COE3DQ5 Project Description 2024
Hardware Implementation of an Image Decompressor

1 Introduction

1.1 Objective

In this project, students will gain experience in digital system design by implementing the custom McMaster Image Compression revision 18 (.mic18) image compression specification in hardware. Compressed data for a 320 x 240 pixel image 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 decoding circuitry will read the compressed data, recover the image using a custom digital circuit you designed, and store it back to the SRAM, from which the video graphics array (VGA) controller will read it and display it to the monitor.

1.2 Preparation

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

1.3 Organization of this Document

Section 2 discusses how an image is compressed with the .mic18 specification. Section 3 demonstrates image decompression by example with a 16x16 image. Section 4 presents the project guidelines. The project is divided into three milestones.

2 Image Compression

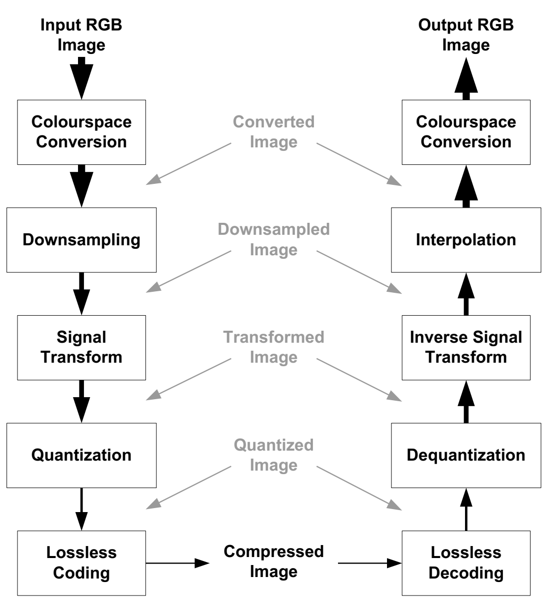


Figure 1. Flow of basic image compression (down the left) and decompression (up the right).

Figure 1 shows the conceptual flow for basic image compression and decompression. The size of the arrow roughly indicates the relative amount of data at each stage. In general, for image compression, we start with an image in the standard red/green/blue (RGB) format. This is converted to YUV format, where Y represents brightness, and U and V are chrominance components (which contain colour information). The YUV image is downsampled and then transformed into the frequency domain. This result is quantized, and finally, we apply lossless encoding to obtain the compressed information bitstream. Decoding proceeds in the reverse order: lossless decoding and dequantization (restores the frequency domain image), then the inverse signal transform (which brings the image back to the spatial domain), and finally, upsampling and colourspace conversion (produces an RGB image).

Compression enables smaller file sizes, which reduces storage and transmission costs. We convert the RGB image to YUV because the human eye is more sensitive to brightness than colour, so if we want to obtain more lossy compression (by reducing the quality), it will be more effective to compress the colour information. In the RGB format, there is no immediate way to compress colour more than brightness. YUV information is compressed by downsampling U and V in the .mic18 specification. Additionally, the human eye is less sensitive to high-frequency components in an image (compared to low frequencies). Transforming into the frequency domain enables us to compress higher-frequency components more than lower-frequency ones. We compress frequency information by applying quantization. Finally, we apply lossless compression as a final effort to reduce the file size.

Downsampling and quantization are not identically reversible, so some information is lost during compression. The decompressed image only approximates the original image. 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.

The focus of this project is the hardware implementation of a compression scheme. The .mic18 specification is not meant to rival existing compression standards for images. Simplifying choices have been made in developing the specification to facilitate a straightforward hardware implementation. This limits the capabilities in terms of the achievable compression/quality tradeoff. Even so, the concepts described above still apply in a general sense to the existing compression schemes. The following subsections will discuss each stage of image compression in more detail.

