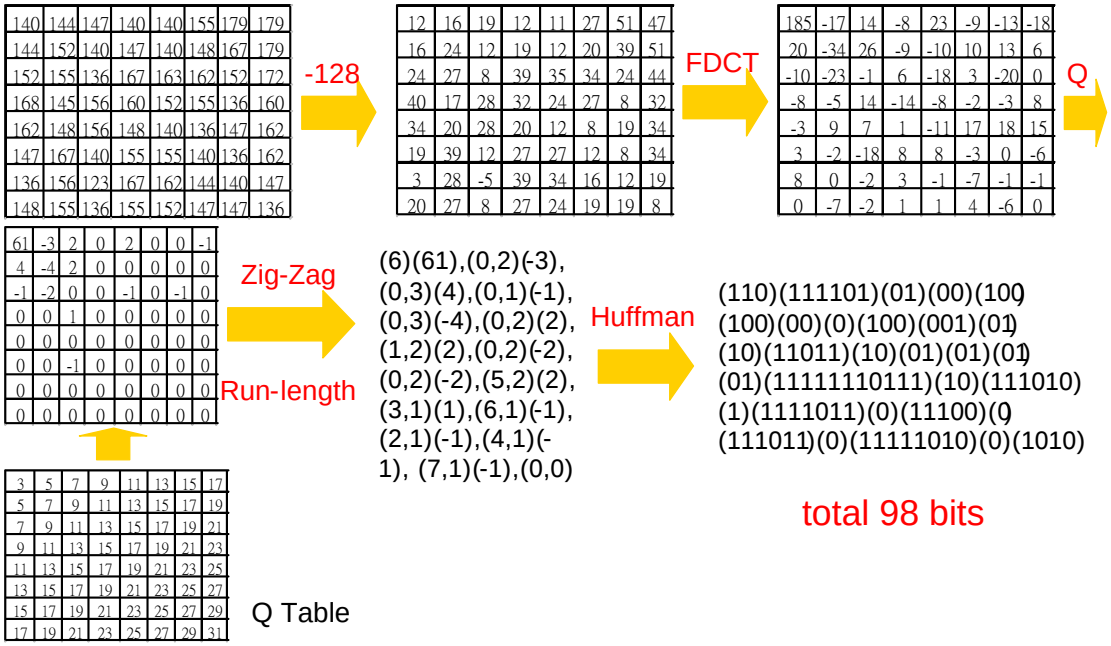
圖十一(b)：彩度 DC 值霍夫曼表

Category	AC Coefficient Range
1	-1,1
2	-3,-2,2,3
3	-7,…,-4,4,…,7
4	-15,…,-8,8,…,15
5	-31,…,-16,16,…,31
6	-63,…,-32,32,…,63
7	-127,…,-64,64,…,127
8	-255,…,-128,128,…,255
9	-511,…,-256,256,…,511
10	-1023,…,-512,512,…,1023
11	-2047,…,-1024,1024,…,2047

圖十二：AC 值霍夫曼表

1.8 序向式以離散餘弦轉換為基礎之模式（Baseline Sequential DCT-based Coding）編碼範例

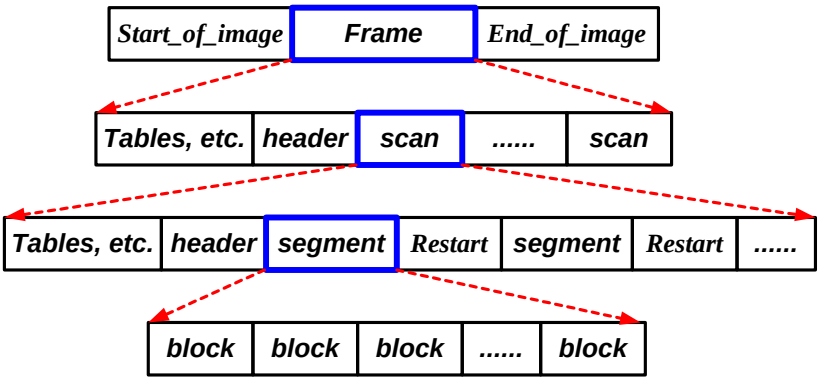
圖十三為是一個序向式以離散餘弦轉換為基礎之模式（Baseline Sequential DCT-based Coding）的編碼範例。透過此範例，可以發現若不經過任何的壓縮方法，將一個 8x8 區塊之值儲存起來需要 8x8x8=512(bits)；然而，透過此壓縮模式後一個 8x8 區塊只需要花 98(bits)即可儲存資訊。而在整個編碼過程中，唯一造成影像會失真的因素只有一個，那就是『量化』的步驟。除此之外，若影像有作取樣的處理，則是另外一個會造成失真的地方。當然上述的考量都不考慮到在系統實現上，浮點數運算轉定點數運算所造成的失真。



