SystemC Based SoC Accelerator Design: JPEG Encoder/Decoder Unit

COE838: System-on-Chip Design

Lab 2b

1. Objectives

The purpose of this lab is to develop a SystemC based JPEG encoder/ decoder unit for an SoC. A hardware/software co-design method will be employed in this design, where students will obtain an understanding of functions that are more suitable for hardware execution versus software. Using the template and JPEG image files provided, students will code the modules and signals needed for designing a JPEG encoder/decoder unit.

2. JPEG Encoding & Decoding Overview

JPEG compression is a common method for implementing lossy compression of digital images. Compression is especially useful for storage and transmission purposes. Using compression, images are formed into a stream of bytes and may be stored (or transmitted) at a fraction of its original size. These images are then decompressed back to an image when they are needed. The steps involved for successful JPEG compression are as follows:

- Discrete Cosine Transform (DCT)
- Quantization
- Zigzag
- Huffman Encoding (i.e. entropy encoding)

JPEG decompression requires the inverse of these steps, i.e. Huffman decoding, unzigzag, etc. The file's header information is also set aside prior to compression, and is included in the compressed file so that the JPEG may be restored to its original state once decompressed.

For the sake of simplicity, you will be provided with a greyscale **.bmp** file for JPEG compression and decompression in this lab. Why not an actual JPEG? JPEG headers often vary in size depending on the file selected, colour depth etc. Bitmaps on the contrary possess a standard header size and are generally easier to decode in this respect. However the basic ideology, theory, and practicality of compression and decompression remain the same for both image types.

The process of JPEG-based encoding and decoding vary according to color depth (8, 24 or 32 bits). The beginning of an image contains header information - in this case, 54 bytes. This includes information pertaining to image width and height, image file size, image color depth, etc. These 54 bytes must be taken into account whenever working with images. Following the 54-byte header, an image stores raw pixel-by-pixel color values in terms of RGB (Red Green Blue). The pixel's color values may vary depending the color depth selected. For an 8-bit image, the depth is one byte (8-bits) per pixel, for a 32-bit image- 4 bytes per pixel, etc. In this lab, we will be working with an 8-bit pixel image.

Taking a look again at the steps in compression: Prior to DCT, Quantization etc, a pre-processing stage is necessary. In the case of this lab which uses an 8-bit pixel image, the pre-processing stage divides image

data into 8x8 blocks, and then shifts all the numbers from unsigned integers with range $[0, 2^8 - 1]$ into signed integers with a range of $[-2^7, 2^7 - 1]$. These integers are then individually compressed as an 8x8 block. Subsequent stages to pre-processing are explained in the following subsections:

(i) DCT and Inverse DCT (IDCT)

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

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

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

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

Note that since DCT and IDCT are time consuming (i.e the equations consist of multiple embedded loops), implementing this stage as a dedicated hardware unit will likely increase system performance. Therefore, these stages will be designed as a hardware module in your SystemC JPEG encoder/decoder. Also note that a cosine table will need to be transferred to the module to keep the hardware area minimized.

(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:

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

Since quantization involves simple value lookups of a hardcoded matrix and multiplication with an 8x8 block, this step may be implemented as a software process.

Note that inverse of quantization is the multiplication of 8x8 block by the quantization table as follows:

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

(iii) Zigzag

The zigzag step takes the quantized 8x8 block and orders them in a 'zig-zag' sequence, resulting in a 1-D array of 64 entries, as 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.

As this step also requires simple matrix manipulation and mathematics, zigzag and un-zigzag may also be implemented as software processes.

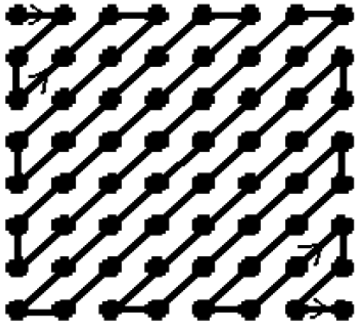

Figure 1: ZigZag Sequence

(iv) Entropy Encoding/Huffman

Subsequent to the zigzag stage, the 64 (or less) values are then encoded for further compression. This stage however will **not be implemented** in this lab to keep it simple.

