ECE744 Project Description
Hardware Implementation of an Image Compressor

Objective

To gain experience in digital system design by implementing the custom *McMaster Image Compression* (.mic) image compression specification in hardware. An image of size between 16 x 16 and 640 x 480 pixels will be delivered to the Altera DE2-115 board via the universal asynchronous receiver/transmitter (UART) interface from a personal computer (PC) and stored in the external static random-access memory (SRAM). The image compression circuitry will read the raw red/green/blue (RGB) image, compress the image using a custom digital circuit designed by you and store the compressed data back to the SRAM, from where it will be sent back to the PC over UART, for decompression in software. It is fair to assume that rows and columns are a multiple of 8.

Preparation

- Revise the first five labs, especially finite state machines, SRAM and UART interfaces
- Read this document and get familiarized with the C source code

The first part of this document gives an overview of image compression basics, introduces the concepts required for the project, and provides a detailed example of the decompression process to aid in understanding image compression/decompression and in implementing this specification. The second part of the document provides project guidelines for each of the three milestones.

Image Compression

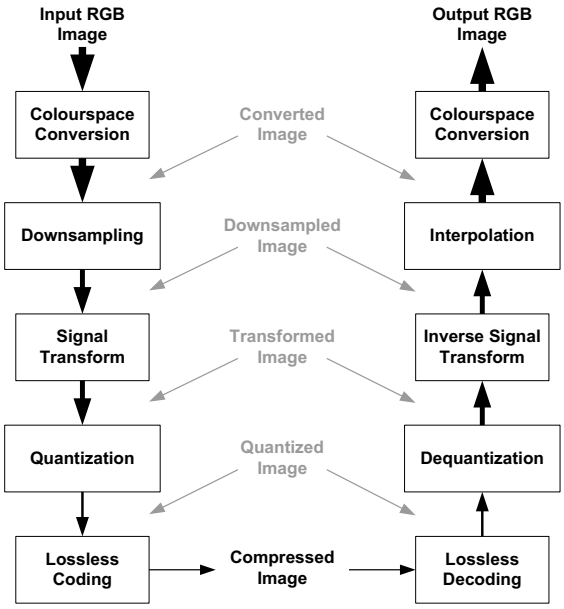


Figure 1 – Basic image compression/decompression

Figure 1 shows the conceptual flow for basic image compression (down the left) / decompression (up the right), where the relative amount of information at each stage is roughly indicated by the size of the arrow. In general, during image compression, an image in standard red/green/blue (RGB) format undergoes colourspace conversion, yielding a brightness/colour or luminance/chrominance (YUV) formatted image, which is downsampled, transformed and quantized before lossless coding is performed to produce the

compressed information bitstream. Decoding proceeds in reverse: lossless decoding is applied to obtain the quantized image, after which the dequantization, the inverse signal transform, interpolation and colourspace conversion will restore the image in RGB format. Due to the fact, however, that downsampling and quantization are not bijective operations (not identically reversible), some information is lost during each of these processes, and the inverse restores only an approximation of the original, leading to a finite signal-to-noise ratio between the original and the decompressed images. A good compression scheme leverages this loss of information to reduce the compressed file size but will attempt to allow information loss only in a way that does not drastically affect the quality of the decompressed image.

Before going into detail regarding the concepts of image compression, it should be noted that this project focuses on the hardware implementation of a compression scheme; it is by no means to develop a rival compression standard to existing compressed image formats. For this reason, simplifying choices have been made in developing the specification, which leads to a more manageable hardware design and implementation cycle but limits the capabilities of the specification in terms of achievable compression/quality tradeoffs. Nonetheless, the concepts described here still apply in a general sense to the existing compression schemes.

The following sub-sections explain in detail each step of the compression process. Then, a comprehensive decompression example, based on a compressed 16 x 16 pixel image, is provided to concretize the concepts.

Colourspace Conversion and Downsampling

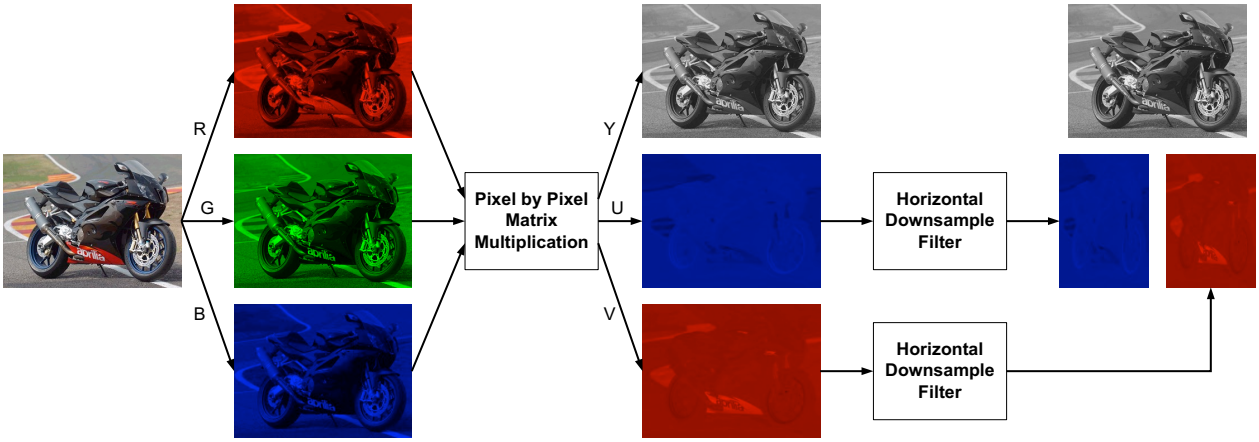


Figure 2 – Colourspace conversion and downsampling

Figure 2 shows the processes of converting the colourspace and downsampling applied to a sample image of size 320 x 240. On a pixel by pixel basis, the R, G and B values which define the colour for a given pixel are interpreted as a vector in 3-space. Multiplication by the colourspace conversion matrix produces another value in 3-space, however the matrix multiplication amounts to a change of basis from the RGB basis to the YUV basis. The values are also range adjusted by vector addition. Equation 1 encapsulates the conversion used for this project (the uppermost part of equation 1 is the floating-point equation, while the part directly below it is the fixed-point approximation for implementation in hardware).

$$\begin{bmatrix} \mathbf{Y} \\ \mathbf{U} \\ \mathbf{V} \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{Y} \\ \mathbf{U} \\ \mathbf{V} \end{bmatrix} = (int)\tfrac{1}{65536}\left(\begin{bmatrix} 16843 & 33030 & 6423 \\ -9699 & -19071 & 28770 \\ 28770 & -24117 & -4653 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{bmatrix} + \begin{bmatrix} 32768 \\ 32768 \\ 32768 \end{bmatrix} \right) + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \quad (1)$$

$$\mathbf{U}[j] = 0.043\mathbf{U}'[2j-5] - 0.102\mathbf{U}'[2j-3] + 0.311\mathbf{U}'[2j-1] + 0.5\mathbf{U}'[2j] + 0.311\mathbf{U}'[2j+1] - 0.102\mathbf{U}'[2j+3] + 0.043\mathbf{U}'[2j+5]$$

$$\mathbf{U}[j] = (int)\tfrac{1}{512}\left(22\mathbf{U}'[2j-5] - 52\mathbf{U}'[2j-3] + 159\mathbf{U}'[2j-1] + 256\mathbf{U}'[2j] + 159\mathbf{U}'[2j+1] - 52\mathbf{U}'[2j+3] + 22\mathbf{U}'[2j+5] + 256\right)$$

Given that the human eye is much more sensitive to brightness than colour (due to the presence of more rods than cones in the eye), the colour information is not as relevant to the human eye as the brightness information of the picture. For this reason, the U and V pictures can be downsampled to reduce the amount of information that needs to be compressed without too noticeably affecting how the picture is perceived. Downsampling is done horizontally, leading to U and V planes, which are half as wide as the original image, and thus, overall, only 2/3 of the data of the original image remains to be compressed. The equation for downsampling is given in the lower half of Equation 1, and as before, the upper part (third row of Equation 1) is the floating-point casting, while the fixed-point approximation is given below the floating-point. The value **U'** (and analogously **V'**) is a sample from the full image, while **U** (and analogously **V**) belongs to the downsampled image.

It is important to note that the terminology YUV originates from the video transmission field, and different notations (depending on the video/colour/transmission standard) can be found in the literature (such as YIQ, YPbPr, YCbCr). In this document and in the C code provided, we use the YUV notation where Y stands for the luma component, and U and V stand for the colour difference components for Blue and Red, respectively.

Signal Transform and Quantization

The next stage of compression involves a signal transform to exchange the spatial location information with the spatial frequency information. Since high spatial frequencies (sharp oscillations in intensity over small spatial ranges) are not as noticeable to the eye, they may be reduced (or even removed) without drastically affecting the overall quality of the image. The signal transform selected for this project is the *Discrete Cosine Transform* (DCT) which is commonly used in different forms for multimedia compression in general (video, image and audio). The DCT used in this project is a two-dimensional transform, and works on a block of values, represented as a matrix (in our case of size 8 x 8) according to the formula:

$$\mathbf{S}' = \mathbf{C}\mathbf{S}\mathbf{C}^T,\ [\mathbf{C}]_{i,j} = \begin{cases} \sqrt{\frac{1}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) & i=0; j=0..7 \\ \sqrt{\frac{2}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) & i=1..7; j=0..7 \end{cases} \quad (2)$$

$$\mathbf{S}' = (int)\tfrac{1}{65536}\left(\mathbf{C} \times (int)\tfrac{1}{256}(\mathbf{S}'\mathbf{C}^T + 2^7\mathbf{1}) + 2^{15}\mathbf{1}\right),\ [\mathbf{C}]_{i,j} = \begin{cases} (int)\left(\sqrt{\frac{1}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & i=0; j=0..7 \\ (int)\left(\sqrt{\frac{2}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & i=1..7; j=0..7 \end{cases}$$

(floating-point above, fixed-point below) where **S** is the original (sample) block and **S'** is the transformed block (**1** denotes a column vector of the appropriate size, where each entry has value 1). Once a block of values has been transformed, it is quantized using pointwise integer division by a quantization matrix, with rounding performed on each entry by adding the value 0.5 and applying the floor function. In this compression scheme, one of the three quantization matrices below (Equation 3) may be selected for image compression to provide a tradeoff between image quality and compression ratio. Pointwise integer division **A** = **B** ./ **C** of matrices **A**, **B** and **C** is defined below in Equation 4.

$$\mathbf{Q}_0 = \begin{bmatrix} 8 & 4 & 8 & 8 & 16 & 16 & 32 & 32 \\ 4 & 8 & 8 & 16 & 16 & 32 & 32 & 64 \\ 8 & 8 & 16 & 16 & 32 & 32 & 64 & 64 \\ 8 & 16 & 16 & 32 & 32 & 64 & 64 & 64 \\ 16 & 16 & 32 & 32 & 64 & 64 & 64 & 64 \\ 16 & 32 & 32 & 64 & 64 & 64 & 64 & 64 \\ 32 & 32 & 64 & 64 & 64 & 64 & 64 & 64 \\ 32 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \end{bmatrix} \mathbf{Q}_1 = \begin{bmatrix} 8 & 4 & 4 & 4 & 8 & 8 & 16 & 16 \\ 4 & 4 & 4 & 8 & 8 & 16 & 16 & 32 \\ 4 & 4 & 8 & 8 & 16 & 16 & 32 & 32 \\ 4 & 8 & 8 & 16 & 16 & 32 & 32 & 32 \\ 8 & 8 & 16 & 16 & 32 & 32 & 32 & 32 \\ 8 & 16 & 16 & 32 & 32 & 32 & 32 & 32 \\ 16 & 16 & 32 & 32 & 32 & 32 & 32 & 32 \\ 16 & 32 & 32 & 32 & 32 & 32 & 32 & 32 \end{bmatrix} \mathbf{Q}_2 = \begin{bmatrix} 8 & 2 & 2 & 2 & 4 & 4 & 8 & 8 \\ 2 & 2 & 2 & 4 & 4 & 8 & 8 & 16 \\ 2 & 2 & 4 & 4 & 8 & 8 & 16 & 16 \\ 2 & 4 & 4 & 8 & 8 & 16 & 16 & 16 \\ 4 & 4 & 8 & 8 & 16 & 16 & 16 & 16 \\ 4 & 8 & 8 & 16 & 16 & 16 & 16 & 16 \\ 8 & 8 & 16 & 16 & 16 & 16 & 16 & 16 \\ 8 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \end{bmatrix} \quad (3)$$

$$\mathbf{A} = \mathbf{B}./\mathbf{C} \Leftrightarrow [\mathbf{A}]_{i,j} = [\mathbf{B}]_{i,j}/[\mathbf{C}]_{i,j} \quad (4)$$

By quantizing each value, information to which the eye is less sensitive (i.e., the high spatial frequency information) is removed, leading to less information overall being stored. This is why the values in the quantization matrices increase when moving to the right and downward; these entries represent higher spatial frequencies. For typical images, many of the transformed values will become zero after quantization, and these zeros may be stored in an efficient manner. This is because, in typical images, low spatial frequencies are dominant. The values which remain non-zero after quantization also benefit by being reduced in magnitude to permit representation on fewer bits.

It should be noted that matrix $\mathbf{Q}_0$ from Equation 3 above has larger values than $\mathbf{Q}_1$ and $\mathbf{Q}_2$ and thus gives a more coarse resolution for each spatial frequency. Because of this, the usage of $\mathbf{Q}_0$ will result in poorer quality of compression, but also a smaller amount of compressed data (due to smaller quotients) than in the case where $\mathbf{Q}_1$ or $\mathbf{Q}_2$ is used during quantization.