圖十三：序向式以離散餘弦轉換為基礎之模式編碼範例

1.9 JPEG 位元流格式

最後，簡單介紹一下 JPEG 之位元流格式，如圖十四所示。



圖十四：JPEG 之位元流格式

2. 引導實驗

2.1 檔案結構

```
Final_project --------- sw.bat — 純軟體執行之批次檔
                  |
                  |---- sw -------- bmp.cpp — 讀取*.bmp 檔
                            |---- jpeg.cpp — JPEG 之區塊編解碼引擎
                            |---- jpeg.h — jpeg.cpp 之宣告
                            |---- main.cpp — JPEG Codec 程式主體
                            |---- marker.h — JPEG 圖檔之標籤
                            |---- picture.cpp — 存取靜態影像
                            |---- stream.cpp — 讀取 bitstream
                            |---- stream.h — stream.cpp 之宣告
                            |---- type.h — 基本型別的宣告
```

2.2 設計方法

本實驗將把一個靜態影像壓縮以及解壓縮成標準 JPEG 格式而設計一個壓縮、解壓縮的程式。在此吾人將思索如何建構程式，接著將探討當程式放入 ARM Integrator/AP 會遭遇到何種問題。最後將簡單評估一下程式的效能，決定是否需要把繁重的資料處理工作交由硬體來完成，藉由此簡單的 JPEG 壓縮、解壓縮例子，吾人可以看到執行靜態影像的壓縮與解壓縮此項工作如何分別交給軟體及硬體以完成軟硬體共同設計（Hardware/Software Co-design）。

首先先思考此 JPEG 系統應該提供哪些功能？經過思索可知，至少需要允許使用者輸入一個圖檔，接著系統將圖檔影像壓縮成編準的 JPEG 格式輸出至使用者的磁碟機裡；或是使用者交給系統一張 JPEG 格式的影像，而系統必須把此影像解壓縮再轉換成其他影像格式輸出。所以，吾人首先要知道 JPEG 影像在壓縮及解壓縮需要做哪些工作，以及標準的 JPEG 格式如何定義資料的擺放。關於這個問題的答案，詳細的資料可參閱〝JPEG: Still Image Data Compression Standard [2]〞此本參考書，裡面詳細記載著影像壓縮與解壓縮時所需要的處理步驟。但

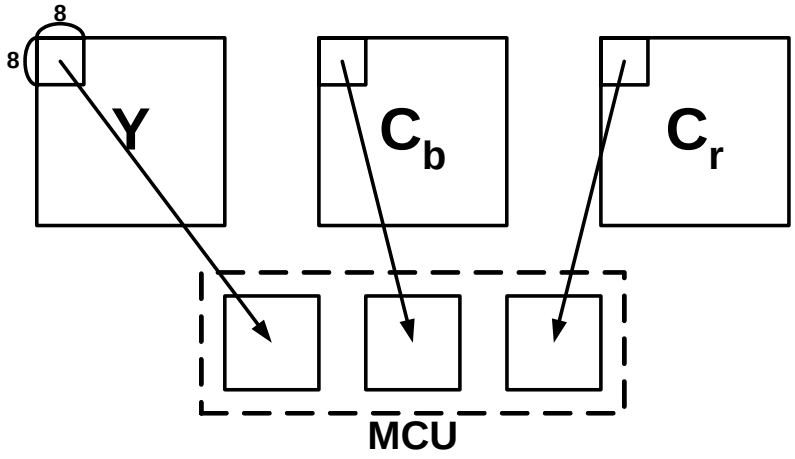
是在此實驗不需要將整本書所規定的要求完全實現出來，基本上，本實驗只實現了序向式以離散餘弦轉換為基礎之模式 JPEG 壓縮、解壓縮系統。既然本實驗只讓系統執行序向式 JPEG 編解碼，因此，先了解一下序向式 JPEG 編解碼需要哪些要求，如下表一：