2.1 Colourspace Conversion and Downsampling

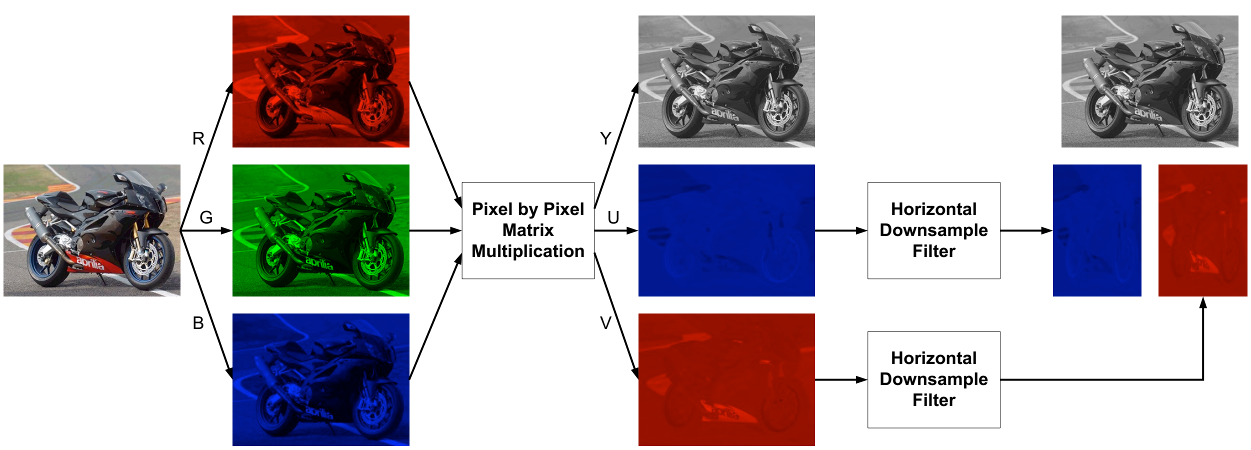


Figure 2. Colourspace conversion (RGB to YUV) and horizontal downsampling of U and V.

Figure 2 shows how a 320x240 image can be decomposed into its red, green, and blue components. Each pixel (composed of 1 red, 1 green, and 1 blue value) is independently transformed into the YUV colourspace. Each resulting pixel contains 1 Y (brightness), 1 U (blue chroma), and 1 V (red chroma). The chroma components contain only colour information and no brightness. In Figure 2, the bottom of the motorcycle is red, and the corresponding bright area in the V image indicates there are large values at these pixels, representing a strong red colouring. Note this says nothing about whether it is bright red or dark red, as the brightness comes only from Y.

Each RGB pixel is converted to a YUV pixel by performing the following matrix multiplication:

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{U} \\ \mathrm{V} \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \tag{1}$$

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).

The human eye is much more sensitive to brightness than colour (due to more rods than cones in the eye). For this reason, we choose to downsample U and V (the chroma components) to reduce the amount of data while having little effect on how the image is perceived. We horizontally downsample by 2, so the U and V images become half the original width (no vertical downsampling so the height is unchanged). Therefore, only 2/3 of the total data remains.

2.2 Signal Transform (into the Frequency Domain) and Quantization

The next compression stage involves transforming the image from the spatial domain to the frequency domain. High spatial frequencies (brightness/colour transitions that occur in a small amount of space) are not as noticeable to the human eye, so they may be reduced in accuracy (or even removed) without drastically affecting the overall quality of the image.

In the .mic18 specification, we use a 2-dimensional discrete cosine transform (DCT), and we work on blocks of 8x8 pixels (each 8x8 block is processed independently; we do not consider images with heights and widths not divisible by 8). The 2D DCT is computed with the following matrix multiplications:

$$\boldsymbol{S'} = \boldsymbol{CSC^T} \quad \text{where each element } [\boldsymbol{C}]_{i,j} = \begin{cases} \sqrt{\frac{1}{8}} \cos\left(\frac{i\pi}{8}(j+0.5)\right) & \text{if } i = 0 \\ \sqrt{\frac{2}{8}} \cos\left(\frac{i\pi}{8}(j+0.5)\right) & \text{otherwise} \end{cases} \tag{2}$$

$\boldsymbol{S}$ is the original 8x8 block (spatial domain), and $\boldsymbol{S'}$ is the transformed 8x8 block (frequency domain). In Equation 2 above, i and j are the row and column indexes, respectively, each goes from 0 to 7 inclusive.

To obtain compression, we quantize $\boldsymbol{S'}$. Each element in $\boldsymbol{S'}$ is divided by a (possibly) different quantization coefficient and then rounded to the nearest integer. A larger coefficient means that after division, we obtain a smaller magnitude integer, which requires fewer bits to be stored. The top left corner corresponds to frequency 0. The frequencies increase as we move down and to the right in the matrix. Thus, we should choose smaller quantization coefficients for the upper left part of the matrix and large coefficients for the lower right part to exploit the perception of the human eye.

For typical images, many values of $\boldsymbol{S}'$ will become 0 after quantization, and these zeros can be stored efficiently, as will be shown in Section 2.3. Even if nonzero, the division results in an integer that requires fewer bits to be represented. Two possible quantization matrices can be used in .mic18:

$$\boldsymbol{Q_0} = \begin{bmatrix} 8 & 4 & 8 & 8 & 16 & 16 & 32 & 32 \\ 4 & 8 & 8 & 16 & 16 & 32 & 32 & 64 \\ 8 & 8 & 16 & 16 & 32 & 32 & 64 & 64 \\ 8 & 16 & 16 & 32 & 32 & 64 & 64 & 64 \\ 16 & 16 & 32 & 32 & 64 & 64 & 64 & 64 \\ 16 & 32 & 32 & 64 & 64 & 64 & 64 & 64 \\ 32 & 32 & 64 & 64 & 64 & 64 & 64 & 64 \\ 32 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \end{bmatrix} \qquad \boldsymbol{Q_1} = \begin{bmatrix} 8 & 2 & 2 & 2 & 4 & 4 & 8 & 8 \\ 2 & 2 & 2 & 4 & 4 & 8 & 8 & 16 \\ 2 & 2 & 4 & 4 & 8 & 8 & 16 & 16 \\ 2 & 4 & 4 & 8 & 8 & 16 & 16 & 16 \\ 4 & 4 & 8 & 8 & 16 & 16 & 16 & 32 \\ 4 & 8 & 8 & 16 & 16 & 16 & 32 & 32 \\ 8 & 8 & 16 & 16 & 16 & 32 & 32 & 32 \\ 8 & 16 & 16 & 16 & 32 & 32 & 32 & 32 \end{bmatrix} \tag{3}$$

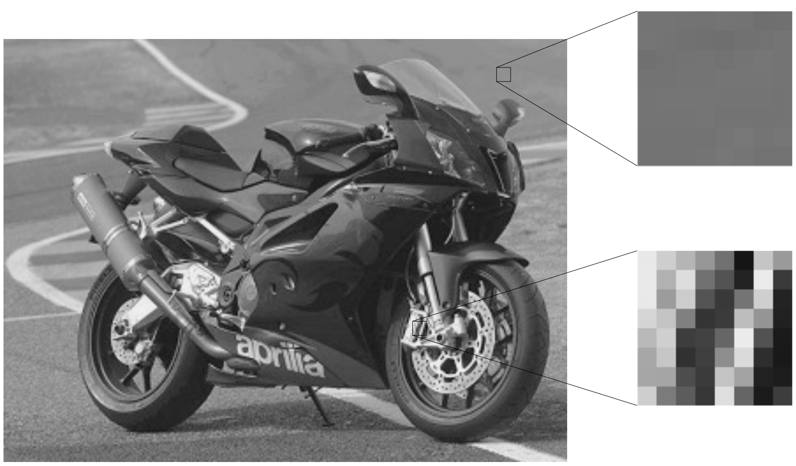
$\boldsymbol{S}'$ is divided element-wise by one of the quantization matrices (suppose we pick $\boldsymbol{Q_0}$) and then it is rounded to the nearest integer:

$$\boldsymbol{Z} = round(\boldsymbol{S}' \ ./ \ \boldsymbol{Q_0}) \tag{4}$$

If we look at each element of $\boldsymbol{Z}$:

$$[\boldsymbol{Z}]_{i,j} = round([\boldsymbol{S}']_{i,j} \ / \ [\boldsymbol{Q_0}]_{i,j}) \tag{5}$$

$\boldsymbol{Z}$ will later be losslessly coded.



126	126	125	126	125	126	126	126
126	126	125	126	125	126	126	126
126	126	125	126	125	127	126	125
126	125	126	126	125	125	125	126
126	125	125	125	125	129	127	126
126	126	126	126	126	129	130	130
125	127	130	129	129	128	129	129
126	126	129	129	129	130	128	126

(a)

232	207	182	150	113	22	197	154
234	178	221	114	84	35	235	59
232	156	204	85	57	116	207	36
213	195	129	53	59	190	150	33
186	208	62	66	103	235	83	27
168	199	49	64	142	219	47	26
166	166	78	59	195	163	33	29
207	123	92	55	223	102	27	62

(b)

126	126	125	126	125	126	126	126
126	126	125	126	125	126	126	126
126	126	125	126	125	127	126	125
126	125	126	126	125	125	125	126
126	125	125	125	125	129	127	126
126	126	126	126	126	129	130	130
125	127	130	129	129	128	129	129
126	126	129	129	129	130	128	126

(c1)

1013	−3	−1	0	−2	1	1	−1
−7	2	2	0	0	0	0	−1
4	0	−2	−1	0	0	0	0
1	−3	2	0	0	0	0	0
−1	1	−1	0	0	0	1	0
1	0	0	−2	0	0	0	0
0	−1	0	0	−1	0	0	−1
0	0	−1	0	−1	0	0	0

(c2)

127	−1	0	0	0	0	0	0
−2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(c3)

232	207	182	150	113	22	197	154
234	178	221	114	84	35	235	59
232	156	204	85	57	116	207	36
213	195	129	53	59	190	150	33
186	208	62	66	103	235	83	27
168	199	49	64	142	219	47	26
166	166	78	59	195	163	33	29
207	123	92	55	223	102	27	62

(d1)

1024	259	93	206	−103	−10	−21	45
123	20	124	−173	−75	201	−69	68
29	11	−4	−110	137	0	−46	118
8	−18	−16	16	19	−76	−29	−86
10	−10	26	3	35	−32	2	−24
3	−8	4	−3	−3	−17	6	−11
4	−6	7	−1	14	0	12	−9
4	−3	−10	1	6	0	5	−3

(d2)

128	65	12	26	−6	−1	−1	1
31	2	16	−10	−5	5	−2	1
4	1	0	−6	3	0	−1	2
1	0	0	1	0	−1	0	0
1	0	1	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	0	0	0

(d3)

Figure 3. DCT (c1→c2 and d1→d2) and quantization with Q_0 (c2→c3 and d2→d3) of 2 blocks of Y data.

