EE521800

Application Acceleration with High-Level Synthesis

Lab#C

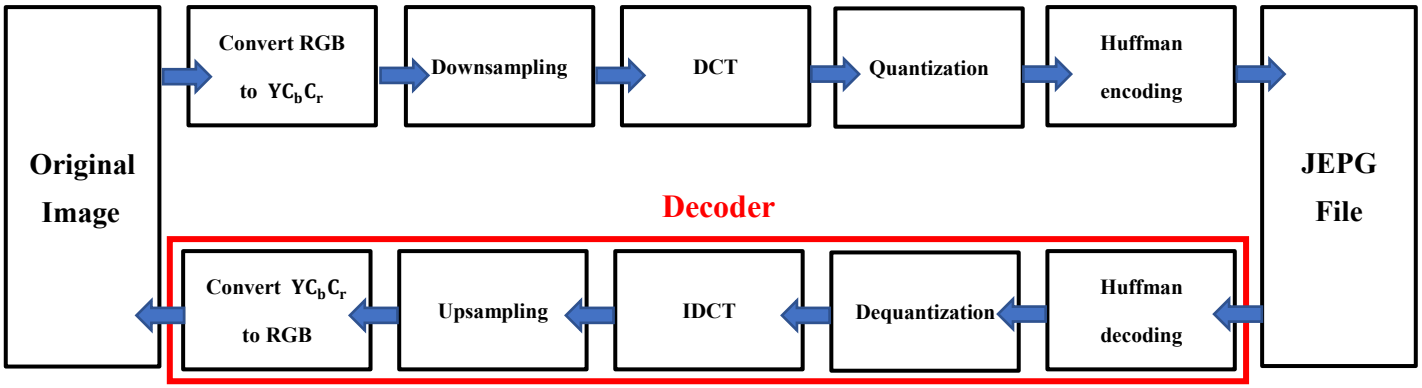
CODEC

Student : 江威霖、吳秉豐、陳思熙

1. Algorithm

➢ Block Diagram

下圖為 JPEG Encode 與 Decode 的流程，而此次 Lab 我們所製作的為 Decoder 的部分。



➢ JIFI Parser

JIFI Parser 的用途是，根據各個區段碼，讀取 JPEG 檔案中的各項資訊，並將其分割，下圖為一 JPEG 檔案中的資訊與其對應的圖片，之後會以此圖做為例子來解釋各個區塊的分割方式。

範例圖片

```
FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 48
00 48 00 00 FF DB 00 43 00 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 FF DB 00 43 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 FF C0
00 11 08 00 02 00 06 03 01 22 00 02 11 01 03 11
01 FF C4 00 15 00 01 01 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 09 FF C4 00 19 10 01 00 02
03 00 00 00 00 00 00 00 00 00 00 00 00 00 06 08
38 88 B6 FF C4 00 15 01 01 01 00 00 00 00 00 00
00 00 00 00 00 00 00 00 04 0A FF C4 00 1C 11 00
01 03 05 00 00 00 00 00 00 00 00 00 00 00 00 08
00 07 B8 09 38 39 76 78 FF DA 00 0C 03 01 00 02
11 03 11 00 3F 00 86 F7 E7 1D A9 16 CA 77 30 D0
14 F7 41 DC 5A 8E FB 31 19 26 5D C4 2A F4 5C 81
7B DB 06 84 A0 75 17 FF D9 00 00 00 00 00 00 00
```

JPEG 檔案內容

a. SOI (Start Of Image)

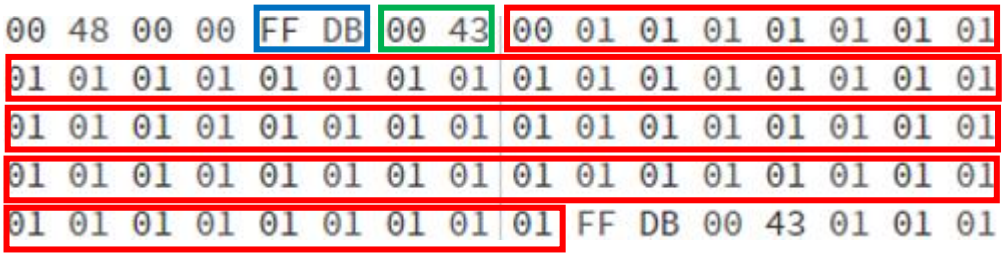
✧ Mark : FF D8

✧ Information : 宣告資料的起點

b. DQT (Define Quantization Table)

✧ Mark : FF DB

✧ Information : 存放 Quantization Table 與其對應編號



✧ 解析方式：FF DB 後面接續的 00 43 表示這個 block 總共有 67 個 byte 的資料，在接下來的 00 為此 Quantization Table 的編號而此圖共有兩組 Quantization Table，再接下去的 64 筆 01 依序為 8×8 的 Quantization Table 內所存的值，所以做後解出 Quantization Table 由下所示：

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

✧ 不同程度的 Quantization 效果比較

以下圖為例：

未處理圖片

Luminance (brightness) table								Chrominance (colour) table							
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

中度量化後圖片

Luminance (brightness) table								Chrominance (colour) table							
26	19	22	22	29	38	78	115	27	29	38	75	158	158	158	158
18	19	21	27	35	56	102	147	29	34	42	106	158	158	158	158
16	22	26	35	59	88	125	152	38	42	90	158	158	158	158	158
26	30	38	46	90	102	139	157	75	106	158	158	158	158	158	158
38	42	64	82	109	130	165	179	158	158	158	158	158	158	158	158
64	93	91	139	174	166	194	160	158	158	158	158	158	158	158	158
82	96	110	128	165	181	192	165	158	158	158	158	158	158	158	158
98	88	90	99	123	147	162	158	158	158	158	158	158	158	158	158