序向式以離散餘弦轉換為基礎之模式 JPEG 編解碼系統
● 以離散餘弦轉換為基礎之影像處理 ● 原始影像（像素取樣值之資料型態大小皆為八位元寬） ● 熵編碼採用霍夫曼編碼 ● 影像最多包含 4 個元素（Component） ● 支援插補與非插補的掃描模式（Interleave and non-interleave scans）

表一：序向式 JPEG 編解碼系統之需求與特性

首先在此先解釋，表一所提到的元素（Component）是指影像的顏色。例如，一般吾人常使用的顏色表示法是採用紅（Red）、綠（Green）、藍（Blue）三原色，所以一張影像可以分成紅、綠、藍三張畫面，而每張畫面就是代表一個元素；只不過 JPEG 裡用的顏色表示法是用 Y、C_b、C_r。因此，吾人在此先定義了元素 1 代表影像的 Y 畫面，元素 2 代表影像的 C_b 畫面，依此類推。元素 4 在此並不會使用，因為，一般 JPEG 圖檔最多就包含有 3 個元素。接下來，吾人還需要知道 JPEG 處理影像的最小單位為一塊 8x8 大小的區塊，簡單的來說，就有如切豆腐一般把一張影像切成數個小方塊，每一個小方塊長寬各為 8（個像素），如之前之圖二所示。

當影像為灰階影像，也就是只有一個元素（即元素 1：Y）時，影像處理的順序理所當然的就是由左至右、由上而下依序處理每個 8x8 區塊，而此種情況就是表一裡所謂的非插補掃描（Non-interleave order）；那當有 3 個元素時要怎麼處理呢？，概念是一樣的，只不過先處理元素 1 的第一塊 8x8 區塊再處理元素 2 的第一塊 8x8 區塊，最後再處理元素 3 的第一塊 8x8 區塊，而掃描 8x8 區塊的順序仍然是由左至右、由上而下依序處理，此種方法就是差補掃描（Interleave order），而這樣一組的插補掃描（Y1C_b1C_r1），吾人稱做為一組 MCU（Minimum Coded Unit），如圖十三所示。



圖十三：MCU 示意圖

因此對於非插補掃描（Non-interleave order）而言，MCU 就是最小的一組資料處理單位。在此還需說明一些關於 MCU 的情況，MCU 並不非就是 3 個元素各取一塊 8x8 區塊（4:4:4 取樣），其實還可以取 4 塊 Y 區塊，C_b、C_r 各取一塊 8x8 區塊（4:1:1 取樣）。每一組 MCU 所包含各元素的 8x8 區塊個數是會依照程式內建的參數來決定，只是本程式省略了其他各類的取樣組合，且一般來說目前的 JPEG 圖檔大部分都是 4:4:4 取樣（Y、C_b、C_r 各一個區塊）或 4:1:1 取樣（4 個 Y 區塊；C_b、C_r 各一個區塊）此二組合。至於 DCT-based 處理是指壓縮和解壓縮的流程，如圖三、圖四所示。也就是說每一塊 8x8 區塊依序會經過 DCT 轉換，再經過量化（Quantize），再對每個符號做熵編碼（Entropy coding）；解壓縮時則做相反的動作。上述提到的熵編碼在 JPEG 標準裡定義了兩種編碼方式霍夫曼編碼以及算數編碼（Arithmetic coding），本實驗之實作只做霍夫曼編碼的部分。JPEG 標準中並非對每個符號直接做霍夫曼編碼；而是針對符號之長度做霍夫曼編碼，在此利用圖十四以及圖十五說明。假設目前要編碼的符號為 3，根據圖十四可知符號 3 的長度（SSSS）為 2；再根據圖十五，可知長度（SSSS）為 2（即圖十五中 Category＝2）時其霍夫曼碼為 011。因此編符號 3 時，是先編其長度之霍夫曼碼為 011；至於此數字符號需要取多少位元呢？就是看長度（SSSS）為多少。因此，符號 3 編碼後為 01111。至於當數字符號為負值該如何解決？解決之道就是將數值之二進位表示直接減 1 即可。因此，－3、－2 編碼後分別為 01100、01101。