To perform the transform on the whole image, each plane (the Y, downsampled U and downsampled V) is divided into blocks of 8 x 8 pixels, and the blocks are transformed one after another, starting at the top left hand corner of the Y plane, proceeding by rows of blocks (from left to right along each row), repeating for the downsampled U and V planes. Figure 3 shows the process applied to two blocks from the Y plane of the sample image introduced earlier. Since this image is of size 320 x 240, transformation and quantization will be performed on the 1200 (320/8 x 240/8) blocks from the Y plane and the 600 blocks from each of the U and V and planes, amounting to 2400 blocks in total. Notice that the first (upper) block of pixels in the figure is fairly uniform, which can be seen in the numerical values of those pixels as shown in the matrix next to it, where the lower block exhibits more variation in intensity.

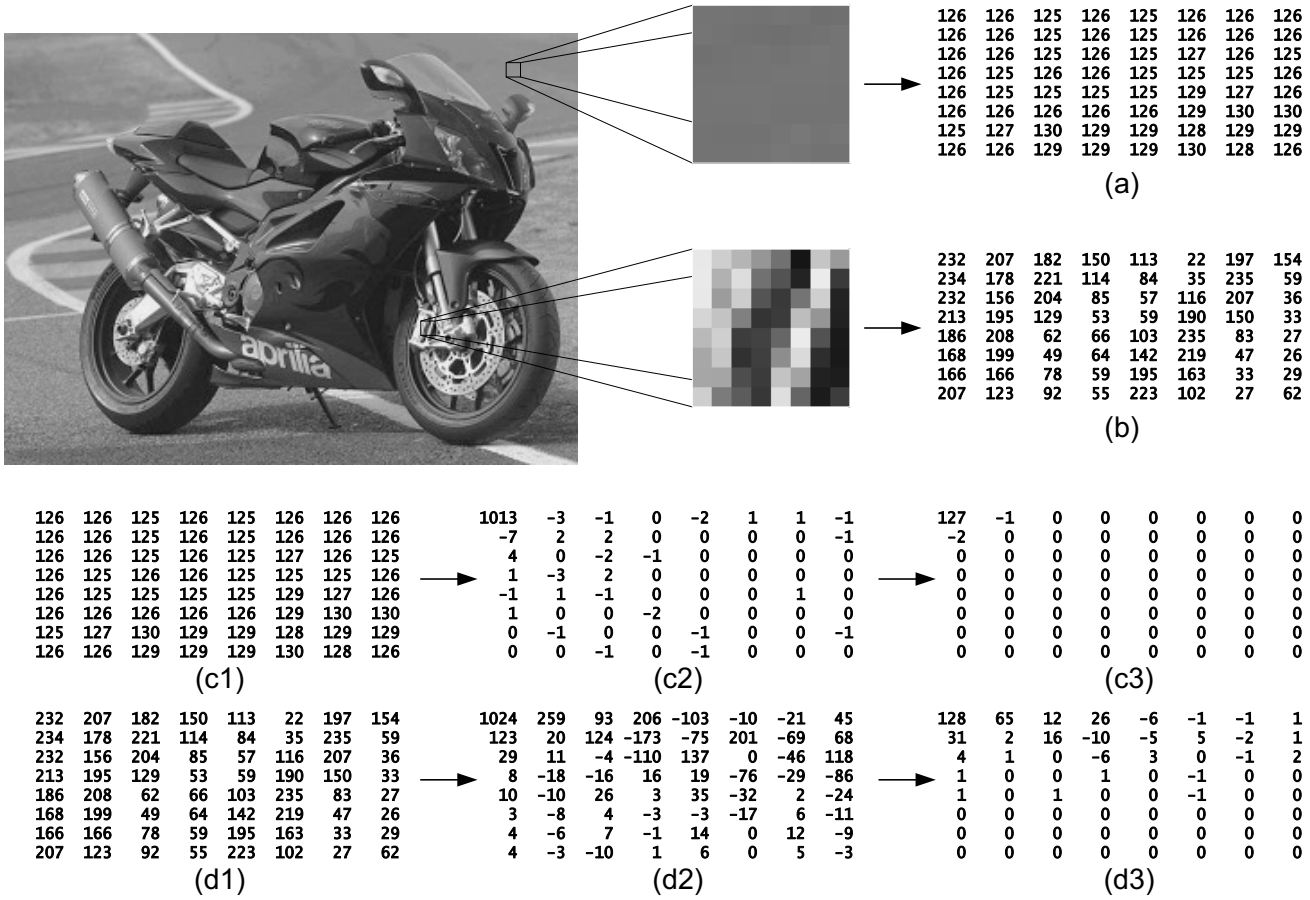


Figure 3 – Transformation and quantization of a block

The first sequence of matrices below the image (Figure 3(c1) to (c3)) shows the transform applied to the upper pixel block (Figure 3(a)), then quantization using $\mathbf{Q}_0$ from Equation 3. Due to the uniformity of the block, the transform coefficients (the second matrix in the sequence) corresponding to the higher frequencies (those lower and to the right) have small or zero values. If efficient lossless coding is employed, as discussed in the next section, a smaller amount of data can represent these small and zero-valued coefficients. Quantization (the results appearing in the third matrix in the sequence) further reduces the amount of information necessary by producing more zeros and shrinking the remaining non-zero coefficients so they can be represented on fewer bits. Although this leads to reduced quality of the decompressed image, the major features of the original block are still preserved.

The second sequence of blocks (Figure 3(d1) to (d3)) shows the same steps applied to the lower block of pixels (Figure 3(b)) from the Y plane. Note the presence of many more high-frequency components, which result in more data being required to represent the block than in the case of the previous block. Even still, the amount of data is reduced from that required to represent the intensity of each pixel explicitly (i.e. the first block in both sequences). The next sub-section discusses lossless coding, which provides an efficient representation to leverage the advantages of the transformed and quantized representation of image blocks.

Lossless Coding

The final stage in the compression process is the lossless coding of the blocks coming from the transform/quantization. Before packing this information into the bitstream, a simple 20-byte header is provided, which indicates the type of quantization matrix used, the image's dimensions and the offset of the encoded Y/U/V data in the bitstream. After this, the losslessly coded blocks are just packed into the bitstream one after another in the same order as described above for performing the transform and

quantization. To losslessly code a single block, the coefficients are first scanned out of the block into the order they will appear in the compressed bitstream (known as the scan pattern). The scan pattern used in this project is shown in Figure 4.

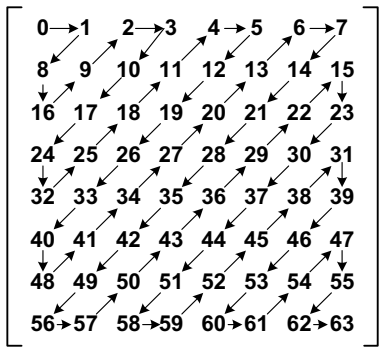


Figure 4 – Scan pattern for placing coefficients in the bitstream

Because quantization leads to a significant portion of the coefficients being 0, it is desirable to group as many zeros together as possible, especially at the end of the block, since we can use an efficient code to represent such situations. As discussed in the previous section, the higher frequencies (toward the bottom right) most often have coefficients with value 0. This motivates the scan pattern above (which is standard for many mainstream video/image compression schemes), where the order tries to take advantage of such scenarios.

Finally, as the coefficients are processed in this order, we identify four possible cases in the bitstream using 2-bit codes followed by either 3 or 9 bits. These cases and their associated actions are as follows, in order of decreasing priority:

1. only zeros remain in the block; provide an end-of-block (EOB) code (bits 11).
2. a run of zeros exists, and some non-zero coefficients remain in the block, and:
    a. the run of zeros is 8 or longer; provide a 0-run code (bits 00), then the 3 bits 000; repeat step 2 (the run of zeros will be shorter by 8 now).
    b. If the run of zeros is less than 8 in length, provide a 0-run code (bits 00), then 3 bits indicating the length of the run.
3. a non-zero coefficient between -4 and 3 inclusive; provide a short coefficient code (bits 10) followed by the 3-bit 2's complement value.
4. a coefficient between -256 and 255 inclusive but not between -4 and 3; provide a long coefficient code (bits 01) followed by the 9-bit 2's complement.

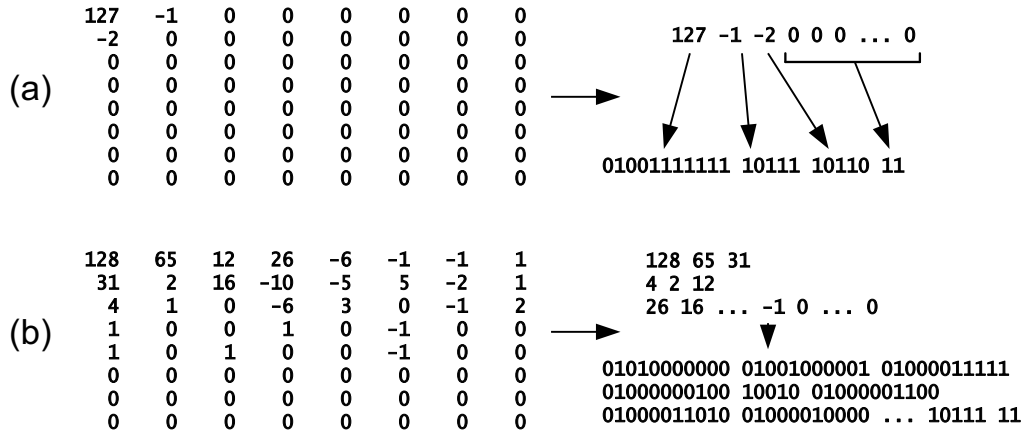


Figure 5 – Lossless coding of a block

Figure 5 shows the coding of the final (quantized) blocks from Figure 3; in both cases, the blocks are scanned according to Figure 4. For the first block, the first coefficient (127) is mapped to a 01 (to indicate a long coefficient, between -256 and 255) followed by 001111111, the 9-bit 2's complement representation of 127. Theoretically, it is possible to produce synthetic images, which, when quantized with $\mathbf{Q}_2$ from Equation 3, can produce values outside the valid range between -256 and 255. To deal with such anomalies, which have not been exercised by any real-life image yet are theoretically possible, the quantized values are checked to see if they fit into the range between -256 and 255 and if not, they are clipped to the range boundaries (i.e., for large negative values -256 and +255 for large positive values).

The next coefficient is -1, which is between -4 and 3, so it is mapped to 10 to indicate a short coefficient, followed by 111, the 3-bit 2's complement representation of -1. Similarly, we obtain 10-110 for -2. Only zeros remain in the block at this point, so an end-of-block code (bits 11) is placed in the bitstream, and the block has been successfully coded.

Coding of the second block from Figure 3 proceeds similarly; however, as alluded to in the previous section, more data is required due to the presence of more non-zero coefficients and the larger magnitude on average of those coefficients. As before, encoding proceeds until the last non-zero coefficient in the scanned sequence (-1) is reached, after which an end of block code (bits 11) is placed in the bitstream. When the last block of the downsampled V plane has been coded, zero padding ensures the file ends at the end of a 16-bit word (for compliance with the 16-bit memory we are using).

To thoroughly demonstrate the concepts which have just been introduced here and to thereby ensure a firm understanding, a detailed example of the decompression process applied to a compressed bitstream is provided in the following sections.

Image Decompression

In this section, the full decompression process for an image of size 16 x 16 is performed for the purpose of illustrating the flow of data from one stage to another, and to elaborate the internal details of each of the steps outlined in the preceding sections. It should be noted that since the aim of this project is the hardware implementation of the specification, fixed point arithmetic is used for all arithmetic operations.

5 byte header replaced
by a 20 byte header

```
0A 00 10 00 10 4E 88 29 03 C5 F6 BF 64 0B 1B 58
31 7F 71 15 AB 18 DE 22 A4 4B 70 C4 77 0D 80 3B
1E C3 BA 7B 7E F6 40 88 3D FC 55 03 20 5B 89 FC
C6 77 AD AE 1B 86 E1 40 A2 88 E3 13 58 37 25 C9
23 D8 57 92 73 41 16 9A 14 AB F8 AC 6D 6B BF 7A
CA 32 0D E2 28 85 02 44 51 88 62 7B 06 20 0C 8F
70 DC 31 50 68 1A 4F E1 F8 45 FD 55 FD 0A 2F DE
F7 8C 63 38 D4 37 8D 22 5B C6 31 B8 62 1B 00 76
3D 87 65 04 85 90 CA 0E 7E 68 2D 02 D8 37 8C 68
81 50 25 5F D3 FA B4 C7 71 C4 37 0C 6C 1B 80 01
BE A3 D0 FB F6 41 C8 1D 08 BE D4 0E FE AD 28 BF
9A C6 72 40 91 BC EF 70 DC 37 88 63 71 5C 31 14
43 1D 3F 20 A8 BF 60 BF 47 EC 1A DA C1 40 97 8C
62 81 25 71 C4 51 BD C3 7B C7 4F 0F D9 02 BF AB
3F 94 0F 59 06 E1 98 AF 7B 06 37 35 C3 78 E0 00
```

Figure 6 – Compressed bitstream "Example.mic"

Consider the compressed bitstream "Example.mic" shown in Figure 6. Note that the 5-byte header from the figure is replaced in the current version of the standard by a 20-byte header. For the 20-byte header, the first three bytes must be "EC", "E7" and "44" and they uniquely identify the current version of the "mic" file (previous/future revisions/extensions may use a different identifier). The following byte indicates which quantization matrix to use. The next 2 bytes provide the vertical size (number of rows), and the following 2 provide the horizontal size (number of columns). Each of the 4 bytes from the remaining 12 bytes provides the byte offset (the first three bytes of the four bytes) and the bit offset (last byte of the four bytes) of the Y, U and V segments, respectively, in the bitstream (these fields have been added in newer revisions to reduced one step during the real-time decompression of a stream of .mic-encoded images). After this, the main content of the compressed image, i.e., the sequence of compressed blocks, begins.