高度量化後圖片

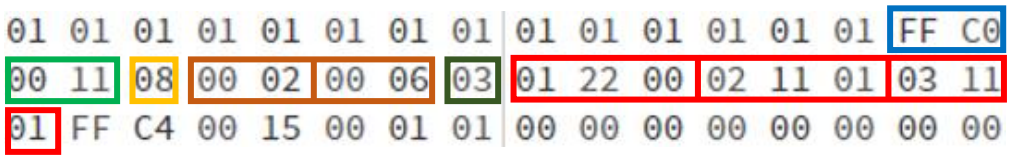
Luminance (brightness) table								Chrominance (colour) table							
112	84	98	98	126	168	343	504	119	126	168	329	693	693	693	693
77	84	91	119	154	245	448	644	126	147	182	462	693	693	693	693
70	98	112	154	259	385	546	665	168	182	392	693	693	693	693	693
112	133	168	203	392	448	609	686	329	462	693	693	693	693	693	693
168	182	280	357	476	567	721	784	693	693	693	693	693	693	693	693
280	406	399	609	763	728	847	700	693	693	693	693	693	693	693	693
357	420	483	560	721	791	840	721	693	693	693	693	693	693	693	693
427	385	392	434	539	644	707	693	693	693	693	693	693	693	693	693

比較上面各個結果可以發現，Quantization 的量越高，所得到的圖片 Quality 會越低，圖片的視覺呈現出塊狀模糊，但優點是其資料儲存空間較小。

c. SOF (Start Of Frame)

✧ Mark : FF C0

✧ Information : 存放圖片的寬高與各個分量（Y、C_b、C_r）的採樣率，與其對應的 Huffman Table 與 Quantization Table 的編號。



✧ 解析方式 : FF C0 後面接續的 00 11 表示這個 block 總共有 17 個 byte 的資料，接下來的 08 為精度、00 02 與 00 06 分別表示圖片的高度與寬度，接下來 03 為總共有幾個 component（Y、C_b、C_r），最後紅色框的部分再表示各個 component 的採樣率與所要用的 Quantization Table 的編號，解析出的資訊如下所示：

資料精度：8

圖片大小：6 × 2

Component 數量：3

Y ：採樣率 → 水平 2、垂直 2，使用 0 號 Quantization Table

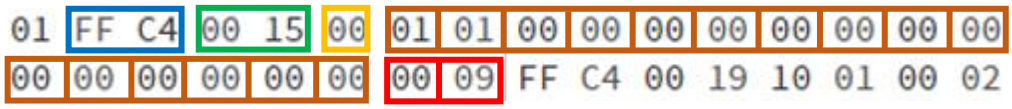
C_b ：採樣率 → 水平 1、垂直 1，使用 1 號 Quantization Table

C_r ：採樣率 → 水平 1、垂直 1，使用 1 號 Quantization Table

d. DHT (Define Huffman Table)

✧ Mark : FF C4

✧ Information : 存放 Huffman Table 與其對應編號



✧ 解析方式：FF C0 後面接續的 00 15 表示這個 block 總共有 21 個 byte 的資料，接下來的 00 表示此表為 DC 0 號表（Ex.若為 01 → DC 1 號表、10 → AC 0 號表），接下來 16 個 byte 分別表示各個碼字長度各有幾筆資料，最後紅色框的部分依序填入各個碼字。

✧ 霍夫曼樹建立方式

- 第一個碼字必定為 0

如果第一個碼字位數為 1，則碼字為 0。

如果第一個碼字位數為 2，則碼字為 00。

依此類推

- 從第二個碼字開始

若與上個碼字位數相同，則目前碼字為前一個碼字加 1。
若位數比上個碼字位數還大，則目前碼字為前一個碼字加 1 後再右移直到滿足碼字長度為止。

則根據解析出的資訊與上述的規則重建後，所的到的 Huffman Table 如下所示：

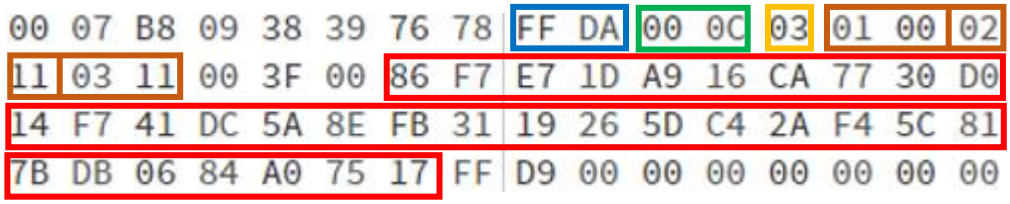
編號：DC 0 號表

序號	碼字長度	碼字	數值
1	1	0	0x00
2	2	1	0x09

e. SOS (Start Of Scan)

✧ Mark : FF DA

✧ Information : 存放各個 component 對應的 Huffman Table 編號，與真正要被 decode 的 data



✧ 解析方式：FF C0 後面接續的 00 0C 表示這個 block 總共有 12 個 byte 的資料，接下來的 03 指有三個 component，再來咖啡色的部分是在記錄每個分量對應到的 Huffman Table，最後紅色的部分是紀錄最後需要被解碼的資料，以下為紅色以外的資訊解析出的結果：

Component 數量：3

Y ：Huffman Table → DC 0 號表、AC 0 號表

C_b ：Huffman Table → DC 1 號表、AC 1 號表

C_r ：Huffman Table → DC 1 號表、AC 1 號表

✧ 讀取壓縮圖項數據方式，以下列例子來講解

圖片的數據皆是以 bit 為單位來儲存資料的，且數據都是在 encode 時經過 FDCT 得到的結果，所以各個 component 應該皆由 1 個直流分量與 63 個交流分量所組成。

各個顏色分量單元皆使用了 RLE 與 Huffman 編碼來壓縮處理，讀取單個顏色分量單元的步驟如下：

- 讀取直流系數：
  由資料的開頭開始讀取，直到讀到的 code 與 DC Huffman Table 的碼字一致，然後查表找到對應的數值即表示接下來要讀取的位元數，在經由下表解碼後即為直流係數