SSSS	Value
0	0
1	-1,1
2	-3,-2,2,3
3	-7,…,-4,4,…,7
4	-15,…,-8,8,…,15
5	-31,…,-16,16,…,31
⋮	⋮

圖十四：符號長度對照表

Category	Code Word
0	00
1	010
2	011
3	100
4	101
……	……

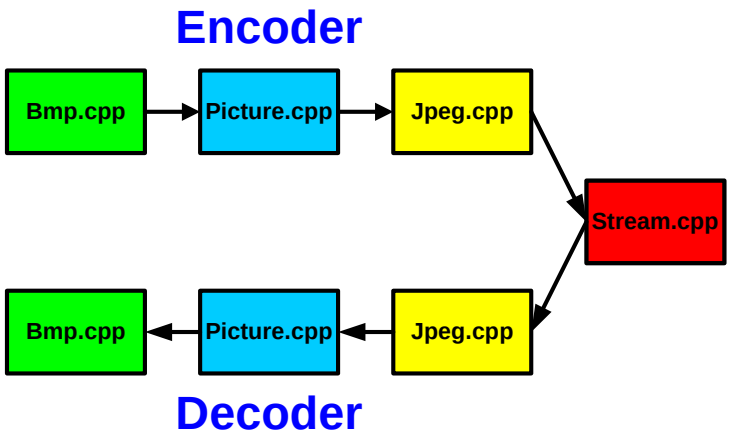
圖十五：符號長度霍夫曼表

2.3 程式碼設計流程與分析

本實驗之 JPEG 編解碼程式基本上由 bmp.cpp、picture.cpp、jpeg.cpp 以及 stream.cpp 此四大部分構成，如圖十六。每個程式碼主要肩負的任務如下列所述：

- bmp.cpp：*.bmp 影像檔的讀取以及寫入。
- picture.cpp：此處的程式碼只是讓使用者看到一塊 8x8 區塊，換句話說，只是把相對應的像素值拷貝到某一個 8x8 大小的矩陣內部，當然，這裡必須讓使用者決定他要取哪一塊 8x8 區塊。
- jpeg.cpp：這裡開始實作二維離散餘弦轉換、量化、可變長度編碼；處理的單位為 8x8 區塊，當然它是由 picture.cpp 所提供的。
- stream.cpp：提供使用者讀跟寫位元流（bit-stream）。也就是說使用者只要呼叫某幾個函式就可以寫入或讀取一個位元（bit）到檔案，當然還允許使用者決定當不足一個位元組（byte）時，用 0 或 1 補齊。此外，當使用者寫 16 個