After dividing each colour plane (Y, U, and V) into blocks of 8x8 pixels, we perform DCT and quantization to each block. Figure 3 shows this on two blocks of Y data. The image is 320x240, so there will be (320/8)*(240/8) = 1200 Y blocks and 600 blocks each of downsampled U and downsampled V (since these planes are half as wide). All the Y blocks are processed, followed by all the U and finally all the V blocks. Within each colour plane, we go across a row of blocks first, then down to the next row. Note that one row of blocks is 8 pixels high.

The block in Figure 3(a) is nearly uniform, so after DCT, in Figure 3(c2) there is a large value for frequency 0 and small values for the high frequencies. After quantization, we end up with mostly zeros, which can be stored efficiently. Many images contain nearly uniform blocks, so the DCT followed by quantization provides a way to represent the image with much less data. Although this leads to reduced quality of the decompressed image, the major features of the original block are still preserved.

The block in Figure 3(b) contains more variation due to the details of the motorcycle's wheel. The DCT and quantization results are shown in Figure 3(d2) and Figure 3(d3) respectively. In the presence of more details (captured by the higher frequencies), more data is needed to represent the block (compared to the block from Figure 3(a)). Even so, the amount of data (in Figure 3(d3)) is reduced from that required to represent the intensity of each pixel explicitly (the block in Figure 3(b)).

2.3 Lossless Coding

In the final stage of decoding, we need a mechanism for efficiently storing the data after DCT and quantization. We start with a simple 8-byte header, which indicates the image's width and height and which quantization matrix was used. After this, the losslessly coded blocks are packed into the bitstream in the same order described at the top of this page.

To losslessly code one block, the elements of $\boldsymbol{Z}$ (see Equation 4) are read out of the block in the order shown in Figure 4. This is known as the scan pattern.

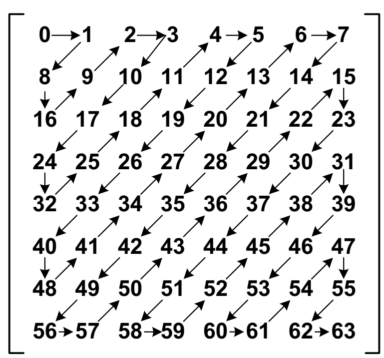


Figure 4. Scan pattern (order of values to be read from an 8x8 block).

Many images naturally have small values in the high frequencies, so after computing the DCT and quantizing, we typically have many zeros in the lower right part of the matrix. The above scan pattern, standard for many mainstream video/image compression schemes, deals with the low frequencies first (upper left) and the high frequencies last (lower right). This is beneficial because, in this order, we can get many consecutive zeros (from the high-frequency part), which can be efficiently coded. For each 8x8 block, as we follow the order of the scan pattern, values are encoded as follows, and a variable length prefix-coded header indicates what data will follow (a higher priority is given to the cases higher in the list):

1. Only zeros remain in the block, output the ZEROS_TO_END header (bits 1010).
2. A run of zeros exists but some nonzero coefficients still remain in the block. Output the ZERO_RUN header (bits 11) and then if
    a. The run of zeros is 8 or more in length, output the 3 bits 000, now the run of zeros will be 8 elements shorter, so repeat step 2 if necessary.
    b. The run of zeros is less than 8 in length (1-7 inclusive), output a 3 bit unsigned number indicating the length of the run.
3. There are two consecutive values between -4 and 3 inclusive, output the DOUBLE_3_BIT header (bits 00) followed by the first value (3 bits signed) and then the second value (another 3 bits signed).
4. There is a value between -4 and 3 inclusive, output the SINGLE_3_BIT header (bits 01) followed by the 3 bit signed value.
5. There is a value between -32 and 31 inclusive, output the 6_BIT header (bits 100) followed by the 6 bit signed value.
6. There is a value between -256 and 255 inclusive, output the 9_BIT header (bits 1011) followed by the 9 bit signed value.

In this entire document, it is implied that the signed representation is the 2's complement. Likewise unsigned is just the usual binary format. In the encoding, note that if a 3 bit value is followed by a run of 2 or more zeros then the 3 bit value is coded alone, but if it is followed by a single zero then the 2 values are coded together (the 3 bit value followed by the 3 bits 000).