實際數值	位數	碼字
-1, 1	1	0, 1
-3, -2, 2, 3	2	00, 01, 10, 11
-7, -6, -5, -4,……,5	3	000, 001,……,111
依此類推至位數 ≤ 11		

正數就是照一般二進位計算，負數就是對正數的碼字取反而已，有點像是一補數的計算。

- 讀取交流係數：
  再接著讀取，直到讀到的 code 與 AC Huffman Table 的碼字一致，然後查表找到對應的數值，此數值的 MSB 4 bits 代表接下來的數值連續有幾個 0、LSB 4 bits 代表接下來要讀取的位元，將讀出的資料精上述轉換表進行轉換後，即可得到要接續在 0 後的數值。

- 當發生以下兩種狀況時，代表交流係數以讀取完畢
  持續讀取交流係數直到讀滿 63 個交流邊碼，或者是讀到到 0x00 時剩下的交流系入全為 0

- 以下列數據做為例子

某個顏色分量單元數據如下：

D3 5E 6E 4D 35 F5 8A 若以二進制表示可以得到 → 110 1001101 01 1 11001 101 11001 001 101 00 11010 1 1111010 11 00 01010

DC Huffman Table

序號	碼字長度	碼字	數值
1	2	00	0x00
2	2	01	0x01
3	2	10	0x02
4	3	110	0x07
5	4	1110	0x1e

AC Huffman Table

序號	碼字長度	碼字	數值
1	2	00	0x00
2	2	01	0x01
3	3	100	0x11
4	3	101	0x02
5	5	11000	0x21
6	5	11001	0x03
7	5	11010	0x31
8	5	11011	0x41
9	5	11100	0x12
10	6	111010	0x51
11	7	1110110	0x61
12	7	1110111	0x71
13	7	1111000	0x81
14	7	1111001	0x91
15	7	1111010	0x22

讀到的第一個 Huffman 編碼為 110 對應到往後讀取 7bits → 1001101 轉換後數值為 77（直流分量），接下來接著讀取並對照 AC Huffman Table 得到的第一個編碼為 01 其碼值為 0x01 所以他的前面沒有 0 並往後讀取 1bits → 1 轉換後數值為 1 依此類推最後讀到霍夫曼編碼 00 其對應的數值為 0x00 滿足結束條件，而剩餘的 01010 則為下一個顏色分量單元的資訊，經由上述的規則進行解析後即可得到下列結果：

77	**1**	**5**	**-6**	**-3**	**0**	**0**	**0**
1	**0**	**0**	**3**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

✧ DC 係數的差分編碼

相鄰的兩個顏色分量得直流變量是以差分來編碼的，經過剛剛解碼出來的直流變量為 D_{diff} 所以實際的直流變量為：

$$D_n = D_{n-1} + D_{diff}$$

✧ MCU (Minimum Coded Unit)

JPEG 資料會被分割成許多 MCU 單元，而一個 MCU 的寬高與各個 Component 的採樣率有關，如下所示：

$\mathrm{MCU_{Width}} = 8\mathrm{px} \times \max(\text{horizaontal sampling fator of } \mathrm{YC_bC_r})$
$\mathrm{MCU_{Heigh}} = 8\mathrm{px} \times \max(\text{vertical sampling fator of } \mathrm{YC_bC_r})$

以 $\mathrm{YC_bC_r}$ 採樣率為 $4 \times 1 \times 1$ 為例：

$\mathrm{MCU_{Width}} = 8\mathrm{px} \times 2 = 16\mathrm{px}$
$\mathrm{MCU_{Heigh}} = 8\mathrm{px} \times 2 = 16\mathrm{px}$

而讀取 MCU 的順序為 Y0、Y1、Y2、Y3、$\mathrm{C_b}$、$\mathrm{C_r}$每一塊對應為大小為 8×8 的 matrix。

另外在 JPEG 的規定中，若圖片的尺寸無法與 MCU 的尺寸整除，則需補上一些 pixel 來填滿一個 MCU，以下圖來舉例：

此圖為尺寸為 616×516 且採樣率為 $4 \times 1 \times 1$ 則

$\frac{616}{16} = 38.5$ $\frac{516}{16} = 32.25$，所以在補足 pixel 後總共會需要

$39(\text{horization}) \times 33(\text{vertical}) = 1287$ 個 MCU 最後 Decode 出來的圖片如下所示

由上圖可以發現 Decode 出來的結果在邊緣會有一些雜點，這就是因為在圖片壓縮時為了補足 MCU 所導致的結果。

- JPEG Decoder

a. Dequantization

不同的顏色分量會使用不同的量化表，各個對應的量化表紀錄在 SOF 中，Dequantization 的方式即對 8×8 的顏色分量單元，逐一乘上量化表上對應位置內的值即可。