1 時，可以控制是否需連續加入 16 個 0，此部份的動作只是 JPEG 標準所要求的。



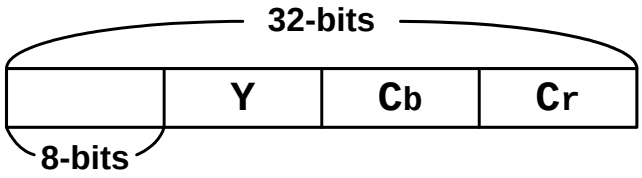
圖十六：JPEG 編解碼程式架構圖

接下來將解釋每個程式以及其內部的函式以及變數之意義。

- type.h —此處只是重新定義幾個基本型別，Byte、Word、DWord 以及 picture.cpp 的宣告，其他的都沒有使用。
- bmp.cpp — *.bmp 影像檔的讀取以及寫入。在讀取或寫入影像值時，同時考慮了色彩座標的轉換。即當讀取時會同時做 RGB 的色彩座標轉換成 JPEG 影像處理中採用的 YC_bC_r 色彩座標；以及寫入時會同時做 YC_bC_r 的色彩座標轉換成 RGB 色彩座標。
- stream.cpp
  - ✓ *Buffer*、*BufferLength*：每次讀取一個位元組（byte）就放到 *Buffer* 變數裡，當使用者存取 1 bit 時就從 *Buffer* 變數裡拿取；至於 *BufferLength* 則是用來記錄 *Buffer* 變數裡現在有幾個位元（bit）。此二個變數是不讓後端使用者直接存取的。
  - ✓ *ByteStuffing*：當 *ByteStuffing* 設為 `true` 時，寫入 0xFF 至檔案時會自動加入 0x00，讀到 0xFF 時會自動捨棄下一個位元組（byte）。
  - ✓ *Name*[32]：只是用來記錄檔名。
  - ✓ **`bool Open(const char* name, const char* attr)`**：和 **`fopen()`** 函式功能相同。

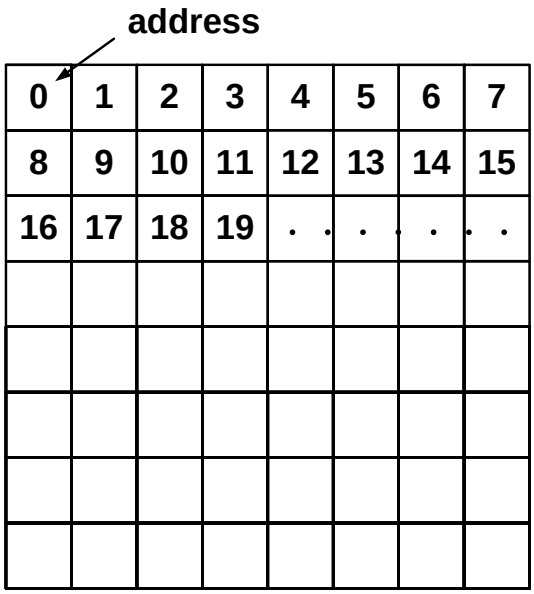
- ✓ **bool Close()**：和 **fclose()**函式功能相同。
- ✓ **bool IsEndOfStream()**：和 **feof()**函式相同。
- ✓ **bool PutStream(DWord code, Byte length)**：用來存取 *Buffer* 變數，使用者藉由參數 *code* 決定寫入位元流（bit-stream）。*length* 則表示 code 取多少位元，失敗時傳回 `false`，例如：**PutStream(0x19,5)** 就是寫入 11001 共 5 個位元（bit）。
- ✓ **bool GetStream(DWord* code, Byte length)**：用來讀取 *Buffer* 變數，*length* 用來表示要讀取多少位元，讀取的位元會存放在參數 *code*，失敗時傳回 `false`，例如：**GetStream(&strm, 5)**就是讀取 5 個位元，存到 *strm* 變數裡。**void Align(bool flag)**：當 *flag* 為 `true` 時，若 *BufferLength* 不為 8，則 *Buffer* 就用 1 補齊至 8 位元；反之，當 *flag* 為 `false` 時，就用 0 把 *Buffer* 補滿。

- ➢ picture.cpp
  - ✓ *Pixel*：一個二維指標，指到的記憶體位置就是存放影像的像素值。每個像素值為 32 位元（4 位元組）寬，且每個欄位意義如下圖所示。