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 value (127) needs to be coded as 9 bits signed (case 6 above), so we output 1011 (the header) followed by 001111111 (the 9 bit signed representation of 127). The next 2 values are -1 and -2 which can be coded together as 3 bit signed values (case 3 above), so we output 00 (header), then 111 (-1 on 3 bits signed) and finally 110 (-2 on 3 bits signed). At this point only zeros remain in the block, so 1010 (the ZEROS_TO_END header) is placed in the bitstream to conclude the coding of this block.

Coding of the second block from Figure 3 proceeds in the same way, however more data is required due to the presence of more nonzero value as well as the larger average magnitude of the values. As before, encoding proceeds until the last nonzero value in the scanned sequence (-1) is reached, after which a ZEROS_TO_END header 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 wide memory we are using).

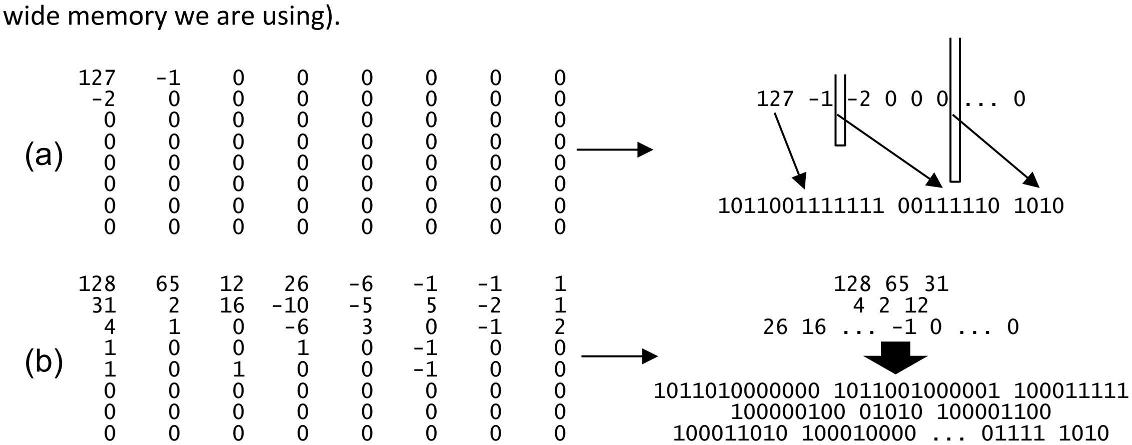


Figure 5. Lossless coding of each block from Figure 3.

3 Image Decompression

In this section, the whole decompression process for an image of size 16 x 16 is performed to illustrate the flow of data from one stage to another and elaborate on each step's internal details in the decompression. Since the aim of this project is the hardware implementation of the decompression specification, ***during the decompression, we only use fixed point arithmetic***, which has a more straightforward hardware design and generally uses fewer logic resources compared to floating point.

3.1 Lossless Decoding and Dequantization (Milestone 3 in Hardware)

Our reference image "example.ppm" is shown in Figure 6. The encoded bitstream is given in Figure 7. The header is 8 bytes. The first 4 bytes must be "DEADBEEF" in hex, identifying a .mic18 file (future revisions may use a different identifier). After this, in the following 2 bytes, the most significant bit indicates which quantization matrix to use (in this example, bit 0 means use $\boldsymbol{Q_0}$), and the 15 least significant bits indicate the width (bits 000 0000 0001 0000 indicate a width of 16). The following 2 bytes indicate the height, which is also 16 in our example. After the header, the main content of the image, the sequence of compressed blocks, begins.

Figure 6. The reference image example.ppm to be compressed.

```
DE EA BE EF 00 10 00 10 B3 A4 2A 0E 99 6C DA 0A
1C 60 99 DA 74 6A 12 73 A2 1E 9C 27 67 0E C6 DD
F7 6D F6 7B 9B BA 09 0F 9C 36 0D 05 7E A7 21 67
A6 F9 38 38 82 E9 3B 09 F1 83 F8 FE 2B EE D1 D5
9C E1 06 71 46 CE 0A 4D 9F 37 2A 0A C9 CE 82 20
89 D0 4E 53 F3 04 E3 2A F9 C1 C2 6D 07 06 54 E2
60 99 D3 67 5A 4C DC FC 24 2C 37 27 24 3D 39 09
38 08 76 36 EF BB 6E B4 14 5A 33 0E 99 C2 E0 A6
07 21 90 58 23 67 53 A3 3D BF 61 07 01 D9 7E 31
97 D5 A3 E3 F3 68 74 1E 23 2D 83 CE 8C 93 39 29
1A 82 07 8F 9C 1C 1C E4 3F 47 09 D0 42 6A CF D0
A4 CD B3 34 9B 64 B8 C2 08 72 13 04 17 DA 74 1E
70 7E 9A B3 C4 DA 0B 3A 74 E6 0E 55 93 85 E8 FC
C0 3F C7 07 34 00
```

Figure 7. In the compressed bitstream example.mic18, the order of the bytes is across first and then down.