b. Zig – Zag decode & IDCT

下圖為圖片經過 IDCT 轉換後的 log scale 結果：

Original Grayscale Image

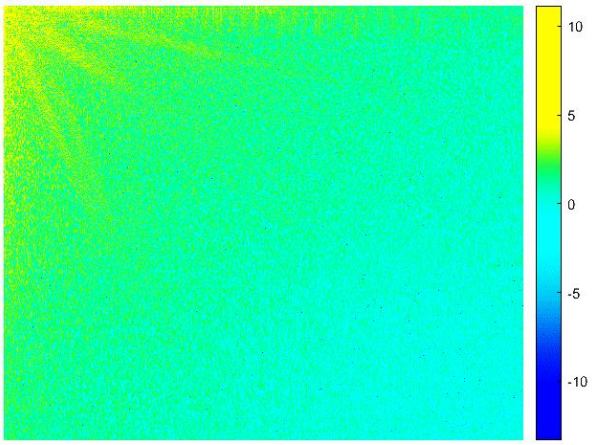


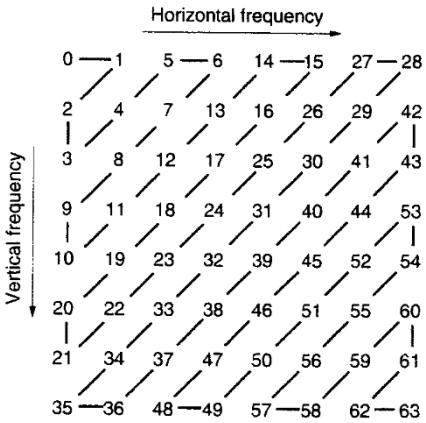
Image data after 2D-DCT with lag scale

可以發現經過 IDCT 轉換後，低頻的資訊會集中在左下角，而高頻的值會集中在右下角，且高頻值通常較低且有許多的 0，而 JPEG 檔案在傳送時連續的 0 可以只用 1 個 byte 的資料進行傳送，因此我們可以配合 zig-zag 轉換讓傳送的資料量大幅縮短。

DCT : $F(k) = \lambda(k) \sum_{i=1}^{N} cos(\frac{\pi}{2N}(2n-1)(k-1)), k = 1, 2, ..., N$

IDCT : $f(n) = \sum_{k=1}^{N} \lambda(n) cos(\frac{\pi}{2N}(2n-1)(k-1)), n = 1, 2, ..., N$

Normalization factor: $\lambda(k) = \begin{cases} \frac{1}{\sqrt{N}} & \text{when } k = 1, \\ \frac{2}{\sqrt{N}} & \text{when } 2 \le k \le N. \end{cases}$



Zig - Zag 轉換圖

2. Development process

因為 Gitgub 中介紹的執流程與我們所使用得裝置不同，所以需要做下列更正使得程式可以正常執行

- Step 1 : Having some problems running the makefile
  > vim Makefile
  Go to keyword “DEVICE” and change the default, device u200 → u50

- Step 2 : Change to the right shell environment and then run the Makefile
  > bash (to change the shell)
  > make run TARGET=sw_emu/hw_emu/hw

- Step 3 : 若出現下圖 ERROR

```
------------ Test for decode image.jpg  -------------
WARNING: /users/course/2022S/HLS17000000/g110064521/HLS_C/Vitis_Libraries/codec/L2
/demos/jpegDec/images/t2.jpg will be opened for binary read.
83060 entries read from /users/course/2022S/HLS17000000/g110064521/HLS_C/Vitis_Lib
raries/codec/L2/demos/jpegDec/images/t2.jpg
XRT build version: 2.12.427
Build hash: 2719b6027e185000fc49783171631db03fc0ef79
Build date: 2021-10-09 05:06:49
Git branch: 2021.2
PID: 58874
UID: 1700000016
[Sat Apr 30 04:43:08 2022 GMT]
HOST: ic21
EXE: /users/course/2022S/HLS17000000/g110064521/HLS_C/Vitis_Libraries/codec/L2/dem
os/jpegDec/build_dir.sw_emu.xilinx_u50_gen3x16_xdma_201920_3/host.exe
[XRT] ERROR: locale::facet::_S_create_c_locale name not valid
[XRT] ERROR: locale::facet::_S_create_c_locale name not valid
[XRT] ERROR: locale::facet::_S_create_c_locale name not valid
Error: Failed to find Xilinx platform
make: *** [run] Error 1
[g110064521@ic21 jpegDec]$
```

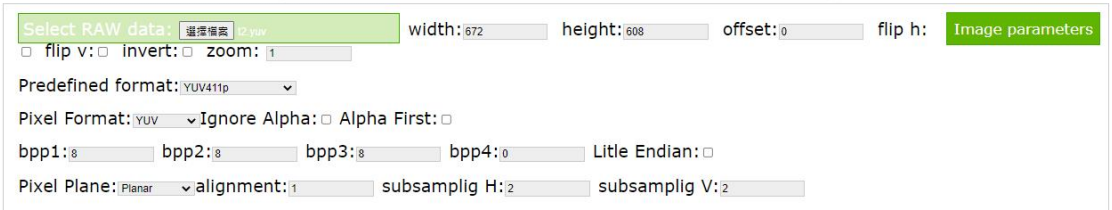
在 terminal 輸入以下指令後再執行 Makefile

> export LC_ALL=C

- Step 4： 在執行 host program 後，在 images 資料夾中得到 yuv 跟 yuv.h 檔案

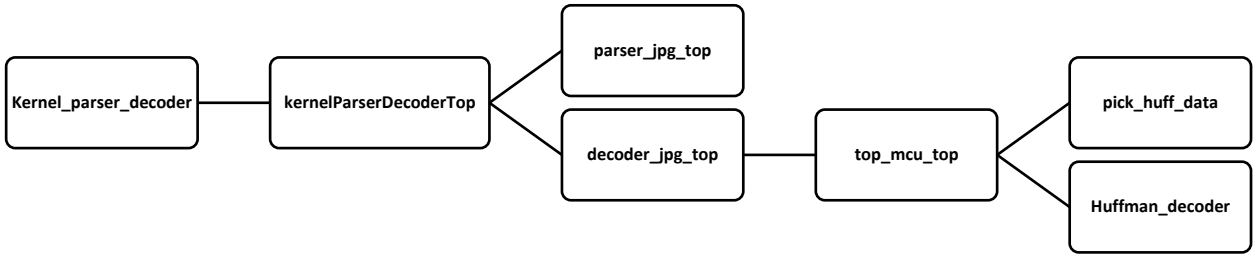
```
nthucad:~/LAB_C_codec/codec/L2/demos/jpegDec/images> ll
total 2892
-rw-r--r-- 1 g110061560 HLS17000000  51193  四   6 16:21 t0.jpg
-rw-r--r-- 1 g110061560 HLS17000000 494208  四  21 22:54 t0.raw
-rw-r--r-- 1 g110061560 HLS17000000 494208  四  21 22:54 t0.yuv
-rw-r--r-- 1 g110061560 HLS17000000    423  四  21 22:54 t0.yuv.h
-rw-r--r-- 1 g110061560 HLS17000000  46190  四   6 16:21 t1.jpg
-rw-r--r-- 1 g110061560 HLS17000000 269568  四  21 22:53 t1.raw
-rw-r--r-- 1 g110061560 HLS17000000 269568  四  21 22:53 t1.yuv
-rw-r--r-- 1 g110061560 HLS17000000    423  四  21 22:53 t1.yuv.h
-rw-r--r-- 1 g110061560 HLS17000000  83060  四   6 16:21 t2.jpg
-rw-r--r-- 1 g110061560 HLS17000000 612864  四  21 22:53 t2.raw
-rw-r--r-- 1 g110061560 HLS17000000 612864  四  21 22:53 t2.yuv
-rw-r--r-- 1 g110061560 HLS17000000    423  四  21 22:53 t2.yuv.h
```