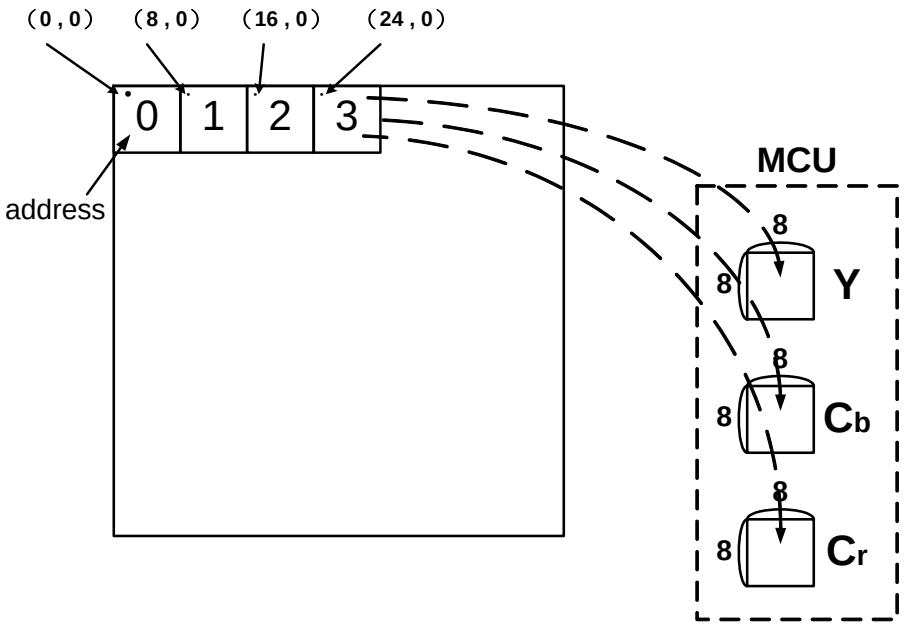
圖十七：*Pixel* 變數之各欄位意義

  - ✓ *Width*、*Height*：紀錄 *Pixel* 矩陣的寬與長。
  - ✓ *MCU*：8x8 大小的矩陣，用來存放每個 8x8 區塊的像素值，整個結構如圖十八所示。其中 *Type* 變數紀錄此 8x8 區塊是 Y 元素或是 C_b 元素。



圖十八：*MCU* 結構圖

- ✓ **bool IsEmpty()**：*Pixel* 變數如果為 0（NULL），就表示矩陣未初始化，**IsEmpty()**函式會傳回 `false`，反之 *Pixel* 不為 0（NULL）傳回 `true`。
- ✓ **void Create(Word w, Word h)**：配置矩陣。被分配到的記憶體位置會存放在 *Pixel* 變數裡。
- ✓ **void Release()**：刪除矩陣，再把 *Pixel* 變數設為 0（NULL）。
- ✓ **bool _get_min_code_unit(int address)**：這個函式把 *Pixel* 的像素值拷貝到 *MCU* 變數裡，也就是讓使用者決定要影像裡哪一個 8×8 區塊，其中參數 *address* 表示 8×8 區塊的編號，編號從 0 開始，由影像的左邊數到右邊，上到下，可參考圖十九；當 *address* 為–1 時，8×8 區塊的位置會改由 *Block_Address* 變數決定，每呼叫一次 *Block_Address* 就會加一，當超過最後一個 8×8 區塊的編號時，會自動設為 0。



