Hardware-Software Codesign of JPEG Compression using SystemC

EE8205: Embedded Computer Systems

1 Objectives

The purpose of this lab is to provide the experience of modeling the hardware-software codesign process. Students will be using SystemC environment to model and implement part of JPEG-based image compression and decompression techniques.

2 Introduction to JPEG Encoding and Decoding

The process of JPEG-based encoding and decoding of images vary according to color depth (8, 24 or 32 bits). However, the basic ideology for all color depths is same. The bitmap image stores raw pixel-by-pixel color values. In addition, 54 bytes are stored at the start of file as header information that includes image width and height, image file size, image color depth, etc. These 54 bytes must be taken into account whenever working with the bitmap images. Following the 54-byte header, the bitmap image holds the color values of each pixel that varies for different color depths. For an 8-bit image, this is simply one byte (8-bits) per pixel and for a 32-bit image; they are 4 bytes per pixel.

For 8-bit pixels, the pre-processing stage divides image data into 8x8 blocks that are shifted from unsigned integers with range $[0, 2^8 - 1]$ into signed integers with a range of $[-2^7, 2^7 - 1]$ and then individually compressed at the 8x8 block level. The compression process for each block goes through the following processes in addition to pre-processing.

- Discrete Cosine Transform (DCT)
- Quantization
- Zigzag
- Entropy Encoding (commonly Huffman)

Decompression is an inverse process that performs the individual inverse of all the above processes.

(i) DCT and Inverse DCT:

Discrete Cosine Transform is the heart of JPEG compression and also time consuming process. The following equations are the idealized mathematical definitions of 8x8 DCT and 8x8 IDCT respectively:

$$F(u,v) = \tfrac{1}{4}C(u)\,C(v)\,[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * cos((2x+1)u\pi)/16 * cos((2y+1)v\pi/16] \quad \text{---- (1)}$$

$$f(x,y) = \tfrac{1}{4}[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)\,C(v)F(u,v) * cos((2x+1)u\pi)/16 * cos((2y+1)v\pi/16] \quad \text{---- (2)}$$

where: $C(u)$, $C(v) = 1/\sqrt{2}$ for $u,v = 0$
$C(u)$, $C(v) = 1$ otherwise
$F(u,v)$ is the Discrete Cosine transformed 8x8 block
$f(x,y)$ is the Inverse Discrete Cosine transformed 8x8 block

(ii) Quantization

The 8x8 block of transformed values is then divided by a distinct quantization value for each transformed block entry.

$F_{quantized}(x,y) = F(x,y) / Quantization\_Table(x,y)$

The quantization table used for this purpose is given below:

$$\begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

Inverse of quantization is the multiplication of 8x8 block by the quantization table as following.

$F_{unquantized}(x, y) = F(x, y) * Quantization\_Table(x, y)$

(iii) Zigzag

Zigzag step takes the quantized 8x8 block and orders them in a 'zig-zag' sequence, resulting in a 1-D array of 64 entries, which is shown in Figure 1. This process helps entropy coding by placing low-frequency coefficients (usually larger values) before the high-frequency coefficients (usually close to zero). One can also ignore any continuous zero values at the end and insert a unique control byte (EOB) at the end of each 8x8 block encoding. The zigzag sequence is explained in Figure 1.

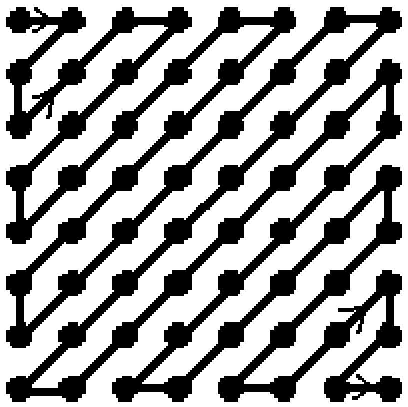

Figure 1: Zigzag sequence

(iv) Entropy Encoding (Huffman)

After zigzag, the 64 or less values are then encoded for further compression.

3 Lab Directions

JPEG compression and decompression algorithm will be developed as described above by using the SystemC knowledge acquired in Lab #1. Assume that the target system consists of one CPU and some additional dedicated hardware. All the software processes will be implemented on the CPU. The main processes involved in JPEG are DCT/IDCT, quantization, zigzag and entropy encoding as given below.

- Pre-processing
- The DCT process is extremely slow in terms of CPU execution, as it consists of four nested loops for each 8x8 block in the image. Implement the DCT in hardware to speed up this process.
- Implement the quantization and zigzag processes in software (CPU). The output of this process should be written to an output file along with the 54 bytes of header information. Students can insert the same 54-byte header employed for bitmap files and its details are provided in the Appendix. Although you will not be using the JPEG file header in this lab, its details are provided for information and completeness.