Figure 8 depicts the block decoding process. Variable length prefix coding is used for the header, which means depending on the first few bits of the header, we may need to read additional bits to determine the remainder of the header. We start by reading 2 bits: if we get 00, then two 3-bit values follow; if we get 01, then one 3-bit value follows; if we get 11, then a run of zeros follows; and if we get 10, then we need to read more bits to complete the header. In the case where the first two bits are 10, if the next bit we read is 0, then one 6-bit value follows; otherwise, we need to read one more bit to complete the header. If this bit is 1, then one 9-bit value follows; otherwise, there is a run of zeros to the end of the block. Following this procedure, values 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 a ZEROS_TO_END header is decoded (in which case the rest of the block is filled with 0s), or when 64 values (including runs of zeros) have been decoded, making the block complete. Note that the values are signed (2's complement format).

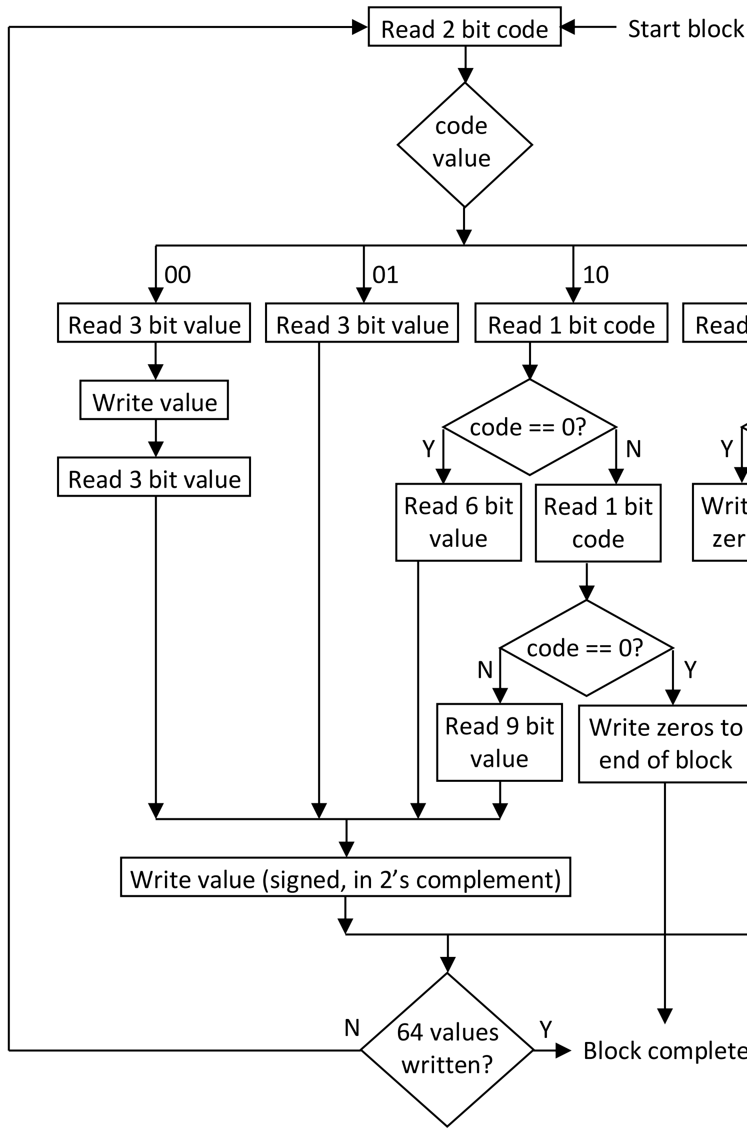


Figure 8. Lossless decoding of a block based on the prefix code.

The sequence begins with hex "B3 A4 2A …" which is "101100111010010000101010 …" in binary, and when applied to this bitstream, the result of the lossless decoding procedure is as shown on the following two pages in Figures 9 and 10.

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 (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.

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

1011001110100 : 116         11010      : 0 0
100001010     : 10          00100001   : -4 1
100000111     : 7           11101      : 0 0 0 0 0
01001         : 1           00111000   : -1 0
100101101     : -19         01001      : 1
100110110     : -10         11011      : 0 0 0
100000101     : 5           00111000   : -1 0
00001110      : 1 -2        01110      : -2
00110000      : -2 0        11000      : 0 0 0 0 0 0 0 0
01001         : 1           11011      : 0 0 0
100111011     : -5          01110      : -2
01001         : 1           11111      : 0 0 0 0 0 0 0
11010         : 0 0         01110      : -2
00110101      : -2 -3       11011      : 0 0 0
00001001      : 1 1         01111      : -1
00111001      : -1 1
```

(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
```

```
Y 0 1 -------------------------------------------------------------------------------------------

1011001111011 : 123         00111000   : -1 0
100110111     : -9          100000101  : 5
01110         : -2          11010      : 0 0
100000100     : 4           01001      : 1
100001111     : 15          11011      : 0 0 0
100111000     : -8          00001001   : 1 1
01101         : -3          11110      : 0 0 0 0 0 0
100000110     : 6           00110000   : -2 0
100000101     : 5           01111      : -1
01111         : -1          11100      : 0 0 0 0
11010         : 0 0         01111      : -1
100111001     : -7          11100      : 0 0 0 0
00001011      : 1 3         01010      : 2
00111101      : -1 -3       11111      : 0 0 0 0 0 0 0
00110111      : -2 -1       01110      : -2
11001         : 0           11010      : 0 0
00111000   : -1 0           00111010   : -1 2
```