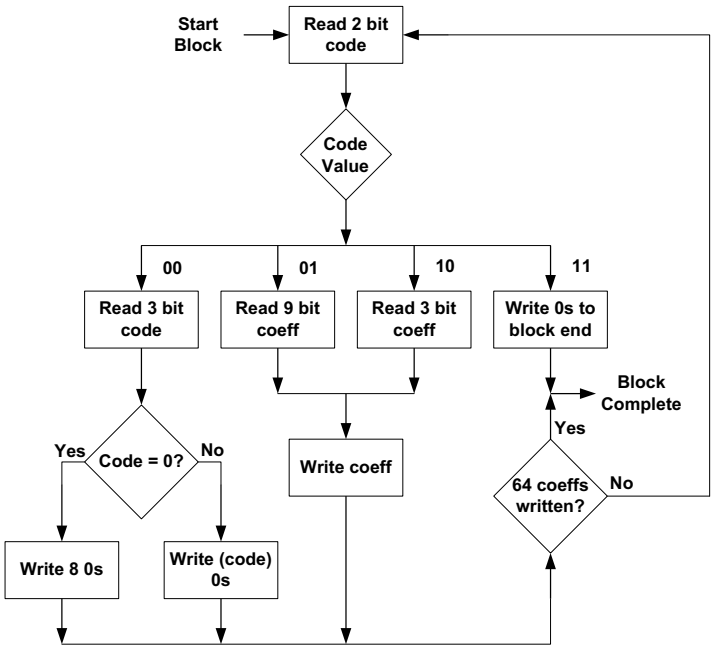


Figure 7 – Decoding a block

Figure 7 depicts the block decoding process. To start, 2 bits are checked for the code type and for the case of (00), a run of zeros is placed; for (01), a 9-bit coefficient follows; for (10), a 3-bit coefficient follows; and for (11), the end of the current block has been reached. Following this procedure, zeros and coefficients are decoded one after another and placed in the 8x8 block according to the scan pattern of Figure 4. A block is complete either when an end-of-block code is decoded (in which case the rest of the block is filled with 0s) or when 64 coefficients (including runs of zeros) have been decoded, making the block full. The sequence begins with hex “4E 88 29 …” which is “010011101000100000101001…” in binary, and when applied to this bitstream, the result of the lossless decoding procedure is as follows:

```
Y 0,0 -------------

01001110100 : 116     10101       : -3        00000       : 0 0 0 0 0 0 0 0
01000001010 : 10      10001       : 1         00011       : 0 0 0
01000000111 : 7       10001       : 1         10110       : -2
10001       : 1       10111       : -1        00111       : 0 0 0 0 0 0 0
01111101101 : -19     10001       : 1         10110       : -2
01111110110 : -10     00010       : 0 0       00011       : 0 0 0
01000000101 : 5       10100       : -4        10111       : -1
10001       : 1       10001       : 1
10110       : -2      00101       : 0 0 0 0 0
10110       : -2      10111       : -1
00001       : 0       00001       : 0
10001       : 1       10001       : 1
01111111011 : -5      00011       : 0 0 0
10001       : 1       10111       : -1
00010       : 0 0     00001       : 0
10110       : -2      10110       : -2

Y 0,1 -------------

01001111011 : 123     10110       : -2        10111       : -1
01111110111 : -9      10111       : -1        00100       : 0 0 0 0
10110       : -2      00001       : 0         10111       : -1
01000000100 : 4       10111       : -1        00100       : 0 0 0 0
01000001111 : 15      00001       : 0         10010       : 2
01111111000 : -8      10111       : -1        00111       : 0 0 0 0 0 0 0
10101       : -3      00001       : 0         10110       : -2
01000000110 : 6       01000000101 : 5         00010       : 0 0
01000000101 : 5       00010       : 0 0       10111       : -1
10111       : -1      10001       : 1         10010       : 2
00010       : 0 0     00011       : 0 0 0
01111111001 : -7      10001       : 1
10001       : 1       10001       : 1
10011       : 3       00110       : 0 0 0 0 0 0
10111       : -1      10110       : -2
10101       : -3      00001       : 0

Y 1,0 -------------

01001110011 : 115     10010       : 2         00001       : 0
01000001000 : 8       00001       : 0         10001       : 1
10110       : -2      10111       : -1        00000       : 0 0 0 0 0 0 0 0
10011       : 3       10001       : 1         00001       : 0
01000010100 : 20      00010       : 0 0       10010       : 2
10101       : -3      10001       : 1         00111       : 0 0 0 0 0 0 0
01111111000 : -8      00001       : 0         10111       : -1
10101       : -3      01000000100 : 4         00001       : 0
10001       : 1       10001       : 1         10111       : -1
10110       : -2      00010       : 0 0       00001       : 0
10110       : -2      10001       : 1         10001       : 1
10111       : -1      10001       : 1
01111110111 : -9      00001       : 0
10101       : -3      10001       : 1
10010       : 2       00111       : 0 0 0 0 0 0 0
10001       : 1       10110       : -2

Y 1,1 -------------

01010000011 : 131     10001       : 1         10001       : 1
01000000110 : 6       10011       : 3         00001       : 0
10010       : 2       10001       : 1         10110       : -2
01111111000 : -8      10101       : -3        00000       : 0 0 0 0 0 0 0 0
01111110000 : -16     00001       : 0         00011       : 0 0 0
10001       : 1       10111       : -1        10110       : -2
01111111010 : -6      10001       : 1         00111       : 0 0 0 0 0 0 0
10101       : -3      10100       : -4        10110       : -2
01111111010 : -6      10001       : 1         00011       : 0 0 0
00010       : 0 0     00101       : 0 0 0 0 0 10110       : -2
10001       : 1       10111       : -1
01111110111 : -9      10001       : 1
10111       : -1      10001       : 1
10111       : -1      10001       : 1
10001       : 1       10111       : -1
10001       : 1       00001       : 0
```

(a)

116	10	-10	5	0	0	0	0
7	-19	1	1	-2	0	0	0
1	-2	-5	-3	1	0	0	0
-2	1	1	-4	-1	0	0	0
0	1	0	0	-2	0	0	0
-1	0	1	0	0	-2	0	0
1	0	-1	0	0	0	-2	0
0	0	0	0	0	0	0	-1

(b)

123	-9	-8	-3	3	-1	1	0
-2	15	6	1	-3	0	0	0
4	5	-7	-2	0	0	-1	0
-1	0	-1	5	1	0	0	0
0	0	0	1	-2	0	0	0
-1	-1	0	0	-1	2	0	0
0	0	0	0	0	0	-2	0
0	0	0	0	0	0	-1	2

(c)

115	8	-3	-8	2	1	0	1
-2	20	-3	-3	2	0	1	0
3	1	-9	0	1	0	1	0
-2	-1	-1	4	1	0	0	0
-2	1	0	0	-2	0	0	0
0	1	0	0	0	2	0	0
0	0	0	0	0	0	-1	-1
0	0	0	0	0	0	0	1

(d)

131	6	1	-6	-1	1	0	0
2	-16	-3	-1	1	0	0	0
-8	-6	-9	1	1	0	0	0
0	1	3	-4	-1	0	0	0
0	1	1	1	-2	0	0	0
-3	-1	1	0	0	-2	0	0
0	1	1	0	0	0	-2	0
-1	0	0	0	0	0	0	-2

The Y blocks appear first in the bitstream, row by row. Y 0,0 contains Y data for pixel rows 0-7 and columns 0-7 of the Y plane from the downsampled image (which is equivalent to the Y plane from the converted image since Y is not downsampled). Similarly, Y 0,1 contains rows 0-7, columns 8-15; Y 1,0 contains rows 8-15, columns 0-7; Y 1,1 contains rows 8-15, columns 8-15.

Next in the bitstream comes the data for the U and V planes of the downsampled image, recall that they have half the columns of the Y plane due to downsampling. In the same way as for Y, U 0,0 contains U data for pixel rows 0-7, columns 0-7 of the downsampled image; U 1,0 contains rows 8-15, columns 0-7; V 0,0 contains V data for rows 0-7, columns 0-7; V 1,0 contains rows 8-15, columns 0-7.

```
u 0,0 -------------

01010000010 : 130     01111111010 : -6        00000       : 0 0 0 0 0 0 0 0 0
01000010110 : 22      01111111010 : -6        00001       : 0
01000011001 : 25      10110       : -2        10111       : -1
01000001110 : 14      10011       : 3         11          : 0 0 0 0 0 0 0 0 0
01111110011 : -13     00011       : 0 0 0                   0 0 0
01000001011 : 11      10111       : -1
01000000101 : 5       00011       : 0 0 0
10110       : -2      10001       : 1
00001       : 0       00001       : 0
10111       : -1      10111       : -1
10001       : 1       00001       : 0
10001       : 1       10001       : 1
10100       : -4      10110       : -2
01000000101 : 5       00001       : 0
01000000100 : 4       10111       : -1
10101       : -3      00000       : 0 0 0 0 0 0 0 0 0

u 1,0 -------------

01010001111 : 143     10010       : 2         10111       : -1
01000011111 : 31      01000000100 : 4         00001       : 0
01111110110 : -10     10001       : 1         10001       : 1
01000001110 : 14      10111       : -1        00010       : 0 0
01000000111 : 7       10011       : 3         10001       : 1
01000010001 : 17      10111       : -1        00001       : 0
01111101101 : -19     10111       : -1        10001       : 1
01000000111 : 7       00001       : 0         11          : 0 0 0 0 0 0 0 0 0
01111111010 : -6      10111       : -1                      0 0 0 0 0 0 0 0 0
10110       : -2      00001       : 0                       0 0 0 0 0 0 0 0
10010       : 2       10111       : -1
10001       : 1       10001       : 1
01111111001 : -7      00001       : 0
10101       : -3      10001       : 1
10001       : 1       10111       : -1
10011       : 3       00010       : 0 0

v 0,0 -------------

01001111110 : 126     10001       : 1         11          : 0 0 0 0 0 0 0 0 0
01000001010 : 10      01000000100 : 4                       0 0 0 0 0 0 0 0 0
10001       : 1       10010       : 2                       0 0 0 0 0 0 0 0 0
01111110110 : -10     10111       : -1                      0 0 0 0 0 0 0 0
00001       : 0       00011       : 0 0 0
01111110100 : -12     10001       : 1
01111110110 : -10     00010       : 0 0
00001       : 0       10001       : 1
10101       : -3      10111       : -1
10110       : -2      10111       : -1
10110       : -2      00001       : 0
00001       : 0       10111       : -1
01000000100 : 4       10111       : -1
10111       : -1      10001       : 1
10001       : 1
10001       : 1

v 1,0 -------------

01001111000 : 120     10110       : -2
01111110110 : -10     00001       : 0
01000000101 : 5       10001       : 1
01111111010 : -6      10111       : -1
10110       : -2      00110       : 0 0 0 0 0 0
01111111001 : -7      10111       : -1
01000000111 : 7       00001       : 0
10101       : -3      10111       : -1
10010       : 2       10001       : 1
00001       : 0       11          : 0 0 0 0 0 0 0 0 0
10111       : -1                    0 0 0 0 0 0 0 0 0
00001       : 0                     0 0 0 0 0 0 0 0 0
10011       : 3                     0 0 0 0 0 0 0 0 0
00010       : 0 0                   0
10111       : -1
10111       : -1
```

(a)

130	22	11	5	4	-3	1	0
25	-13	-2	5	-6	0	-1	0
14	0	-4	-6	0	0	0	0
-1	1	-2	0	1	0	0	0
1	3	-1	-2	0	0	-1	0
0	0	0	0	0	0	0	0
0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b)

143	31	17	-19	1	3	1	0
-10	7	7	-3	2	-1	1	0
14	-6	-7	4	0	-1	0	0
-2	1	1	-1	0	1	0	0
2	-1	0	0	0	0	0	0
3	-1	-1	1	0	0	0	0
-1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

(c)

126	10	-12	-10	1	1	-1	-1
1	0	0	-1	1	1	0	0
-10	-3	4	4	0	-1	0	0
-2	0	2	0	-1	0	0	0
-2	-1	1	1	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

(d)

120	-10	-7	7	0	-1	-1	0
5	-2	-3	0	-1	0	-1	0
-6	2	3	-2	0	1	0	0
0	0	0	0	0	0	0	0
-1	1	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Upon completion of decoding, we have 8 blocks of data, 4 for Y (16/8 x 16/8), 2 for U and 2 for V. The matrices down the right above belong to the quantized image from Figure 1, and the next step is dequantization, which leads to the transformed image. This is done using pointwise multiplication between the losslessly decoded data (matrices along the right above) and the appropriate quantization matrix ($\mathbf{Q}_0$ in this case). For matrices **A**, **B** and **C**, pointwise multiplication is defined in Equation 5 below.

$$\mathbf{A} = \mathbf{B}. \times \mathbf{C} \Leftrightarrow [\mathbf{A}]_{i,j} = [\mathbf{B}]_{i,j} \times [\mathbf{C}]_{i,j} \tag{5}$$

$$\mathbf{S} = (int)\frac{1}{65536}\left(\mathbf{C}^T \times (int)\frac{1}{256}(\mathbf{S}'\mathbf{C})\right), [\mathrm{C}]_{i,j} = \begin{cases} (int)\left(\sqrt{\frac{1}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & i = 0; j = 0..7 \\ (int)\left(\sqrt{\frac{2}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & i = 1..7; j = 0..7 \end{cases} \tag{6}$$

After dequantizing the blocks, the *inverse discrete cosine transform* (IDCT) must be performed to recover the downsampled image. The IDCT, defined above in Equation 6, is defined similarly to the DCT from Equation 2, where the matrix **S'** is the dequantized block, and **S**, the output of the transform, is the block of samples from the downsampled image. The important differences are the intermediate and final scalar divisions and the differing (fixed-point) coefficients. To re-iterate, the reason for choosing fixed-point (i.e. integer) operations is that they facilitate a more efficient hardware implementation when compared to floating-point.