The source code for JPEG compression excluding Huffman coding is provided at course directory /ee8205/project/jpeg/. Study the compression code, compile it and execute it by providing the input bitmap file and an output name for the compressed file.
You are required to implement the inverse of compression, i.e. reading from the compressed file, performing the inverse zigzag and quantization in software (CPU) and IDCT via the dedicated hardware. The resulting values from the IDCT are to be written to a bitmap file, which can be matched with the input bitmap file. In this way, you will be able to verify your implementation as intended. You are not required to implement Entropy/Huffman coding or encoding.

When making the hardware modules sensitive to a clock signal, all the processes must be performed in order, i.e. reading the 8x8 block, DCT, quantization and zigzag. Therefore, careful planning should be done to keep them in order. The same goes for the inverse process, i.e. reading the intermediate file, inverse zigzag, inverse quantization, IDCT and writing the 8x8 blocks to a file. Some of the useful modules, source code and its shell are available for you to copy from the course directory . /ee8205/project/jpeg/

5 What to Hand In

Provide a printout of your properly labeled *.cpp and *.h files of SystemC and the input and output generated by your program. In addition to submitting all the SystemC and other files, hand in 1-2 page report describing the strategy you have used in your implementation. You may also be required to demonstrate and answer any questions about the working of your JPEG encoding and decoding techniques.

6 References and Reading Material

i) The JPEG still picture compression standard *by Gregory K. Wallace,* IEEE Transactions on Consumer Electronics, Vol 38, No. 1, February 1992.

ii) Video compression makes big gains *by Ang et. al,* IEEE Spectrum, October 1991.

Appendix

(a) Bitmap File Header Details

The bitmap header comprises of two structured sections: Bitmap section, which lets us know that the file is a bitmap file, and the Info section, which gives all the information about the following bitmap. The header information only consists of 54 bytes. The structures are as follows:

```
struct Bitmap_Header {
        unsigned short int type; /* Bitmap Identifier = 'B''M' */
        unsigned int size; /* File size in bytes */
        unsigned int reserved; /* = 0 */
        unsigned int offset; /* Offset to image data in bytes = 54 */
} Bitmap_Header;
```

size is the file size, and must include the header size. Therefore, it is calculated as:

54 + height * ((width + *fillingbits*) * 3)

fillingbits is the number of bits required to make image into a square image: i.e., if the image is 62x64 pixels then, the fillingbits would equal 2 to make the image 64x64 pixels.

The number 3 is the bytes per pixel, which comes from bits per pixel. Therefore, for an 8-bit image, this number would be replaced with a 1.

```
struct Info_Header {
        unsigned int size; /* Header size in bytes = 40 or 0x28 */
        int width_image, height_image; /* Width and height of image */
        unsigned short int planes; /* Number of colour planes = 1 */
        unsigned short int bits; /* Bits per pixel = 24 */
        unsigned int compression; /* Compression type = 0 */
        unsigned int imagesize; /* Image size in bytes 0 */
        int xresolution, yresolution; /* Pixels per meter = 0xB40 & 0xB40 */
        unsigned int ncolours; /* Number of colours = 0 */
        unsigned int impcolours; /* Important colours = 0 */
} Info_Header;
```

imagesize can be simply set to zero, without any change to the bitmap image

(b) JPEG File Header Details

The jpeg header is a little bit more complicated and is not so ordered as the bitmap header. It consists of a few more structures and takes into account the use of markers to distinguish between the different parts of the header, as it can be written in the file in any given order. The size of the header information is therefore not constant and varies from file to file. The header structures is given as following:

The jpeg image must start with a Start of Image marker, which is a two byte word

```
    unsigned short int SOI; /* start of image marker = 0xFFD8 */
```

Following this is the jpeg application header segment as follows:

```
    /* JPEG application Header */
    struct JPG_App_Header {
        unsigned short int APP; /* Application segments marker = 0xFFE0 */
        unsigned short app_len;
            /* length of header = 16 for usual JPEG, no thumbnail */
        char identifier[5]; /* = "JFIF",'\0' */
        unsigned short version; /* = hi nibble = 1, low nibble = 1, so, = 0x0101 */
        unsigned char units; /* = 0 */
        unsigned short x_density; /* = 0x60 */
        unsigned short y_density; /* = 0x60 */
        unsigned char x_thumbnail; /* = 0 */
        unsigned char y_thumbnail; /* = 0 */
    } JPG_App_Header;
```