- Step 5： 使用 YUV viwer 開啟檔案



3. HLS Implementation

➢ Function Structure



a. Paresr_jpg_top

用來解析 JPEG 資料中的各個 mark（Huffman Table、Quantization Table、Data、Information），並將其傳送回 kernelParserDecoderTop 中再傳送給 decoder_jpg_top 去做後續的計算。

b. decoder_jpg_top

用來計算 IDCT 與 Dequantization 等運算並將結果組成 YUV 檔案中的資訊。

➢ Example of Optimization

a. Data dependency (Inter LOOP RAW)

```
GEN_NEW_TBL_LOOP:
    for (huff_len = 1; huff_len <= 16; ++huff_len) {
#pragma HLS PIPELINE

        huff_cnt[huff_len - 1] = segment[r](c * 8 + 7, c * 8);
        uint16_t tbl3_code = dc_huff_start_code[cmp_huff][huff_len - 1];
        uint16_t tbl4_code = ac_huff_start_code[cmp_huff][huff_len - 1];
        if (huff_len <= 15) {
            if (ac) {

                ac_huff_start_code[cmp_huff][huff_len] = (tbl4_code + huff_cnt[huff_len - 1]) << 1;
                ap_uint<16> tmp = (tbl4_code + huff_cnt[huff_len - 1]) << 1; // huff_len+1 bit
                // origin huff_len 10bit~15, now count the 11bit start code(so, huff_len-3+1),  index from
                // 10~15 cut to 0~5
                if (huff_len > DHT1) {
                    ac_start_code[cmp_huff][huff_len - DHT1 - 1] = tmp(huff_len - 2, 0);
                }
            } else {
                dc_huff_start_code[cmp_huff][huff_len] = (tbl3_code + huff_cnt[huff_len - 1]) << 1;
                ap_uint<16> tmp = (tbl3_code + huff_cnt[huff_len - 1]) << 1;
                // origin huff_len 10bit~15, cut 2 bit
                if (huff_len > DHT1 && (huff_len < 12)) {
                    dc_start_code[cmp_huff][huff_len - DHT1 - 1] = tmp(huff_len - 2, 0);
                }
            }
        }
```

GEN_NEW_TBL_LOOP

II	: 2
LATENCY	: 165
BRAM (curr + child = total)	: - + - = -
URAM (curr + child = total)	: - + - = -
DSP (curr + child = total)	: - + - = -

parser_jpg_top_Pipeline_GEN_NEW_TBL_LOOP　　II Violation

在原本的 source cdoe 中上述的 LOOP 中的 ac_huff_start_code 與 dc_huff_start_code 會有 inter loop RAW 的問題，導致合成後的 II 無法為 1 所以我們將 code 改寫為下列形式使用 data forwarding 的方法讓 II 可以為 1

```
            uint16_t temp_ac = 0;
            uint16_t temp_dc = 0;
            GEN_NEW_TBL_LOOP:
                for (huff_len = 1; huff_len <= 16; ++huff_len) {
#pragma HLS PIPELINE
                    huff_cnt[huff_len - 1] = segment[r](c * 8 + 7, c * 8);
                    uint16_t tbl3_code = temp_dc;
                    uint16_t tbl4_code = temp_ac;

                    if (huff_len <= 15) {
                        if (ac) {
                            temp_ac = (tbl4_code + huff_cnt[huff_len - 1]) << 1
                            ac_huff_start_code[cmp_huff][huff_len] = temp_ac;

                            //ac_huff_start_code[cmp_huff][huff_len] = (tbl4_code + huff_cnt[huff_len - 1]) << 1;
                            ap_uint<16> tmp = (tbl4_code + huff_cnt[huff_len - 1]) << 1; // huff_len+1 bit
                            // origin huff_len 10bit~15, now count the 11bit start code(so, huff_len-3+1),  index from
                            // 10~15 cut to 0~5
                            if (huff_len > DHT1) {
                                ac_start_code[cmp_huff][huff_len - DHT1 - 1] = tmp(huff_len - 2, 0);
                            }
                        } else {
                            temp_dc = (tbl3_code + huff_cnt[huff_len - 1]) << 1;
                            dc_huff_start_code[cmp_huff][huff_len] = temp_dc;


                            //dc_huff_start_code[cmp_huff][huff_len] = (tbl3_code + huff_cnt[huff_len - 1]) << 1;
                            ap_uint<16> tmp = (tbl3_code + huff_cnt[huff_len - 1]) << 1;
                            // origin huff_len 10bit~15, cut 2 bit
                            if (huff_len > DHT1 && (huff_len < 12)) {
                                dc_start_code[cmp_huff][huff_len - DHT1 - 1] = tmp(huff_len - 2, 0);
                            }
                        }
                    }
```