Fixed-point, however, brings in necessary approximations that we achieve by using an intermediate bitwidth larger than the one necessary to store the IDCT outputs. For example, the cosine coefficients (which are between -1 and 1) are left-shifted by 12 positions (i.e. multiplied by 4096 as shown in Equation 6) to obtain an integer-only fixed-point value before processing (between -4096 and 4095). Because the sample matrix is multiplied on both sides by the coefficient matrix, the data will be left-shifted by 24 positions in total. A first adjustment of a right shift by 8 positions is provided after the post-multiplication (**S'C**) and another adjustment is given by a right shift by 16 positions after the pre-multiplication ($\mathbf{C}^T$(**S'C**)). The reason why the first adjustment is only on 8 bits is because it retains more precision, and thus, it is more immune to the truncation errors.

Finally, a clipping function is employed to ensure that rounding and overflows leading to values out of the 8 bit range (0 to 255) are addressed by saturation (i.e. values smaller than 0 are forced to 0 and values greater than 255 are forced to 255). The results of dequantization and IDCT are as follows:

Y 0,0

116	10	-10	5	0	0	0	0		928	40	-80	40	0	0	0	0		7	142	145	128	133	136	140	127
7	-19	1	1	-2	0	0	0		28	-152	8	16	-32	0	0	0		131	8	122	139	129	151	134	131
1	-2	-5	-3	1	0	0	0		8	-16	-80	-48	32	0	0	0		122	116	23	152	195	157	139	97
-2	1	1	-4	-1	0	0	0	→	-16	16	16	-128	-32	0	0	0	→	134	139	111	17	131	163	112	89
0	1	0	0	-2	0	0	0		0	16	0	0	-128	0	0	0		133	128	150	121	36	139	124	64
-1	0	1	0	0	-2	0	0		-16	0	32	0	0	-128	0	0		115	140	135	167	146	33	107	79
1	0	-1	0	0	0	-2	0		32	0	-64	0	0	0	-128	0		146	120	122	107	120	96	0	127
0	0	0	0	0	0	0	-1		0	0	0	0	0	0	0	-64		133	122	136	176	158	116	101	3

Y 0,1

123	-9	-8	-3	3	-1	1	0		984	-36	-64	-24	48	-16	32	0		130	154	125	134	145	139	156	5
-2	15	6	1	-3	0	0	0		-8	120	48	16	-48	0	0	0		148	134	153	152	143	129	0	163
4	5	-7	-2	0	0	-1	0		32	40	-112	-32	0	0	-64	0		103	138	147	135	130	10	149	152
-1	0	-1	5	1	0	0	0	→	-8	0	-16	160	32	0	0	0	→	108	85	159	109	1	195	119	147
0	0	0	1	-2	0	0	0		0	0	0	32	-128	0	0	0		139	96	109	33	182	144	110	139
-1	-1	0	0	-1	2	0	0		-16	-32	0	0	-64	128	0	0		133	82	18	164	170	153	89	153
0	0	0	0	0	0	-2	0		0	0	0	0	0	0	-128	0		97	7	144	168	160	126	138	142
0	0	0	0	0	0	-1	2		0	0	0	0	0	0	-64	128		12	145	153	189	172	112	118	153

Y 1,0

115	8	-3	-8	2	1	0	1		920	32	-24	-64	32	16	0	32		142	94	171	173	128	109	54	13
-2	20	-3	-3	2	0	1	0		-8	160	-24	-48	32	0	32	0		128	111	175	141	169	80	8	143
3	1	-9	0	1	0	1	0		24	8	-144	0	32	0	64	0		131	146	154	168	95	14	99	143
-2	-1	-1	4	1	0	0	0	→	-16	-16	-16	128	32	0	0	0	→	126	144	155	114	13	63	146	101
-2	1	0	0	-2	0	0	0		-32	16	0	0	-128	0	0	0		102	139	104	15	122	96	143	122
0	1	0	0	0	2	0	0		0	32	0	0	0	128	0	0		124	111	8	136	137	128	132	143
0	0	0	0	0	0	-1	-1		0	0	0	0	0	0	-64	-64		139	14	134	139	154	107	153	125
0	0	0	0	0	0	0	1		0	0	0	0	0	0	0	64		15	120	150	127	127	144	127	135

Y 1,1

131	6	1	-6	-1	1	0	0		1048	24	8	-48	-16	16	0	0		19	133	126	120	113	121	141	152
2	-16	-3	-1	1	0	0	0		8	-128	-24	-16	16	0	0	0		110	7	179	202	181	151	140	141
-8	-6	-9	1	1	0	0	0		-64	-48	-144	16	32	0	0	0		141	149	37	163	136	120	134	152
0	1	3	-4	-1	0	0	0	→	0	16	48	-128	-32	0	0	0	→	157	164	169	29	141	147	157	154
0	1	1	1	-2	0	0	0		0	16	32	32	-128	0	0	0		173	190	186	152	4	147	153	132
-3	-1	1	0	0	-2	0	0		-48	-32	32	0	0	-128	0	0		178	193	182	135	108	6	162	160
0	1	1	0	0	0	-2	0		0	32	64	0	0	0	-128	0		98	115	135	135	127	135	18	145
-1	0	0	0	0	0	0	-2		-32	0	0	0	0	0	0	-128		152	132	131	144	145	139	139	13

U 0,0

130	22	11	5	4	-3	1	0		1040	88	88	40	64	-48	32	0		131	185	168	178	185	159	168	154
25	-13	-2	5	-6	0	-1	0		100	-104	-16	80	-96	0	-32	0		168	149	130	104	149	193	156	150
14	0	-4	-6	0	0	0	0		112	0	-64	-96	0	0	0	0		168	124	87	57	121	187	154	150
-1	1	-2	0	1	0	0	0	→	-8	16	-32	0	32	0	0	0	→	175	154	86	80	111	110	113	110
1	3	-1	-2	0	0	-1	0		16	48	-32	-64	0	0	-64	0		182	140	86	97	114	79	72	101
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		187	94	89	91	116	105	54	133
0	-1	0	0	0	0	0	0		0	-32	0	0	0	0	0	0		193	122	138	123	114	86	41	143
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		159	167	181	174	121	46	61	160

U 1,0

143	31	17	-19	1	3	1	0		1144	124	136	-152	16	48	32	0		188	158	194	173	130	126	128	176
-10	7	7	-3	2	-1	1	0		-40	56	56	-48	32	-32	32	0		174	174	192	136	78	62	83	177
14	-6	-7	4	0	-1	0	0		112	-48	-112	64	0	-32	0	0		177	171	165	113	83	77	90	172
-2	1	1	-1	0	1	0	0	→	-16	16	16	-32	0	64	0	0	→	183	159	159	116	68	79	119	174
2	-1	0	0	0	0	0	0		32	-16	0	0	0	0	0	0		186	161	191	137	16	29	130	180
3	-1	-1	1	0	0	0	0		48	-32	-32	64	0	0	0	0		163	132	168	141	37	53	134	137
-1	0	0	0	0	0	0	0		-32	0	0	0	0	0	0	0		173	155	172	182	160	171	184	147
1	0	0	0	0	0	0	0		32	0	0	0	0	0	0	0		139	167	172	174	180	174	165	157

V 0,0

126	10	-12	-10	1	1	-1	-1		1008	40	-96	-80	16	16	-32	-32		115	114	90	106	93	105	117	113
1	0	0	-1	1	1	0	0		4	0	0	-16	16	32	0	0		108	125	139	171	128	104	112	116
-10	-3	4	4	0	-1	0	0		-80	-24	64	64	0	-32	0	0		104	129	169	215	151	106	112	118
-2	0	2	0	-1	0	0	0	→	-16	0	32	0	-32	0	0	0	→	107	129	164	203	142	110	120	117
-2	-1	1	1	0	0	0	0		-32	-16	32	32	0	0	0	0		104	139	162	180	126	113	124	114
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		97	151	165	170	126	114	118	113
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		99	142	133	145	130	117	106	112
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		108	122	84	108	130	120	98	110

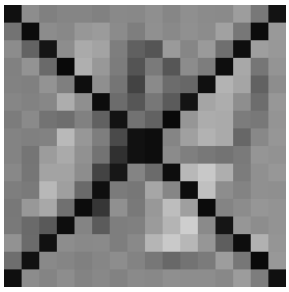
V 1,0

120	-10	-7	7	0	-1	-1	0		960	-40	-56	56	0	-16	-32	0		99	116	100	120	117	108	134	99
5	-2	-3	0	-1	0	-1	0		20	-16	-24	0	-16	0	-32	0		102	122	106	130	142	132	145	113
-6	2	3	-2	0	1	0	0		-48	16	48	-32	0	32	0	0		98	122	104	130	160	147	138	110
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		100	122	102	127	166	150	123	100
-1	1	0	0	0	0	0	0	→	-16	16	0	0	0	0	0	0	→	109	123	107	128	167	153	122	106
-1	0	0	0	0	0	0	0		-16	0	0	0	0	0	0	0		113	114	106	122	150	144	123	116
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		116	103	102	110	119	121	115	114
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		126	103	105	106	100	104	108	110

We now have all the data required to reconstruct the downsampled image, which is in YUV format as shown below.

Y

7	142	145	128	133	136	140	127	130	154	125	134	145	139	156	5
131	8	122	139	129	151	134	131	148	134	153	152	143	129	0	163
122	116	23	152	195	157	139	97	103	138	147	135	130	10	149	152
134	139	111	17	131	163	112	89	108	85	159	109	1	195	119	147
133	128	150	121	36	139	124	64	139	96	109	33	182	144	110	139
115	140	135	167	146	33	107	79	133	82	18	164	170	153	89	153
146	120	122	107	120	96	0	127	97	7	144	168	160	126	138	142
133	122	136	176	158	116	101	3	12	145	153	189	172	112	118	153
142	94	171	173	128	109	54	13	19	133	126	120	113	121	141	152
128	111	175	141	169	80	8	143	110	7	179	202	181	151	140	141
131	146	154	168	95	14	99	143	141	149	37	163	136	120	134	152
126	144	155	114	13	63	146	101	157	164	169	29	141	147	157	154
102	139	104	15	122	96	143	122	173	190	186	152	4	147	153	132
124	111	8	136	137	128	132	143	178	193	182	135	108	6	162	160
139	14	134	139	154	107	153	125	98	115	135	135	127	135	18	145
15	120	150	127	127	144	127	135	152	132	131	144	145	139	139	13

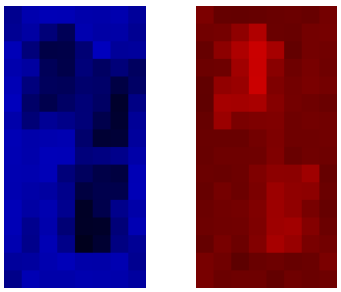


131	185	168	178	185	159	168	154
168	149	130	104	149	193	156	150
168	124	87	57	121	187	154	150
175	154	86	80	111	110	113	110
182	140	86	97	114	79	72	101
187	94	89	91	116	105	54	133
193	122	138	123	114	86	41	143
159	167	181	174	121	46	61	160
188	158	194	173	130	126	128	176
174	174	192	136	78	62	83	177
177	171	165	113	83	77	90	172
183	159	159	116	68	79	119	174
186	161	191	137	16	29	130	180
163	132	168	141	37	53	134	137
173	155	172	182	160	171	184	147
139	167	172	174	180	174	165	157

U

115	114	90	106	93	105	117	113
108	125	139	171	128	104	112	116
104	129	169	215	151	106	112	118
107	129	164	203	142	110	120	117
104	139	162	180	126	113	124	114
97	151	165	170	126	114	118	113
99	142	133	145	130	117	106	112
108	122	84	108	130	120	98	110
99	116	100	120	117	108	134	99
102	122	106	130	142	132	145	113
98	122	104	130	160	147	138	110
100	122	102	127	166	150	123	100
109	123	107	128	167	153	122	106
113	114	106	122	150	144	123	116
116	103	102	110	119	121	115	114
126	103	105	106	100	104	108	110

V



U V

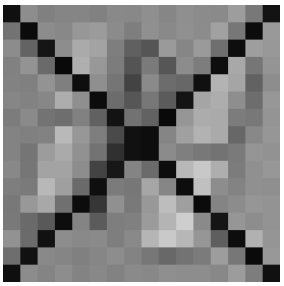
Figure 8(a) – Decompressed downsampled YUV planes