Following the Application Header, are the following headers in any order:

```
    /* Quantization table Header */
    struct Quantization_Header {
        unsigned short int DQT; /* Quantization table marker = 0xFFDB */
        unsigned short quant_len; /* = 132 */
        unsigned char ty_number; /* 0x00 */
            /* bit 0..3: number of Quantization table: table for Y component*/
            /* bit 4..7: precision of Quantization table, 0 for 8 bit */
        unsigned char ty_values[64]; /* the table values */
        unsigned char tcb_number; /* = 0x10: (quant table for Cb, Cr)
        unsigned char tcb_values[64]; /* the table values */
    } Quantization_Header;
```

```
/* Start of Frame Header */

struct SOF_Components {
        unsigned char id; /* the component id */
        unsigned char samp_f; /* sampling factor: bit 0..3 vertical, 4..7 horizontal */
        unsigned char quant_no;
                /* quantization number of the table to use for the component */
} SOF_Components;

struct SOF_Header {
        unsigned short int SOF; /* Start of Frame marker = 0xFFC0 */
        unsigned short sof_len; /* = 17 for truecolor */
        unsigned char precision; /* 8 for 8 bit images */
        unsigned short height_image; /* height of image */
        unsigned short width_image; /* width of image */
        unsigned char components; /* 1 for grayscale JPEG, 3 for color */
        struct SOF_components sofcomponents[3];
                /* the number of elements depends on the value of components */
                /* id, sampling factor and quantization table # for each component */
                /* component[0] = Y, component[1] = Cb, component[2] = Cr */
} SOF_Header;

/* Huffman Table Headers */
struct HUFF_DC_Y { /* Huffman table header for luminance DC values */
        unsigned short int DHT; /* Huffman table marker = 0xFFC4 */
        unsigned short huff_len;
                /* huffman table header length = 31 for this header */
        unsigned char DCy_huffinfo;
                /* bit 0..3: number of Huffman table (0..3), for Y=0 */
                /* bit 4: type of Huffman table , 0 = DC, 1 = AC */
                /* bit 5..7: not used must be 0 */
        unsigned char DCy_nrcodes[16]; /* = DC_luminance_nrcodes */
                        /* nrcodes are = at index i = num of codes with length i */
        unsigned char DCy_values[12]; /* = DC_luminance_values */
} HUFF_DC_Y;

struct HUFF_AC_Y { /* Huffman table header for luminance AC values */
        unsigned short int DHT; /* Huffman table marker = 0xFFC4 */
        unsigned short huff_len; /* header length = 181 */
        unsigned char ACy_huffinfo; /* = 0x10 */
        unsigned char ACy_nrcodes[16]; /* = AC_luminance_nrcodes */
        unsigned char ACy_values[162]; /* = AC_luminance_values */
} HUFF_AC_Y;
```

```
struct HUFF_DC_CB_CR { /* Huffman table header for chrominance DC values */
        unsigned short int DHT; /* Huffman table marker = 0xFFC4 */
        unsigned short huff_len; /* header length = 31 */
        unsigned char DCcbcr_huffinfo; /* = 0x01 */
        unsigned char DCcbcr_nrcodes[16]; /* = DC_chrominance_nrcodes */
        unsigned char DCcbcr_values[12]; /* = DC_chrominance_values */
} HUFF_DC_CB_CR;

struct HUFF_AC_CB_CR { /* Huffman table header for chrominance AC values */
        unsigned short int DHT; /* Huffman table marker = 0xFFC4 */
        unsigned short huff_len; /* header length = 181 */
        unsigned char ACcbcr_huffinfo; /* = 0x11 */
        unsigned char ACcbcr_nrcodes[16]; /* = AC_chrominance_nrcodes *.
        unsigned char ACcbcr_values[162]; /* = AC_chrominance_values */
} HUFF_AC_CB_CR;
```

Following these headers is the Start of Scan header, which is the last header before the image data begins.

```
/* START OF SCAN header
struct SOS_Components {
        unsigned char id; /* id for the component */
        unsigned char tbl_no;
                /* bit 0..3: AC table (0..3), bit 4..7: DC table (0..3) */
} SOS_Components;
struct SOS_Header {
        unsigned short int SOS; /* Start of Scan header marker  = 0xFFDA */
        unsigned short sos_len; /* length of header = 12 */
        unsigned char components; /* number of color components in the header */
                /* 1 for grayscale, 3 for truecolor */
        struct SOS_Components soscomponents[3];
                /* the number 3 depends on the value of components */
                /* id and huffman table number for each component */
                /* component[0] = Y, component[1] = Cb, component[2] = Cr */
        unsigned char Ss, Se, Bf; /* not important, they should be 0,0x3F,0 */
} SOS_Header;
```

After these headers is the jpeg image data and at the end of image data i.e. End of Image marker must be placed as following.

```
unsigned short int EOI; /* End of Image marker = 0xFFD9 */
```