GEN_NEW_TBL_LOOP

II	: 1
LATENCY	: 148
BRAM (curr + child = total)	: - + - = -
URAM (curr + child = total)	: - + - = -
DSP (curr + child = total)	: - + - = -

b. Loop Merge

```
62  #pragma HLS ARRAY_PARTITION variable = pout->q_tables complete dim = 2
63  #pragma HLS ARRAY_PARTITION variable = pout->idct_q_table_x complete dim = 3
64  #pragma HLS ARRAY_PARTITION variable = pout->idct_q_table_y complete dim = 3
65      unsigned short RESIDUAL_NOISE_FLOOR = 7;
66      for (int idx_cmp = 0; idx_cmp < pout->axi_num_cmp_mcu; idx_cmp++) {
67          uint8_t c = pout->axi_map_row2cmp[idx_cmp];
68
69          for (int i = 0; i < 64; i++) {
70  #pragma HLS pipeline
71              pout->idct_q_table_x[c][i >> 3][i & 7] =
72                  hls_icos_base_8192_scaled[(i & 7) << 3] * pout->q_tables[c][i & 7][i >> 3];
73              pout->idct_q_table_y[c][i >> 3][i & 7] =
74                  hls_icos_base_8192_scaled[(i & 7) << 3] * pout->q_tables[c][i >> 3][i & 7];
75              // pout->idct_q_table_t[c][i>>3][i&7] = hls_icos_idct_linear_8192_scaled[i] *
                pout->q_tables[c][0][i&7];
76              //}
77
78              // for (int coord = 0; coord < 64; ++coord) {
79              freqmax_[c][i] = (freqmax[i] + pout->q_tables[c][i >> 3][i & 7] - 1) / pout-
                >q_tables[c][i >> 3][i & 7];
80              // uint8_t max_len = uint16bit_length(freqmax_[c][i]);
81              uint8_t max_len = 16 - freqmax_[c][i].countLeadingZeros();
82              // bitlen_freqmax_[c][i] = max_len;
83              if (max_len > (int)RESIDUAL_NOISE_FLOOR) {
84                  pout->min_nois_thld_x[c][i] = pout->min_nois_thld_y[c][i] = max_len -
                    RESIDUAL_NOISE_FLOOR;
85              } else {
86                  pout->min_nois_thld_x[c][i] = pout->min_nois_thld_y[c][i] = 0;
87              }
88
89          } // end for
90      }
91  }
```

上述 Code 將 IDCT 與 Dequantization 兩個步驟合二為一，在進行 IDCT 的同時將 Dequantization 的係數一同考慮，將原本需要兩個 for 迴圈的計算 merge 在一起，節省了進入與退出迴圈所需要時間。

- Performance and Utilization
  a. Synthesis result

Timing Estimate

Target	Estimated	Uncertainty
5.70 ns	4.161 ns	1.54 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	V	Dis	Slack	Latency(cycles)	Latency(ns)	Ite	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel_parser_decoder				0.00	-	-	-	-	-	no	7	11	9824	18792	0
kernel_parser_decoder_Pipeline_1				-	5	28.500	-	5	-	no	0	0	4	44	0
parser_jpg_top				0.00	-	-	-	-	-	no	0	10	6248	10329	0
decoder_jpg_top				0.00	-	-	-	-1	-	dataflow	1	1	2434	6525	0

可以發現 FF 與 LUT 的數量較高，是因為需要儲存較大量的 Table 與等待解碼的 Data，所以需要較多的 resource。
而 Estimated 的 throughput 為 240.327MHz

b. Co-Sim result

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kernel_parser_decoder				137689	137689	137689
kernel_parser_decoder_Pipeline_1				3	3	3
parser_jpg_top				58055	58055	58055
decoder_jpg_top				79626	79626	79626

c. Software and Hardware result compression

- Software emulation result

```
------------------------------------------------------
INFO: Data transfer from host to device: 495 us
------------------------------------------------------
INFO: Data transfer from device to host: 177730 us
------------------------------------------------------
INFO: kernel 0: execution time 1507841 usec
INFO: kernel 1: execution time 1515837 usec
INFO: kernel 2: execution time 1531445 usec
INFO: kernel 3: execution time 1499947 usec
INFO: kernel 4: execution time 1511723 usec
INFO: kernel 5: execution time 1517877 usec
INFO: kernel 6: execution time 1515949 usec
INFO: kernel 7: execution time 1556665 usec
INFO: kernel 8: execution time 1478098 usec
INFO: kernel 9: execution time 1509851 usec
INFO: Average kernel execution per run: 1514523 us
------------------------------------------------------
INFO: Average E2E per run: 16835934 us
------------------------------------------------------
```

- Hardware result

```
----------------------------------------------------
INFO: Data transfer from host to device: 227 us
----------------------------------------------------
INFO: Data transfer from device to host: 11217 us
----------------------------------------------------
INFO: kernel 0: execution time 878 usec
INFO: kernel 1: execution time 794 usec
INFO: kernel 2: execution time 814 usec
INFO: kernel 3: execution time 759 usec
INFO: kernel 4: execution time 800 usec
INFO: kernel 5: execution time 743 usec
INFO: kernel 6: execution time 745 usec
INFO: kernel 7: execution time 759 usec
INFO: kernel 8: execution time 809 usec
INFO: kernel 9: execution time 753 usec
INFO: Average kernel execution per run: 785 us
----------------------------------------------------
INFO: Average E2E per run: 566537 us
----------------------------------------------------
```

由上面兩個結果比較後可以發現，Hardware 的執行速度比起 Software 快上非常多。

4. Conclusion

- JPEG algorithm provides a method with high compression ratio, and has good performance in restoring images.
- There are a lot of repetitive operations caused by the JPEG algorithm, so the decoding performance can be improved with the paralleled computing hardware.
- HLS can be useful with optimizing loops, pragma PIPELINE help us to speed up development process.