圖十九：**bool _get_min_code_unit(int address)**示意圖

- ✓ **bool _put_min_code_unit(int address)**：同上，功能改成將 *MCU* 變數之值寫入至 *Pixel*。

- jpeg.cpp
  - ✓ **HuffmanDC*DC*[3]**：記錄霍夫曼編碼。此變數有兩個欄位 *CodeWord*、*CodeLength*，*CodeWord* 用來記錄霍夫曼碼字，*CodeLength* 用來記錄 *CodeWord* 有多少位元，這兩個變數都是矩陣，其矩陣的索引值就是 SSSS。舉例來說：當要為–15 編碼時，它的 SSSS 值就是 4，霍夫曼表建立後，***DC.CodeWord*[4]**就記錄 0x0005，***DC.CodeLength*[4]**會存放 3，所以編碼時就會編成 101，之後再編 0000，所以–15 這個數值會編碼成 1010000。
  - ✓ **HuffmanAC *AC*[3]**：同上，但 *CodeWord* 和 *CodeLength* 在此是二維矩陣，矩陣的第一個索引值代表 RRRR，代表係數前面有幾個 0，第二個索引值就是 SSSS。
  - ✓ **void ChenFDCT(int (*block)[8])**：利用 Chen's Algorithm 此快速演算法實現離散正餘弦轉換演算法之實作。此段函式是直接引用他人的程式碼；而 Chen's Algorithm 詳細的做法請參閱 【1】。
  - ✓ **void ChenIDCT(int (*block)[8])**：利用 Chen's Algorithm 此快速演算法實現離散逆餘弦轉換演算法之實作。此段函式亦直接引用他人

程式碼，詳細的做法請參閱 【1】。

- ✓ **void _initial_dc_table(Byte identifier, Byte (*dc_length)[17], Byte (*dc_value)[12])**：初始化 DC 表。簡單的來說就是建立 *DC（jpeg.h 34 行）*變數的數值，參數 *identifier* 指明初始化哪個 DC 表。
- ✓ **void _initial_ac_table(Byte identifier, Byte (*ac_length)[17], Byte (*ac_value)[162])**：做法同上。
- ✓ **void _encoding_block_value(Byte identifier, int (*block)[8])**：VLC 編碼。
- ✓ **void _decoding_block_value(Byte identifier, int (*block)[8])**：VLC 解碼。

2.4 實驗步驟

本實驗之目的為實現一軟硬體共同設計（HW/SW Co-design）之 JPEG 編解碼器（JPEG Codec）。然而，實現軟硬體共同設計（HW/SW Co-design）之前有一個重要的課題即是軟硬體分割（HW/SW Partition）。因此，首先必須先分析利用 ARM 平台做全軟體模擬時，JPEG 編解碼系統中哪些運算是最複雜，最值得將此部份運算利用硬體執行，使用軟硬體共同設計達到效能大幅增加的好處。故，在此將把實驗步驟切割為兩部分：第一部份為純軟體模擬，並利用實驗二中 Profiling 的方法分析此 JPEG 編解碼器。透過 Profiling 的分析，得知二維離散餘弦轉換（2-D DCT/IDCT）運算為運算量最高的函式，再將二維離散餘弦轉換（2-D DCT/IDCT）運算以硬體的形式將電路掛在 AMBA 上作為一個 Slave 裝置，完成軟硬體共同設計之 JPEG 編解碼器。

《JPEG 編解碼器軟體模擬以及效能分析》

在執行軟體模擬之前，先了解在此部份實驗所需要的一些先前準備工作。本實驗已經提供了一個 `sw.bat` 讓使用者可以直接執行此批次檔即可產生 JPEG 編解碼器的 ARM 可執行檔—`sw.axf`。利用此批次檔可以省略在命令提示字元或是利用 CodeWarrior 編譯以及連結所有 C++檔案的步驟。此外，本實驗提供了四個測試用的 bmp 影像檔—`test1.bmp`、`test2.bmp`、`test3.bmp` 以及

test4.bmp。至於 sw 此資料夾，於之前的介紹已知道此資料夾內放的即是所有C++的檔案。有了這些基本概念後，接下來即是整個模擬以及效能分析。

- 當欲壓縮之影像過大將會造成無法將整張影像壓縮完成，請試著修改原始碼解決此問題。

參考文件及網頁

[1] W. Chen, CH Smith, and S. Fralic, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-25, pp. 1004-1009, Sept 1977.

[2] JPEG: Still Image Data Compression Standard by William B. Pennebaker and Joan L. Mitchell, Kluwer Academic Publishers, ISBN: 0442012721