We now apply horizontal interpolation to the U and V planes to recover the missing pixels and restore them to the full image width. The interpolation proceeds row by row and is done with an interpolation filter known as a finite impulse response (FIR) filter, which uses a weighted sum of a finite number of samples to produce each reconstructed sample. In our case, that finite number is 6, so the filter is called a 6-tap filter. We must obtain from the n samples in each row of the U and V planes 2n samples for the corresponding row of the reconstructed plane (for this particular image of size 16x16, n will equal 8). Note, for an image of size 320x240, n equals 320/2=160. For a given row **U**[0..(n-1)] (and analogously for **V**), and the corresponding row **U'**[0..(2n-1)] from the corresponding reconstructed plane, the equation for obtaining the sample **U'**[j] is:

$$\mathbf{U}'[j] = \begin{cases} \mathbf{U}[\frac{j}{2}] & j = even \\ (int)\frac{1}{256}\left(21\mathbf{U}[\frac{j-5}{2}] - 52\mathbf{U}[\frac{j-3}{2}] + 159\mathbf{U}[\frac{j-1}{2}] + 159\mathbf{U}[\frac{j+1}{2}] - 52\mathbf{U}[\frac{j+3}{2}] + 21\mathbf{U}[\frac{j+5}{2}] + 128\right) & j = odd \end{cases} \quad (7)$$

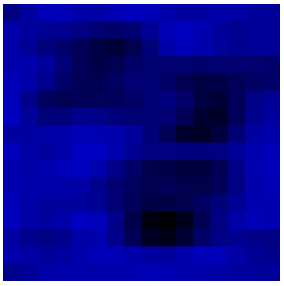
The ends of the row (where j-5, j-3, j-1, j+1, j+3, j+5 would land outside the range 0..2n-1) are handled by replicating the respective border sample, i.e. **U**[-2] = **U**[-1] = **U**[0], and **U**[n] = **U**[n+1] = **U**[n+2] = **U**[n-1]. Note again the use of integer operations. These coefficients are borrowed from the ISO/IEC 13818-2 document on MPEG2 digital video decoding. If we apply this filter to the U and V planes, we obtain:

7	142	145	128	133	136	140	127	130	154	125	134	145	139	156	5
131	8	122	139	129	151	134	131	148	134	153	152	143	129	0	163
122	116	23	152	195	157	139	97	103	138	147	135	130	10	149	152
134	139	111	17	131	163	112	89	108	85	159	109	1	195	119	147
133	128	150	121	36	139	124	64	139	96	109	33	182	144	110	139
115	140	135	167	146	33	107	79	133	82	18	164	170	153	89	153
146	120	122	107	120	96	0	127	97	7	144	168	160	126	138	142
133	122	136	176	158	116	101	3	12	145	153	189	172	112	118	153
142	94	171	173	128	109	54	13	19	133	126	120	113	121	141	152
128	111	175	141	169	80	8	143	110	7	179	202	181	151	140	141
131	146	154	168	95	14	99	143	141	149	37	163	136	120	134	152
126	144	155	114	13	63	146	101	157	164	169	29	141	147	157	154
102	139	104	15	122	96	143	122	173	190	186	152	4	147	153	132
124	111	8	136	137	128	132	143	178	193	182	135	108	6	162	160
139	14	134	139	154	107	153	125	98	115	135	135	127	135	18	145
15	120	150	127	127	144	127	135	152	132	131	144	145	139	139	13



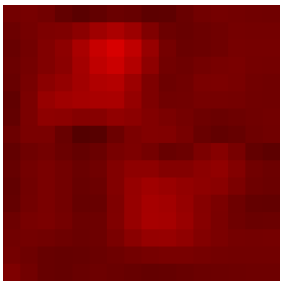
Y

131	161	185	182	168	164	178	188	185	170	159	161	168	164	154	152
168	159	149	144	130	114	104	117	149	183	193	177	156	145	150	152
168	148	124	109	87	69	57	78	121	168	187	174	154	143	150	152
175	172	154	121	86	73	80	101	111	114	110	109	113	112	110	109
182	168	140	108	86	83	97	115	114	101	79	66	72	89	101	105
187	141	94	82	89	93	91	101	116	126	105	67	54	88	133	147
193	154	122	122	138	137	123	115	114	114	86	48	41	89	143	159
159	161	167	171	181	179	174	156	121	84	46	37	61	118	160	171
188	167	158	171	194	195	173	147	130	128	126	124	128	153	176	182
174	167	174	185	192	172	136	102	78	73	62	64	83	134	177	187
177	170	171	171	165	142	113	94	83	86	77	75	90	133	172	181
183	167	159	157	159	146	116	89	68	71	79	98	119	150	174	177
186	165	161	170	191	185	137	74	16	4	29	85	130	166	180	178
163	141	132	141	168	175	141	87	37	25	53	104	134	144	137	131
173	163	155	158	172	184	182	171	160	157	171	185	184	166	147	141
139	153	167	173	172	170	174	177	180	178	174	169	165	160	157	157



U

115	119	114	99	90	98	106	103	93	94	105	114	117	115	113	112
108	117	125	127	139	159	171	156	128	108	104	108	112	117	116	116
104	115	129	141	169	199	215	191	151	117	106	108	112	119	118	118
107	117	129	139	164	191	203	179	142	114	110	116	120	122	117	116
104	120	139	148	162	176	180	156	126	109	113	123	124	121	114	112
97	123	151	160	165	169	170	149	126	113	114	119	118	117	113	112
99	123	142	140	133	135	145	140	130	123	117	110	106	109	112	114
108	122	122	104	84	87	108	124	130	129	120	105	98	102	110	113
99	111	116	107	100	106	120	125	117	104	108	124	134	120	99	93
102	116	122	115	106	112	130	142	142	132	132	140	145	131	113	108
98	114	122	115	104	108	130	150	160	154	147	142	138	124	110	107
100	115	122	115	102	104	127	151	166	162	150	134	123	110	100	99
109	120	123	117	107	109	128	151	167	165	153	135	122	111	106	107
113	116	114	110	106	109	122	138	150	151	144	131	123	117	116	117
116	110	103	101	102	106	110	115	119	121	121	118	115	114	114	114
126	114	103	101	105	109	106	103	100	101	104	107	108	109	110	110



V

Figure 8(b) – Upsampled YUV data

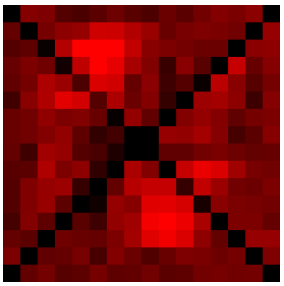
Having restored the full Y, U and V planes, the only remaining step is the colourspace conversion from YUV back to RGB. As discussed above (Equation 1), the conversion from RGB to YUV is in the form of a matrix multiplication. In essence, the RGB values of a given pixel are viewed as a vector in 3-space, and the matrix multiplication is a change of basis yielding another vector in 3-space. Since this is a linear transformation and the determinant of the matrix is non-zero, we may find its inverse, which will provide the transformation back to the RGB basis from the YUV basis, which may be applied to each pixel.

Accounting for the vector addition to shift the basis, the inverse conversion of the one above is:

$$\begin{bmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{bmatrix} = (int)\frac{1}{65536}\left(\begin{bmatrix} 76284 & 0 & 104595 \\ 76284 & -25624 & -53281 \\ 76284 & 132251 & 0 \end{bmatrix}\left(\begin{bmatrix} \mathbf{Y} \\ \mathbf{U} \\ \mathbf{V} \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}\right)\right) \tag{8}$$

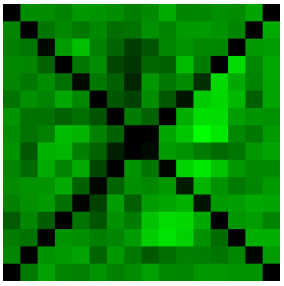
As in the case of IDCT, an identical clipping function is used for addressing the rounding and overflow errors.

0	132	127	84	75	91	109	89	76	106	90	115	132	122	139	0
101	0	118	141	149	206	205	178	153	105	121	126	122	113	0	151
85	95	9	179	255	255	255	194	137	124	117	106	107	0	138	142
103	125	112	18	191	255	231	166	129	57	137	89	0	198	102	133
97	117	173	154	77	219	208	100	139	62	84	11	186	137	87	117
65	136	175	226	210	85	172	106	132	52	0	157	163	141	61	133
105	113	145	125	129	104	8	148	97	0	131	148	132	97	116	124
104	113	130	147	95	50	67	0	0	151	146	164	133	70	90	135
100	63	161	149	85	73	31	0	0	97	96	114	122	109	99	102
88	91	175	124	142	48	0	170	131	0	196	235	219	161	120	113
85	128	151	156	53	0	99	182	196	196	54	193	155	114	108	124
83	128	152	93	0	16	149	135	224	226	213	24	137	123	119	114
69	130	94	0	89	62	147	160	244	255	237	169	0	125	124	101
101	91	0	110	105	100	125	163	223	242	218	143	99	0	150	150
124	0	97	100	119	70	130	106	81	104	127	122	108	116	0	127
0	98	116	86	92	118	94	98	113	91	95	115	118	112	114	0



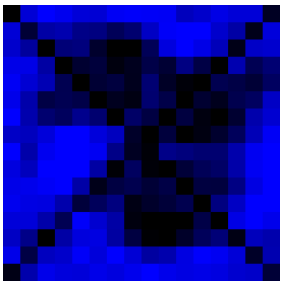
R

0	141	139	132	151	149	142	126	138	171	133	135	143	139	164	0
134	0	117	137	121	137	111	115	145	132	153	155	149	133	0	171
127	119	8	155	191	129	100	62	85	135	147	136	135	0	154	157
136	134	99	0	121	141	69	54	102	97	188	125	0	219	135	169
134	121	142	113	12	121	95	38	150	119	139	48	218	169	131	165
117	143	133	167	136	0	86	66	142	89	22	203	216	184	95	165
149	114	114	98	113	83	0	124	98	0	174	222	219	158	149	145
140	115	129	188	180	129	97	0	0	166	198	255	232	136	120	154
146	89	178	183	127	99	33	0	11	155	145	125	108	118	150	165
133	105	171	133	170	70	0	146	117	7	212	231	195	152	137	138
139	146	148	170	97	8	100	143	137	150	28	180	146	122	134	154
129	144	154	113	5	67	156	95	156	166	179	21	153	158	168	165
92	135	93	0	115	86	144	125	194	220	216	169	0	151	157	132
124	115	0	149	143	127	134	155	206	227	209	145	108	0	176	175
135	0	147	153	164	101	152	120	90	109	127	124	117	135	6	156
0	122	161	133	130	148	129	139	160	137	135	150	151	146	146	0



G

0	213	255	239	216	212	245	250	247	245	189	203	230	215	215	35
214	53	165	175	135	128	88	111	196	248	255	255	204	165	25	219
204	156	0	119	125	45	0	0	87	222	255	231	185	23	199	206
232	231	163	0	49	60	14	30	72	52	130	69	0	176	83	114
245	211	180	81	0	52	63	29	114	38	9	0	80	70	54	96
234	170	69	82	72	0	31	18	111	72	0	49	29	78	95	197
255	173	111	93	141	111	0	102	66	0	64	15	0	49	172	209
198	189	218	255	255	219	191	41	0	61	0	17	46	91	183	246
255	169	240	255	255	243	135	34	7	136	124	112	112	172	242	255
223	189	255	255	255	163	6	95	8	0	56	87	101	169	243	255
232	236	247	255	166	25	66	79	54	70	0	64	62	131	226	255
239	227	224	172	59	91	127	20	43	57	79	0	127	196	255	255
217	217	169	83	250	208	165	14	0	0	0	71	0	229	255	235
196	136	0	165	221	225	161	65	4	0	41	90	119	20	188	173
233	68	191	203	249	218	255	213	160	173	225	253	242	215	40	176
21	171	234	220	217	233	222	237	255	235	226	231	224	207	201	55



B

Figure 9 – Colourspace conversion result: R, G and B planes

The final image, as shown on the left (a) in Figure 10, requires 16x16x3 = 768 bytes (plus a header of 13 bytes), whereas the compressed image requires only 240 bytes, giving a compression ratio of ~3x. The original image appears on the right (c), a comparison which shows the reduction in quality due to information lost during the compression/decompression process. The center image (b) shows the decompressed image for the case when $\mathbf{Q}_2$ from Equation 3 is used (instead of $\mathbf{Q}_0$ as is the case for this example, image (a)), which, during compression, retains more frequency information for use during reconstruction. The use of this matrix clearly improves the quality but reduces the compression. This discussion is summarized in the signal-to-noise ratio (SNR) values and file sizes given for both cases in Table 1.

(a) (b) (c)

Figure 10 – Final decompressed images (using Q_0, using Q_2 and original)

	Q_0 image	Q_2 image	Original image
SNR (dB)	18.6	20.7	∞
File size (bytes)	240	423	781
Compression (x)	3.25	1.85	1

Table 1 – Quality / compression tradeoff for Q_0 and Q_2

Equation 9 shows the equation for calculating the signal-to-noise ratio, where N is the number of pixels. The signal-to-noise ratio is 20 times $\log_{10}$ of 255 divided by the root mean squared error between each pixel of the image and each pixel in the reference.

$$SNR = 20log_{10}\left(\frac{255}{\sqrt{\frac{error}{N}}}\right) \quad \text{where } error = \sum_N (reference\_pixel - image\_pixel)^2 \tag{9}$$

This part of the document has provided the barebones of image compression (sufficient to work on the project). Further information on image compression can be found in the following references.