5. Reference

- https://github.com/Xilinx/Vitis_Libraries
- https://zh.m.wikipedia.org/zh-tw/JPEG
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7077268
- https://github.com/MROS/jpeg_tutorial
- http://twins.ee.nctu.edu.tw/courses/soclab_04/lab_hw_pdf/proj1_jpeg_introduction.pdf?fbclid=IwAR2Gu2K17iUQ6HBCnA_EciUFAAWZQMPjbZoJ0tlEY2SWJoNpdf3k7wprY4M

6. Appendix

以下圖片舉例做 JPEG 解碼矩陣變化的過程

Picture size : 32×16

Sample factor : 4:1:1 = 1:1:1

MCU size : 8×8

Number of MCU : $4(\text{horizational}) \times 16(\text{vertical})$

Y Component : DC Huffman Table 0, AC Huffman Table 0

C_b Component : DC Huffman Table 1, AC Huffman Table 1

C_r Component : DC Huffman Table 1, AC Huffman Table 1

Y DC Huffman Table

00 0x09
010 0x05
011 0x06
100 0x07
101 0x08
110 0x0A

C_b、C_r DC Huffman Table

0 0x00
10 0x06
110 0x03
1110 0x01
11110 0x02

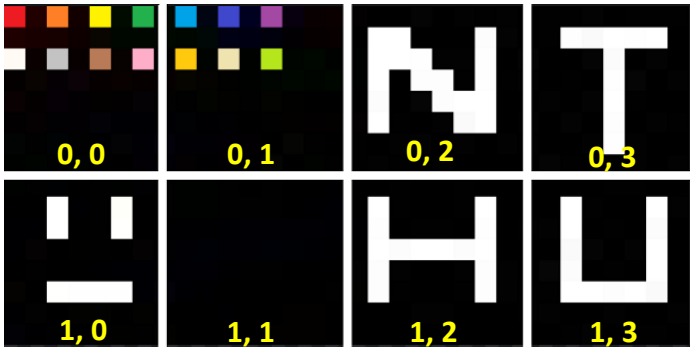
Y AC Huffman Table

00 0x06
010 0x04
011 0x05
100 0x07
1010 0x01
1011 0x08
11000 0x02
11001 0x03
11010 0x15
11011 0x17
11100 0x18
111010 0x09
111011 0x11
111100 0x16
1111010 0x00
1111011 0x14
1111100 0x19
11111010 0x21
11111011 0x22
111111000 0x24
111111001 0x27
111111010 0x28
111111011 0x41
111111100 0x51
111111101 0x61
111111110 0xB1

C_b、C_r AC Huffman Table

00 0x04
010 0x01
011 0x02
100 0x03
101 0x05
1100 0x06
11010 0x00
11011 0x11
11100 0x21
111010 0x31
111011 0x61
1111000 0x12
1111001 0x51
1111010 0x71
11110110 0x07
11110111 0x13
11111000 0x14
11111001 0x15
11111010 0x22
11111011 0x41
11111100 0x81
11111101 0xF0
111111100 0xC1
111111101 0xE1
111111110 0xF1

此圖的分割方式如下，總共會有 8 塊 MCU



MCU (0, 0)未處理的資料如下所示：(Y(左)、C_b (中)、C_r (右))

$$\begin{bmatrix} -834 & 37 & 187 & 46 & 31 & -3 & 35 & -16 \\ -8 & -49 & -25 & -33 & -39 & 35 & 2 & 47 \\ 3 & 2 & -53 & -24 & 82 & 161 & 10 & -42 \\ -18 & 5 & 38 & 3 & 152 & 0 & -23 & 6 \\ -13 & -14 & 39 & 132 & 33 & 10 & 15 & 4 \\ -61 & -8 & 164 & 52 & -13 & -43 & 1 & 37 \\ 18 & 32 & 0 & 14 & -12 & -23 & -8 & -1 \\ 60 & -1 & 56 & 5 & 76 & 144 & 9 & 115 \end{bmatrix} \quad \begin{bmatrix} -39 & -7 & -56 & -41 & -3 & 21 & -11 & 26 \\ -7 & -29 & -28 & -5 & 22 & -10 & -11 & 15 \\ -13 & -5 & 15 & -3 & -23 & -19 & -2 & 13 \\ -2 & -7 & 16 & 2 & -37 & 1 & 9 & -4 \\ -4 & 16 & -1 & -12 & 0 & 15 & -6 & -4 \\ 5 & -2 & -50 & -41 & -2 & 7 & -6 & -7 \\ 11 & -6 & -9 & 8 & -1 & -33 & -28 & 1 \\ 10 & -7 & 7 & 2 & -26 & -23 & 2 & -14 \end{bmatrix} \quad \begin{bmatrix} 40 & 38 & 42 & 15 & 56 & 1 & 6 & 2 \\ 53 & 6 & 7 & 52 & 6 & 5 & 14 & -1 \\ 20 & -1 & 8 & 36 & 13 & 17 & 16 & 5 \\ -5 & 20 & -3 & 14 & 18 & 25 & -5 & 18 \\ -2 & 0 & 4 & 12 & -1 & -3 & 3 & 14 \\ -6 & 35 & 20 & 10 & 39 & -4 & 11 & 8 \\ -2 & 7 & 6 & -3 & 33 & 0 & 3 & 15 \\ 0 & 3 & 1 & 1 & 8 & 16 & -3 & 13 \end{bmatrix}$$