(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
```

```
Y 1 0 -------------------------------------------------------------------------------------------

1011001110011 : 115         100000100  : 4
100001000     : 8           01001      : 1
00110011      : -2 3        11010      : 0 0
100010100     : 20          00001001   : 1 1
01101         : -3          11001      : 0
100111000     : -8          01001      : 1
00101001      : -3 1        11111      : 0 0 0 0 0 0 0
00110110      : -2 -2       00110000   : -2 0
01111         : -1          01001      : 1
100110111     : -9          11000      : 0 0 0 0 0 0 0 0
00101010      : -3 2        11001      : 0
00001010      : 1 2         01010      : 2
11001         : 0           11111      : 0 0 0 0 0 0 0
00111001      : -1 1        00111000   : -1 0
11010         : 0 0         00111000   : -1 0
00001000      : 1 0         01001      : 1
```

(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
```

```
Y 1 1 -------------------------------------------------------------------------------------------

1011010000011 : 131         11001      : 0
100000110     : 6           00111001   : -1 1
01010         : 2           00100001   : -4 1
100111000     : -8          11101      : 0 0 0 0 0
100110000     : -16         00111001   : -1 1
01001         : 1           00001001   : 1 1
100111010     : -6          00111000   : -1 0
01101         : -3          00001000   : 1 0
100111010     : -6          01110      : -2
11010         : 0 0         11000      : 0 0 0 0 0 0 0 0
01001         : 1           11011      : 0 0 0
100110111     : -9          01110      : -2
00111111      : -1 -1       11111      : 0 0 0 0 0 0 0
00001001      : 1 1         01110      : -2
00001011      : 1 3         11011      : 0 0 0
00001101      : 1 -3        01110      : -2
```

(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
```

Figure 9. Lossless decoding of the Y blocks in example.mic18.

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 -----------------------------------------------------------------------------------------

1011010000010 : 130          00110011  : -2 3
100010110     : 22           11011     : 0 0 0
100011001     : 25           01111     : -1
100001110     : 14           11011     : 0 0 0
100110011     : -13          00001000  : 1 0
100001011     : 11           00111000  : -1 0
100000101     : 5            00001110  : 1 -2
00110000      : -2 0         11001     : 0
00111001      : -1 1         01111     : -1
00001100      : 1 -4         11000     : 0 0 0 0 0 0 0 0
100000101     : 5            11000     : 0 0 0 0 0 0 0 0
100000100     : 4            11001     : 0
01101         : -3           01111     : -1
100111010     : -6           1010      : 0 0 0 0 0 0 0 0
100111010     : -6                       0 0 0
```

(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

```
U 1 0 -----------------------------------------------------------------------------------------

1011010001111 : 143          00011111  : 3 -1
100011111     : 31           00111000  : -1 0
100110110     : -10          00111000  : -1 0
100001110     : 14           00111001  : -1 1
100000111     : 7            11001     : 0
100010001     : 17           00001111  : 1 -1
100101101     : -19          11010     : 0 0
100000111     : 7            00111000  : -1 0
100111010     : -6           01001     : 1
00110010      : -2 2         11010     : 0 0
01001         : 1            00001000  : 1 0
100111001     : -7           01001     : 1
00101001      : -3 1         1010      : 0 0 0 0 0 0 0 0
00011010      : 3 2                      0 0 0 0 0 0 0 0
100000100     : 4                        0 0 0 0 0 0 0
00001111      : 1 -1
```

(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

```
V 0 0 -----------------------------------------------------------------------------------------

1011001111110 : 126          100000100 : 4
100001010     : 10           00010111  : 2 -1
01001         : 1            11011     : 0 0 0
100110110     : -10          01001     : 1
11001         : 0            11010     : 0 0
100110100     : -12          00001111  : 1 -1
100110110     : -10          00111000  : -1 0
11001         : 0            00111111  : -1 -1
00101110      : -3 -2        01001     : 1
00110000      : -2 0         1010      : 0 0 0 0 0 0 0 0
100000100     : 4                        0 0 0 0 0 0 0 0
00111001      : -1 1                     0 0 0 0 0 0 0 0
00001001      : 1 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