[1] JPEG: Still Image Data Compression Standard, by William B., Pennebaker, Joan L., Mitchell, Springer; 1 edition (2006), ISBN: 0442012721

[2] Digital Video and HDTV Algorithms and Interfaces, by Charles Poynton, Publisher: Morgan Kaufmann (2003), ISBN: 1558607927

[3] Video Codec Design: Developing Image and Video Compression Systems, by Iain Richardson, Publisher: John Wiley & Sons (2002), ISBN: 0471485535

[4] Introduction to Data Compression, Third Edition, by Khalid Sayood, Publisher: Morgan Kaufmann (2005), ISBN: 012620862X

[5] Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards, by Yun Q. Shi, Huifang Sun, Publisher: CRC Press (1999), ISBN: 0849334918

[6] Digital Video Compression, by Peter Symes, Publisher: McGraw-Hill (2003), ISBN: 0071424873

Project Guidelines

The design task of implementing the overall encoder has been partitioned into three milestones, which should be completed within one week (with a one-week margin overall for the entire project). As part of the validation methodology, implementation will proceed from the RGB image forward to the compressed output. The initial input of the RGB image is loaded into the SRAM, and as each successive milestone is completed, its output is written to the SRAM. This allows a progress check at each milestone, providing the framework for incremental integration of each newly designed module into the overall system.

Milestone 1 – Color space conversion and downsampling

Figure 11 shows the implementation and validation plan for milestone 1. In this first week, colourspace conversion and downsampling will be tackled as indicated in the portion of the figure, which shows the flow from Figure 1. This will also require some modifications to the existing state machines to grant the encoder access to the external memory; in fact, all the shaded portions of the figure will have to be built or modified. The SRAM Controller and UART SRAM Interface shown in the figure have been provided in lab 5. Using the software model provided, intermediate data can be obtained thus enabling the flow to be "tapped". Specifically, the command "Project –encode file_1 0 file_2 –debug 1" will encode file_1.ppm using quantization matrix 0, producing file_2.mic and downsampled YUV data in the file file_2.d1e. This data is applied to the module being designed through the testbench to confirm proper operation. Once the circuit has been validated through the testbench, the same data may be uploaded to the DE2 board via UART to validate further the interfaces working in real-time. In this way, the encoding software provides a snapshot after the first part of the encoding task, and the intermediate data coming either from a testbench or from the board can be compared with this snapshot to determine if the circuit is operating correctly. In the case of the testbench, detailed feedback can be provided, which aids in tracing implementation errors, while the board provides less feedback but a full practical operational test.

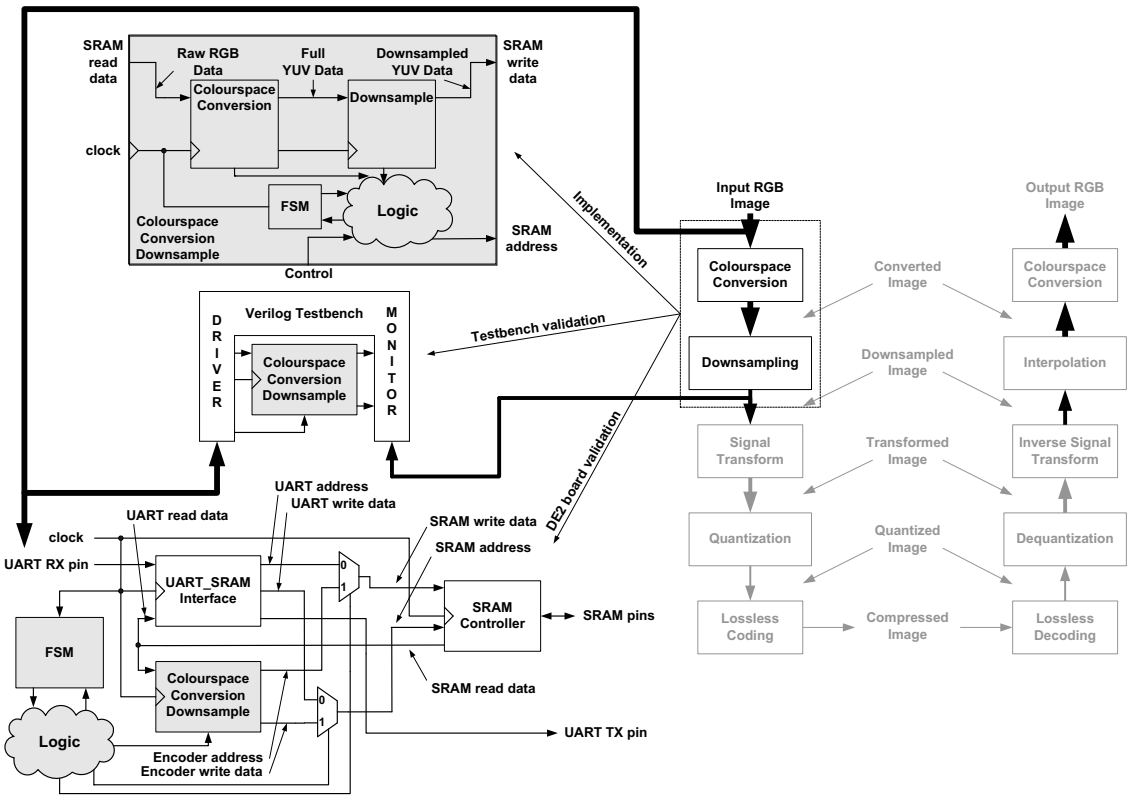


Figure 11 – Overview of colourspace conversion and downsampling implementation (milestone 1)

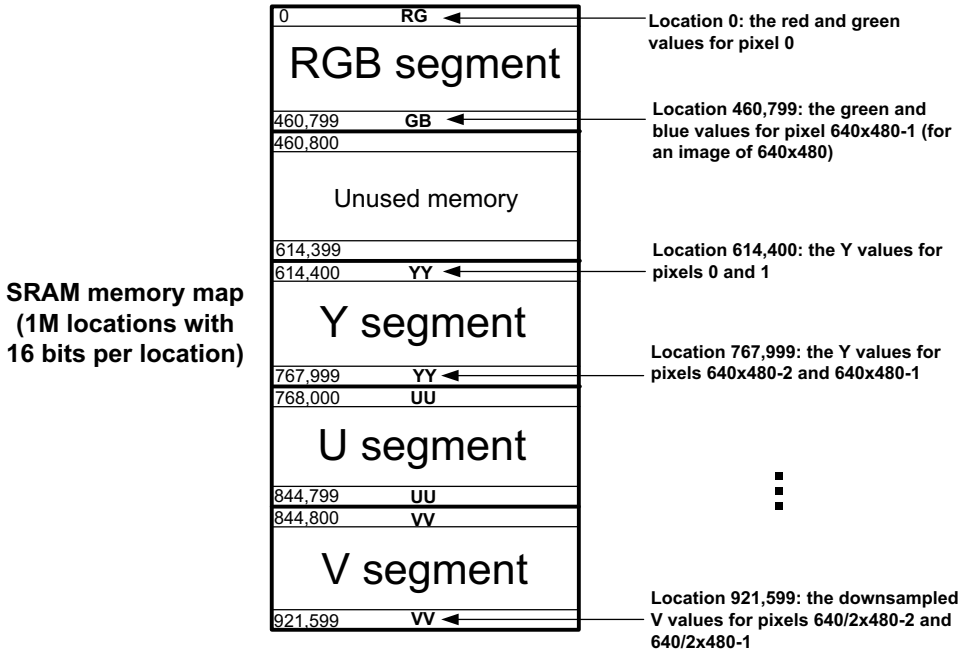


Figure 12 – Memory layout for milestone 1 (assuming an image of 640x480)

Figure 12 shows the memory layout which is used for milestone 1. The first portion of the memory holds the raw RGB image stored in "RGBRGB…" format (as in lab 5). This is where the compressor will read the image from. The downsampled YUV data resides in the latter part of the memory. This data will form the input for the subsequent milestone; however, for this milestone, the YUV data will be read from this section by the UART SRAM interface and transmitted back to the PC to perform validation.

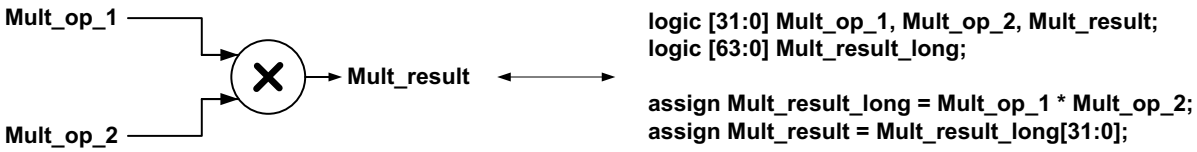


Figure 13 – Instantiation of a single 32-bit multiplier

Figure 13 shows how a single 32-bit integer multiplier may be instantiated in your design. As the forthcoming sections will elaborate, to end up with a resource-efficient implementation, multipliers (which are costly in terms of resources) must be shared. Specifically, for the encoding project, you are constrained to use four 32-bit multipliers that must be time-shared for colourspace conversion/downsampling and DCT. Thus, there will be four instantiations of the form for Figure 13 that are time-shared between milestones 1 and 2. In the next paragraphs, some architectural examples (not directly applicable to this project instance) are provided regarding how the multipliers can be shared. Note that constant coefficient multiplication is not permitted, except for the multiplication by 256 for downsampling and multiplication by 640 for address generation in milestone 2. This will also be clarified during project meetings (when in doubt about key design decisions, ask for clarifications).

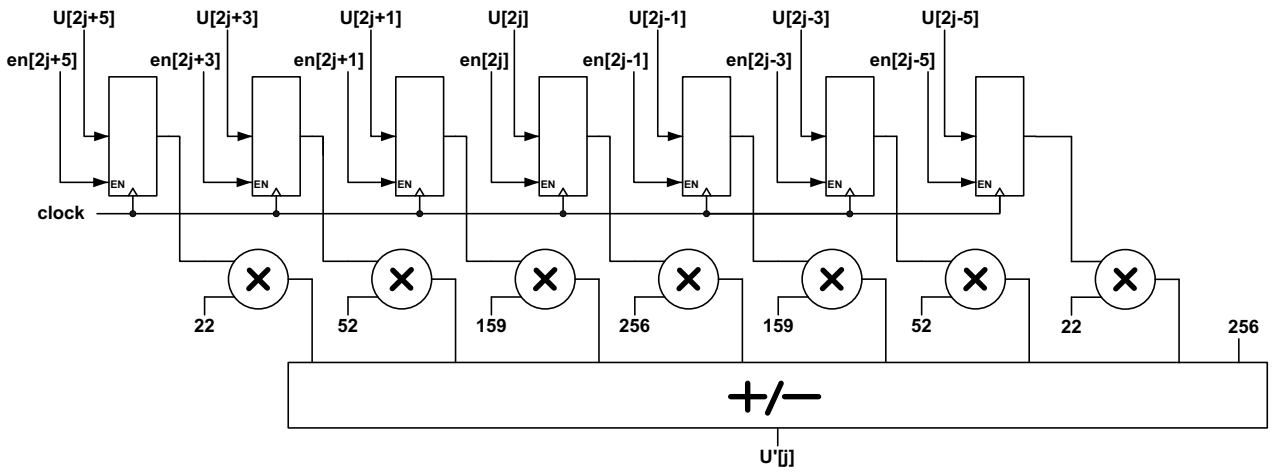


Figure 14 – Resource ineffective FIR filter implementation

Figure 14 shows a straightforward implementation of an FIR filter, where the samples are stored in a shift structure. In this case, each sample undergoes a constant multiplication, and the results are all combined using an adder tree. Although this is attractive from the throughput perspective, as one filtered sample may be obtained every clock cycle, it is costly from the resource standpoint, especially considering that one such filter would be required for each component, which must be downsampled (U and V).

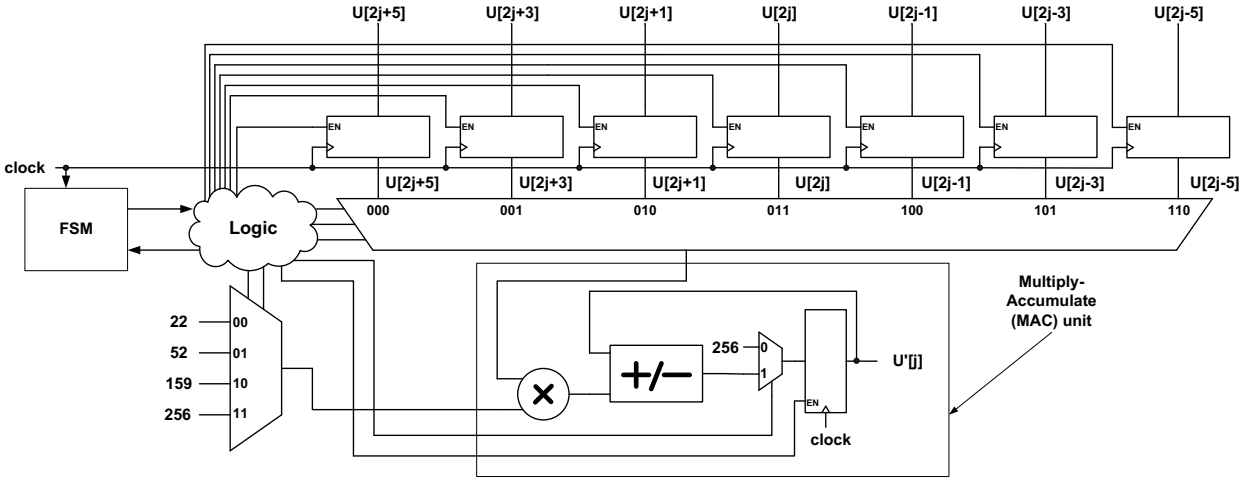


Figure 15 – Improved FIR filter implementation

In Figure 15, we see how a single multiplier can be shared to accomplish the same end as Figure 14, at the expense however of throughput. In this case, the boxed portion is known as a *Multiply-Accumulate* (MAC) unit, which works by always multiplying the two inputs, and adding the result to the register (known as the accumulation register or accumulator), and then storing that result back in the accumulator, to become an operand in the next clock cycle. The FSM guides the selection of both the sample and constant factor, as well as initialization, which in this case is to account for the +256 in the downsampling equation (last line) of Equation 1, also shown in Figure 14. Using this architecture, only one multiplier is required, however, each reconstructed sample requires 7 clock cycles to produce instead of 1 as in the previous case. This is an example of a typical architectural tradeoff between resource usage and throughput.