因為 Quantization Table 的值皆為 1，所以反量化後的結果相同，之後經過 Zig-Zag 排列後結果如下：(Y(左)、C_b (中)、C_r (右))

$$\begin{bmatrix} -834 & 37 & -3 & 35 & 2 & 47 & 3 & 152 \\ 187 & 31 & -16 & 35 & 3 & 38 & 0 & 164 \\ 46 & -8 & -39 & 2 & 5 & -23 & -8 & 52 \\ -49 & -33 & -53 & -18 & 6 & -61 & -13 & -23 \\ -25 & -24 & -42 & -13 & 4 & -43 & -12 & -8 \\ 82 & 10 & -14 & 15 & 1 & 14 & -1 & 76 \\ 161 & 39 & 10 & 37 & 0 & 60 & 5 & 144 \\ 132 & 33 & 18 & 32 & -1 & 56 & 9 & 115 \end{bmatrix} \quad \begin{bmatrix} -39 & -7 & 21 & -11 & -11 & 15 & 2 & -37 \\ -56 & -3 & 26 & -10 & -13 & 16 & 1 & -50 \\ -41 & -7 & 22 & -5 & -7 & 9 & -2 & -41 \\ -29 & -5 & 15 & -2 & -4 & 5 & -2 & -33 \\ -28 & -3 & 13 & -4 & -4 & 7 & -1 & -28 \\ -23 & -2 & 16 & -6 & -6 & 8 & 1 & -26 \\ -19 & -1 & 15 & -7 & -9 & 10 & 2 & -23 \\ -12 & 0 & 11 & -6 & -7 & 7 & 2 & -14 \end{bmatrix} \quad \begin{bmatrix} 40 & 38 & 1 & 6 & 14 & -1 & 14 & 18 \\ 42 & 56 & 2 & 5 & 20 & -3 & 25 & 20 \\ 15 & 53 & 6 & -1 & 20 & -5 & 35 & 10 \\ 6 & 52 & 8 & -5 & 18 & -6 & 39 & 0 \\ 7 & 36 & 5 & -2 & 14 & -4 & 33 & 3 \\ 13 & 16 & 0 & 3 & 11 & -3 & 15 & 8 \\ 17 & 4 & -3 & 8 & 6 & 0 & 1 & 16 \\ 12 & -1 & -2 & 7 & 3 & 1 & -3 & 13 \end{bmatrix}$$

最後在經過 IDCT 後就可以得到最終的結果：

$$\begin{bmatrix} 91 & 10 & 155 & 5 & 218 & 7 & 123 & 5 \\ 9 & 7 & 7 & 4 & 6 & 4 & 4 & 5 \\ 251 & 3 & 195 & 2 & 137 & 3 & 201 & 1 \\ 1 & 5 & 2 & 5 & 2 & 3 & 2 & 2 \\ 4 & 1 & 2 & 1 & 1 & 1 & 3 & 2 \\ 1 & 2 & 1 & 1 & 3 & 0 & 0 & 0 \\ 3 & 1 & 2 & 1 & 0 & 3 & 1 & 3 \\ 1 & 2 & 1 & 2 & 1 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 96 & 122 & 62 & 125 & 4 & 124 & 102 & 125 \\ 123 & 124 & 123 & 126 & 125 & 126 & 125 & 125 \\ 124 & 126 & 128 & 127 & 100 & 126 & 128 & 127 \\ 127 & 126 & 129 & 129 & 127 & 126 & 127 & 127 \\ 127 & 127 & 130 & 129 & 127 & 128 & 129 & 128 \\ 128 & 128 & 129 & 129 & 130 & 130 & 130 & 130 \\ 128 & 128 & 129 & 128 & 129 & 129 & 130 & 132 \\ 127 & 127 & 128 & 128 & 128 & 128 & 130 & 130 \end{bmatrix} \quad \begin{bmatrix} 232 & 144 & 199 & 137 & 154 & 123 & 65 & 125 \\ 143 & 141 & 139 & 135 & 130 & 125 & 125 & 125 \\ 131 & 133 & 128 & 132 & 162 & 131 & 166 & 129 \\ 130 & 131 & 130 & 132 & 131 & 132 & 131 & 131 \\ 132 & 130 & 131 & 129 & 129 & 129 & 129 & 128 \\ 129 & 130 & 128 & 129 & 128 & 128 & 128 & 128 \\ 127 & 128 & 126 & 128 & 128 & 128 & 128 & 129 \\ 127 & 126 & 128 & 126 & 128 & 128 & 129 & 128 \end{bmatrix}$$

而 MCU (0, 2) 與 (0, 3) Y Component 使用同樣的方式 decode 可以得到如下的矩陣，此兩個 MCU 對應到圖片上的 N 與 T 的位置可以發現白色部分亮度明顯較高

Y Component MCU (0, 2)

$$\begin{bmatrix} 2 & 0 & 0 & 4 & 0 & 2 & 1 & 2 \\ 0 & 255 & 255 & 0 & 0 & 0 & 252 & 3 \\ 1 & 255 & 255 & 254 & 2 & 2 & 254 & 0 \\ 1 & 255 & 0 & 253 & 253 & 1 & 254 & 1 \\ 0 & 252 & 0 & 5 & 253 & 255 & 255 & -1 \\ 2 & 255 & 1 & -1 & 2 & 250 & 256 & 0 \\ 1 & 0 & 2 & 3 & 0 & 2 & 0 & 2 \\ 0 & 2 & 1 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Y Component MCU (0, 3)

$$\begin{bmatrix} 0 & 1 & 1 & 3 & -1 & 1 & 1 & 1 \\ 0 & 1 & 254 & 250 & 255 & 254 & 255 & 0 \\ 4 & 0 & 0 & 4 & 251 & 0 & 3 & 0 \\ 2 & 0 & 1 & -1 & 254 & 5 & 0 & 2 \\ 0 & 2 & -1 & 3 & 255 & 1 & 0 & 0 \\ 2 & 0 & 3 & 0 & 253 & 0 & 2 & 1 \\ 1 & 0 & 0 & 2 & 254 & 2 & -1 & 0 \\ 0 & 4 & 1 & 0 & 2 & 2 & 0 & 2 \end{bmatrix}$$