3. Provided Materials and Requirements

Students are to design a SystemC hardware/software unit that performs JPEG compression and decompression using the steps specified in Section 2. Consider the functions that should be implemented in software and those that should be executed in hardware (as hinted above). For this lab, assume that your design consists of:

- A CPU (sc_main) for executing software processes needed by the JPEG unit, and
- Dedicated hardware modules for executing the performance critical functions. A block diagram of the system to be designed is provided in Figure 2.

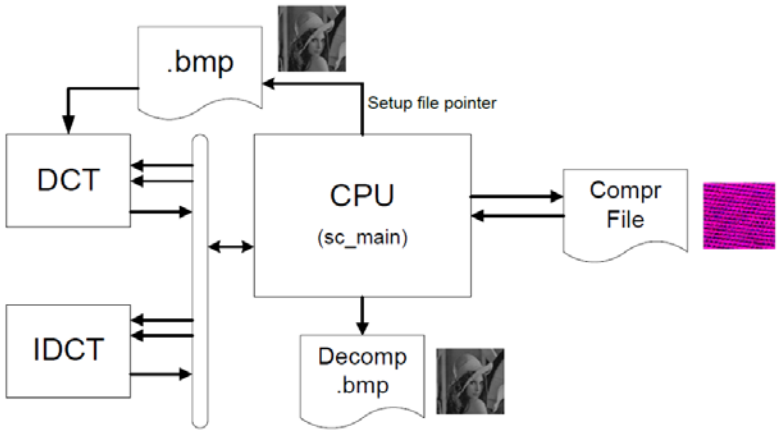

Figure 2: SystemC JPEG Unit Block Diagram

Navigate to the course folder coe838/lab2b/jpeg. This folder contains template code for the lab, along with the .bmp file to be used for testing. The functions.cpp/.h file provides all the software functions necessary for compression and decompression. It is your responsibility to understand these functions and call them in the correct order. Note that functions.cpp also provides helper software for transferring input image header information to your output compressed/decompressed image.

To get you started with hardware design, you are also provided with a DCT hardware module (DCT.cpp and DCT.h). sc_main.cpp should be used as a template for implementing your lab, as it contains many comments that will help you call the ordered functions, and bind hardware modules accordingly.

Copy and paste all files in this folder to your home directory. Open the files and try to understand all code and functionality provided. Note that compression and decompression on the JPEG is completed in two separate steps. Therefore, you must understand the usage of the executable- to compress an image:

```
./sc_jpeg.x <input_filename>.bmp <compressed_file> C
```

and to decompress:

```
./sc_jpeg.x <compressed_file> <output_filename>.bmp D
```

For JPEG compression: ensure that you call the necessary software processes and integrate the provided DCT block according to the steps and diagrams outlined above. For JPEG decompression: you will need to code the IDCT block and call the necessary software functions in the reverse order, i.e unzigzag, unquantization, IDCT, and write back to file. Once more, the order which you call these functions is imperative to the correctness of your JPEG unit.

For the interested reader, details pertaining to bitmap and JPEG header information are provided in the Appendix.

4. What To Hand In

This lab is due week 5 during your lab session. Using the description and diagrams provided above, students are to develop a JPEG encoder/decoder unit using SystemC. To verify that your unit is functioning correctly, your compressed file should resemble the picture given in Figure 2, with your final decompressed .bmp identical (with some image quality loss) to that of the original bitmap.

Please print out and hand in all the SystemC code necessary for implementing this design. Include the input and outputted files generated by your program. In addition, students are to hand in a 1-2 page report describing the strategy undergone to implement the solution to this problem. This lab is to be handed in with the University cover page, dated and signed. Your lab instructor may also ask you to demo your work at the time of submission. Be prepared to answer any questions about your lab.

Appendix

(a) Bitmap File Header Details

The bitmap header comprises of two structured sections: Bitmap section, which identifies the file as 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 filling-bits would equal 2 to make the image 64x64 pixels.

The number 3 is the bytes per pixel, which comes from the 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, and 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 are given below:

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 */
```

It is followed by the jpeg application header segment as given below:

```
/* 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 various headers that can be 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;
```

After these headers is the Start of a 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 where at the end of the image data an “End of Image marker” must be placed as following.

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