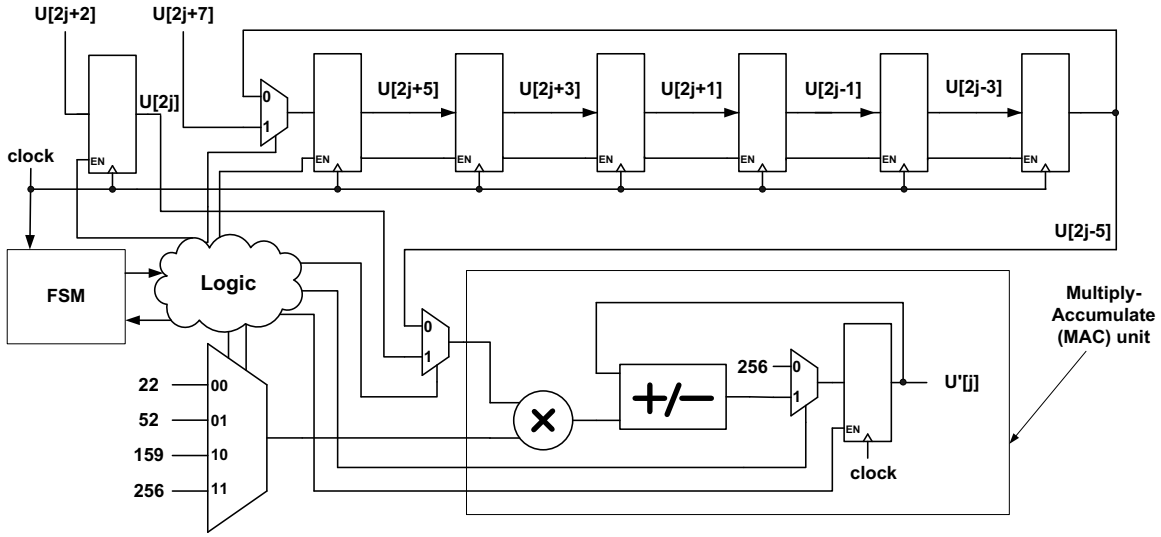


Figure 16 – Resource-efficient FIR filter implementation

Finally, Figure 16 shows a cost-effective FIR implementation resembling Figure 15, but with the important distinction that the multiplexer that selects between the different samples (as set up by the FSM) is eliminated. The same result can be obtained by leveraging the shift structure that is already present due to the reuse of the odd samples in calculating adjacent reconstructed samples. Now, instead of the FSM setting up select lines on a mux to obtain a desired sample, the calculation proceeds by holding the shift for the first clock cycle (during initialization of the accumulator to 256), and during the next 6 clock cycles, the odd samples are circularly shifted, each time the final one being accumulated. In a further (and final) clock cycle, the center (even) sample is accumulated into the result. At the end of these 7 clock cycles, the data is in its original order, and the accumulation is complete; thus, a new reconstructed sample has been produced. Although identical to Figure 15 in throughput, eliminating the large sample selector mux makes this design substantially more efficient.

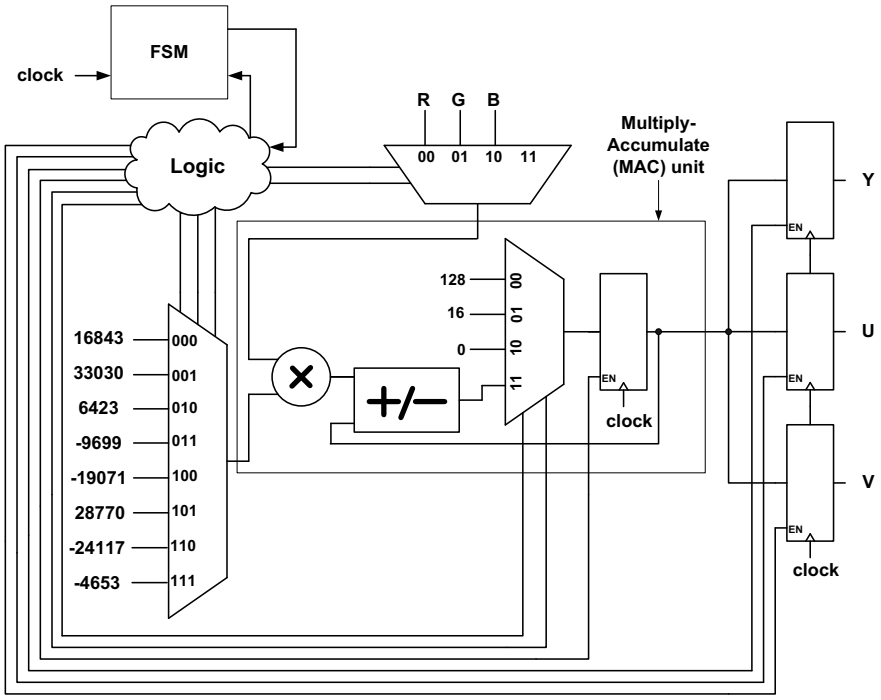


Figure 17 – Resource-efficient colourspace conversion implementation

In the same way that the MAC concept may be used to share multiplications for the FIR filters, it may be used for colourspace conversion. In this case, the use of the mux between samples is sufficient since there are fewer samples from which to select, and the logic for the shift structure is not already in place as was the case for the FIR filter.

State code / Clock cycle	0	1	2	3	4	…
SRAM_address						
SRAM_read_data						
SRAM_write_data						
SRAM_write_enable						
U[2j+7]						
U[2j+2]						
U'[j]						
Y						
U						
V						
R						
G						
B						
…						
…						

Table 2 – State table for milestone 1; a starting point

As emphasized during lab 5, state tables are a powerful way of capturing and synthesizing information about a state machine. Table 2 provides a good starting point for a state table that can be used for milestone 1. Although the report will not require a state table, we strongly urge you to use it throughout development, verification, and debugging at this stage of your technical development.

Milestone 2 – DCT

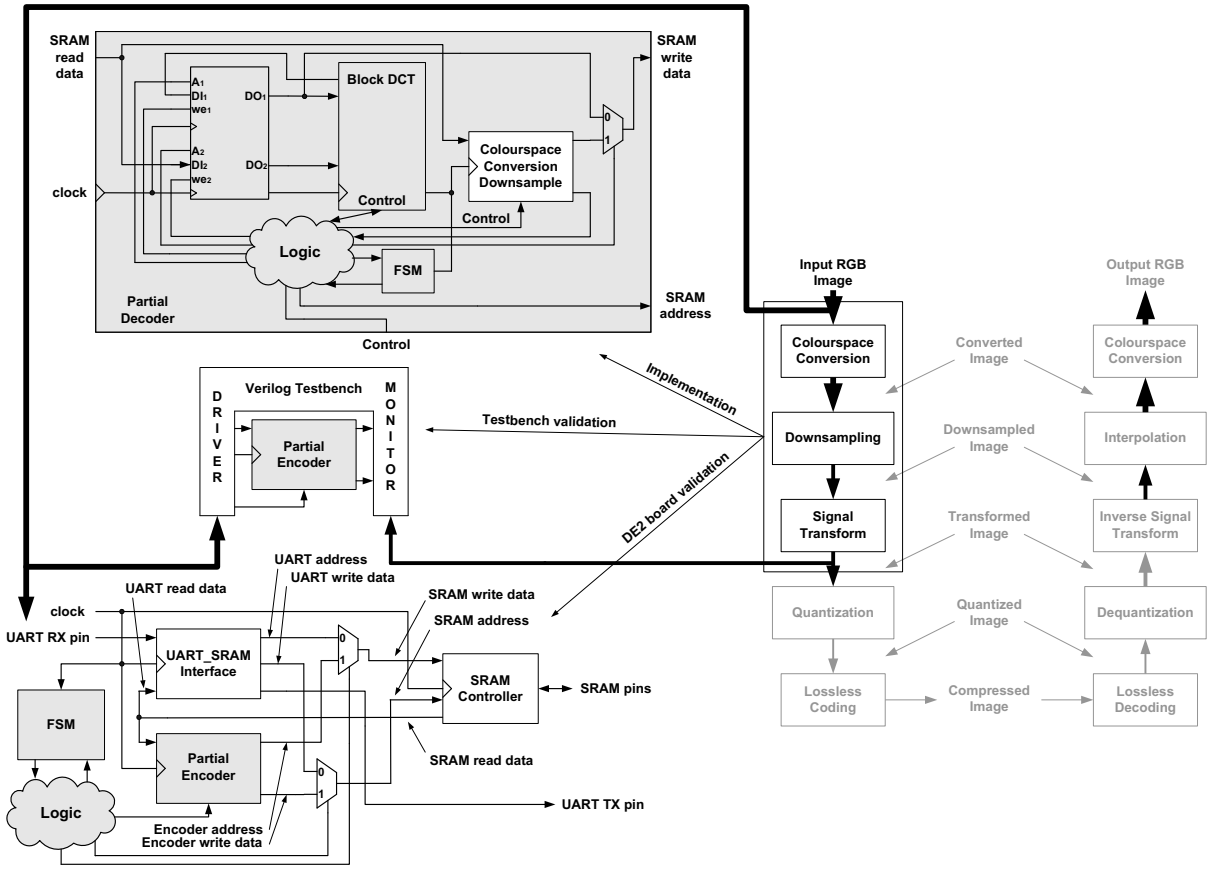


Figure 18 – Overview of the DCT implementation (milestone 2)

Figure 18 shows the overview for milestone 2, which is the implementation of DCT. As before, the shaded portions of the figure indicate what needs to be built/modified. In particular, note that the FSM and the logic in the top state machine (bottom portion of the figure) do not need to be modified since in the previous milestone, you have made provision for the encoder circuitry (all of which will eventually reside in the block currently labelled “Partial Decoder”) to access the SRAM. In a similar way as before then, the command “Project –encode file_1 0 file_2 –debug 2” will produce data from the raw image file_1, stored in file_2.d2e, which will contain the data blocks just after DCT but before quantization. This data is used by the testbench for validation and can be sent to the board with a working design to get confirmation that the colourspace conversion, downsampling and DCT are working together.

In Figure 19, the memory map for milestone 2 is provided, which is similar to that for milestone 1. The main distinction comes from the fact that where before the output data was downsampled YUV data in the segments provided, the output data is in the form of post-DCT blocks. The same segments apply, all the Y blocks will be slotted into the Y segment, etc. however, the way the data is accessed is modified. In this case, we want to fetch an entire 8x8 block of pre-DCT data, perform DCT and write it back to the coefficient memory space. To determine the addressing for writing post-DCT samples back to the memory, observe Figure 8(a) and note that the block from block row 0, block column 0 of the Y plane (Y 0,0) contains data from row 0, row1, … row 7 of the image, all in columns 0 to 7 of the image. Extending this to the case of a 640x480 image, and noting that we have one sample per address, our addressing for block 0 should be (0+): 0 … 7, …, 4480, …, 4487 (as shown in Figure 20). For the second block of the first block row (Y 0,1), we have (0+) 8, … 15, …, 4488, … 4495. Once all the Y blocks have been processed, we proceed to U and V, with the important distinction that these two planes are downsampled and thus contain half the number of columns which Y contained. Thus U 0,0 would reside in addresses (307,200+): 0, … 7, …, 2240, …, 2247.

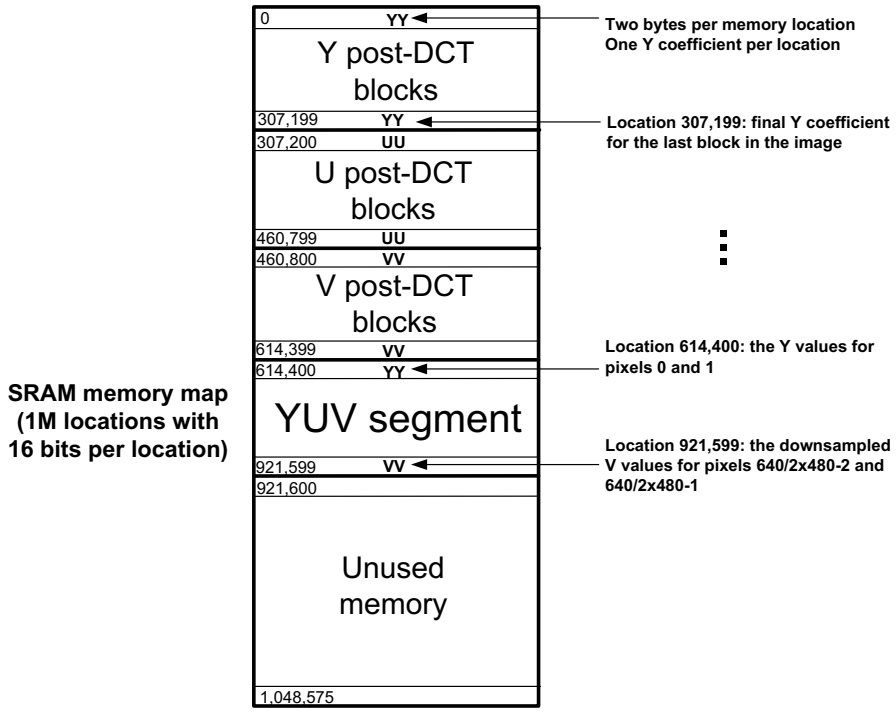


Figure 19 – Memory layout for milestone 2

In addition to addressing for writing back the samples (which can be reused in the full decoder implementation), the fetch addressing for passing post-DCT over UART must be developed for the purpose of validation and can also be used again in the final implementation. The pre-DCT data is stored starting at location 614,400 of the SRAM (as in Milestone 1). To read block Y 0,0, we use addresses 614,400, …, 614,403, …, 616,640, …, 616,643 (as in Figure 20) and for block Y 0,1, we use 614,404, …, 614,407, …, 616,644, …, 616,647. For U 0,0 we use (614,400+307,200+) 0, …, 3, …,2240, …, 2243. For V 0,0 we use (614,400+460,800+) 0, …, 3, …, 2240, …, 2243. As a final note, in order to circumvent the need for a hardware multiplier (resulting from the variable size of the image), rows of blocks can always begin at the same address, and constant-coefficient multiplication with 640 is used for addressing. For example, for an image of 320x240, we can write the second row of post-DCT block Y 0,0 at address 640 and addresses 320 to 639 will not be used. Alternatively, one can "compact" the memory segment for post-DCT data and rely on an additional multiplier for generating the address (not required for this project instance).

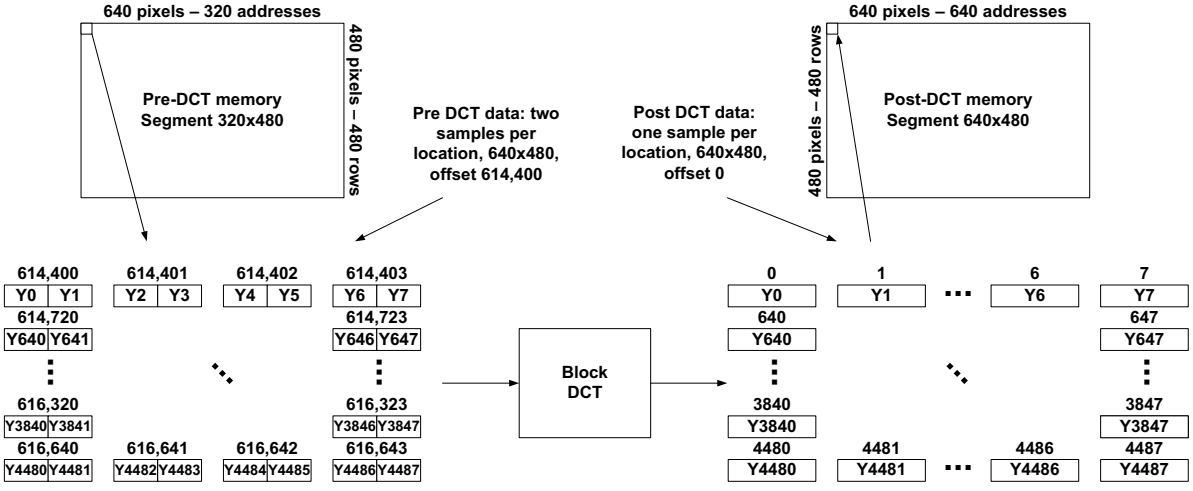


Figure 20 – Reading pre-DCT stimuli and writing back post-DCT coefficients to the external SRAM

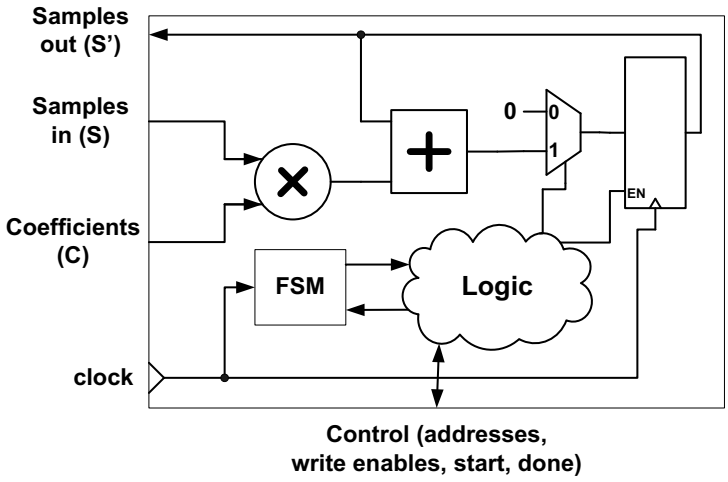


Figure 21 – Block DCT MAC unit

Figure 21 shows a MAC unit which may be used for calculating part of the DCT on a single block of samples. Recall that the DCT is simply a pair of matrix multiplications. Since a matrix multiplication with a constant coefficient matrix is nothing more than the sum of a number of products of the samples with constants, the MAC unit will meet the architectural need. The role of the FSM is to initialize the accumulator at the appropriate times, and to set up the addresses properly to obtain the correct sample and coefficient in each clock cycle. Using such a unit, the pre- and post matrix multiplication can be done concurrently (since there are four multipliers). It should be noted that, as shown in Figure 18 both the samples and the coefficients are coming from dual port memories. The onus is on you to determine a suitable memory layout for the dual-port RAMs. Note the fixed-point cosine coefficients can be obtained by using the reference software model as follows: "Project –encode file_1 0 file_2 –debug 3". The fixed point values will be written in file_2.d3e and you must include them in an "HEX" file used for dual-port memory initialization. You are encouraged to use a state table such as the following one during your development and validation.

State code / Clock cycle	0	1	2	3	4	…
SRAM_address						
SRAM_read_data						
SRAM_write_data						
SRAM_write_enable						
A1						
DI1						
we1						
DO1						
A2						
DI2						
we2						
DO2						
S						
S'						
C						
…						

Table 3 – State table for milestone 2; a starting point

Milestone 3 – Quantization and lossless coding

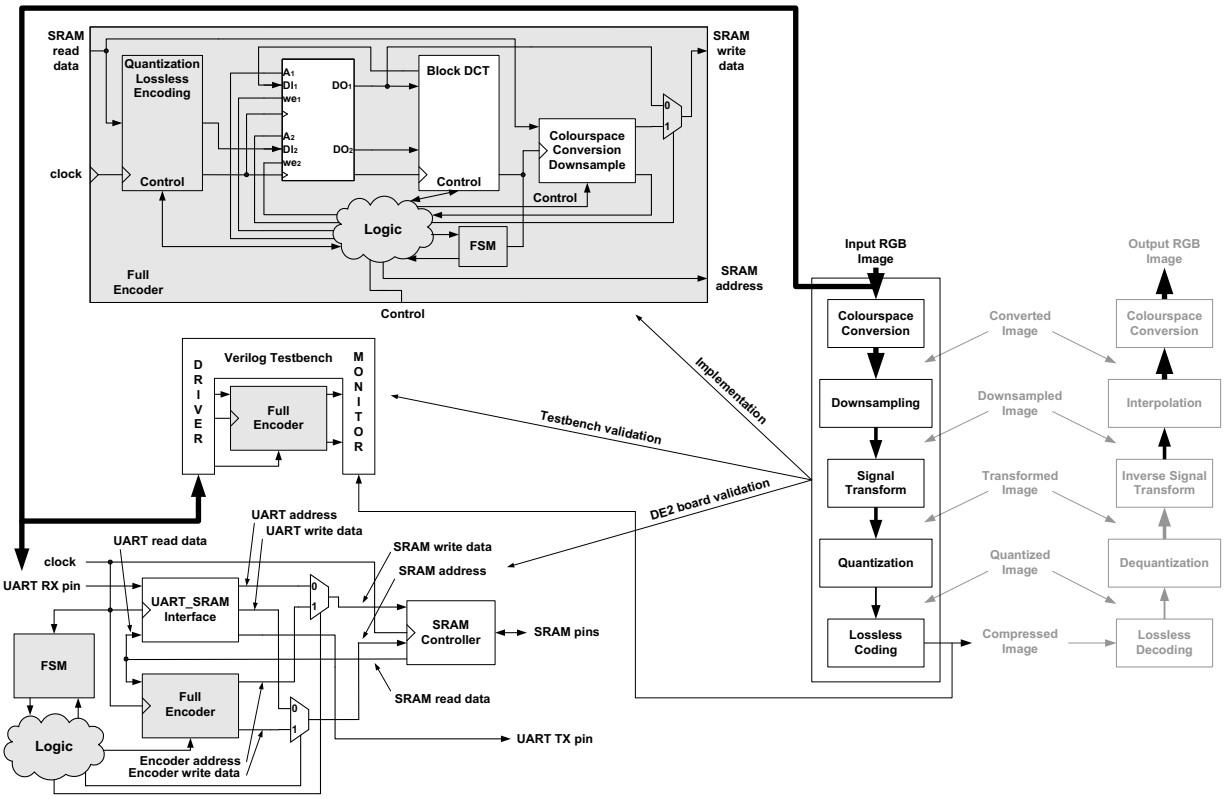


Figure 22 – Quantization, lossless coding, and complete integration (milestone 3)

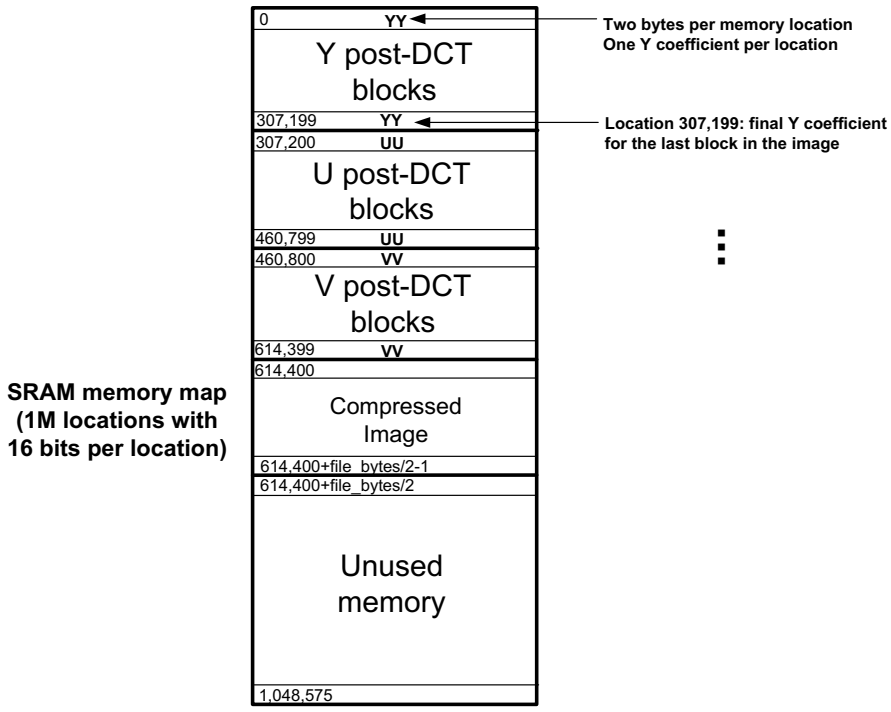


Figure 23 – Memory layout for milestone 3

Finally, once colourspace conversion, downsampling, and DCT are complete, we can move on to lossless encoding, where the compressed bitstream is actually produced based on the raw blocks quantized from the DCT output. Once this milestone is complete, you will have a working image encoder.

The memory layout for the completed project and, therefore, for milestone 3 is shown in Figure 23. As you can see, the compressed file is written starting at offset 614,400, and space is reserved for up to 640x480 image data starting from offset 0 of the memory. Lossless encoding proceeds by extracting coefficients from the YUV post-DCT data region, quantizing and converting them to coded form.

The basic encoding process is related to Figure 7 at the beginning of the image decompression example. This flow chart can be used as a starting point to construct a state table for guiding your state machine implementation. The main challenge of lossless encoding is the implementation of circuitry that assembles the variable length codes into full 16-bit words to be written to the SRAM. A custom FSM is required to drive the SRAM address and to increment it only if a full word has been assembled. Note that quantization is necessary before passing coefficients to the lossless encoder. Due to the special structure of the possible quantization matrices (as in Equation 3), which contain only powers of two, a barrel shifter-like structure can be used to implement the quantization (the type of the quantization matrix must also be used as an input to this barrel shifter). Furthermore, note the scan sequence of Figure 4 must be implemented so that the post-DCT coefficients are written to the bitstream in the correct block-scanned order.

ENJOY!

And one more thing … even for the largest image supported (640 x 480), the entire encoding process should not take more than 50 ms when using the 50 MHz clock for the datapath logic with 4 multipliers in total. Besides, while satisfying these latency/clock speed constraints, you should not waste hardware resources in the FPGA, such as multipliers, embedded memories and logic elements.