```
V 1 0 -----------------------------------------------------------------------------------------

1011001111000 : 120          00111111  : -1 -1
100110110     : -10          00110000  : -2 0
100000101     : 5            00001111  : 1 -1
100111010     : -6           11110     : 0 0 0 0 0 0
01110         : -2           00111000  : -1 0
100111001     : -7           00111001  : -1 1
100000111     : 7            1010      : 0 0 0 0 0 0 0 0
00101010      : -3 2                     0 0 0 0 0 0 0 0
11001         : 0                        0 0 0 0 0 0 0 0
00111000      : -1 0                     0 0 0 0 0 0 0 0
01011         : 3                        0
11010         : 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

Figure 10. Lossless decoding of the downsampled U and V blocks in example.mic18.

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 in the rightmost column of Figures 9 and 10 collectively make up the quantized image. From the flow illustrated in Figure 1, the next step is dequantization, which leads to the transformed image. This is done using pointwise multiplication between the losslessly decoded data (matrices in the rightmost column above) and the appropriate quantization matrix ($\boldsymbol{Q_0}$ in this case). For matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$, pointwise (or element-wise) multiplication is defined in Equation 6 below.

$$\boldsymbol{A} = \boldsymbol{B} .* \boldsymbol{C} \quad \Longleftrightarrow \quad [\boldsymbol{A}]_{i,j} = [\boldsymbol{B}]_{i,j} * [\boldsymbol{C}]_{i,j} \tag{6}$$

The quantized images are shown on the leftmost column on page 12 and the result of dequantization is shown in the central column.

3.2 IDCT (Inverse Discrete Cosine Transform) (Milestone 2 in Hardware)

After dequantizing the blocks, the IDCT must be performed to recover the downsampled image. The IDCT is defined in Equation 7:

$$\boldsymbol{S} = (int)\frac{1}{65536}\left(\boldsymbol{C^T} \times (int)\frac{1}{256}(\boldsymbol{S'C})\right) \tag{7}$$

where the matrix $\boldsymbol{S'}$ is the dequantized 8x8 block, and the output of the transform $\boldsymbol{S}$ is the block of 8x8 samples from the downsampled image. The element on the i^{th} row and j^{th} column of the 8x8 matrix $\boldsymbol{C}$ is:

$$[\boldsymbol{C}]_{i,j} = \begin{cases} (int)\left(\sqrt{\frac{1}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & \text{if } i = 0 \\ (int)\left(\sqrt{\frac{2}{8}}\cos\left(\frac{i\pi}{8}(j+0.5)\right) \times 4096\right) & \text{otherwise} \end{cases} \tag{8}$$

$$\boldsymbol{C} = \begin{bmatrix} 1448 & 1448 & 1448 & 1448 & 1448 & 1448 & 1448 & 1448 \\ 2008 & 1702 & 1137 & 399 & -399 & -1137 & -1702 & -2008 \\ 1892 & 783 & -783 & -1892 & -1892 & -783 & 783 & 1892 \\ 1702 & -399 & -2008 & -1137 & 1137 & 2008 & 399 & -1702 \\ 1448 & -1448 & -1448 & 1448 & 1448 & -1448 & -1448 & 1448 \\ 1137 & -2008 & 399 & 1702 & -1702 & -399 & 2008 & -1137 \\ 783 & -1892 & 1892 & -783 & -783 & 1892 & -1892 & 783 \\ 399 & -1137 & 1702 & -2008 & 2008 & -1702 & 1137 & -399 \end{bmatrix} \tag{9}$$

Evaluating all cases in Equation 8 produces Equation 9. Equation 7 is defined similarly to the DCT from Equation 2. However, the important differences are the intermediate and final scalar divisions and the use of fixed-point coefficients. To reiterate, the reason for choosing fixed-point (i.e. integer) operations is that they facilitate a more efficient hardware implementation than floating-point operations.

Fixed-point, however, brings in necessary approximations that we achieve by using an intermediate bit-width 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 7) to obtain an integer-only fixed-point value before processing (-4096 to 4096). Because the sample matrix is multiplied on both sides by the coefficient matrix, the data will be left-shifted by 24 positions in total. The first adjustment of a right shift by 8 positions is provided after the post-multiplication $\boldsymbol{S'C}$ and the second adjustment is given by a right shift by 16 positions after the pre-multiplication $\boldsymbol{C^T}(\boldsymbol{S'C})$. 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 used to ensure that rounding and overflows that have led to values out of the 8-bit unsigned 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
```

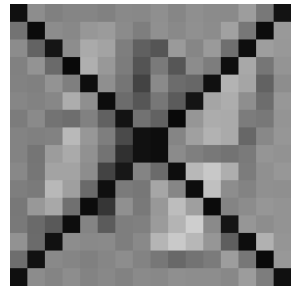
Figure 11. Dequantization (left to center column) followed by IDCT (center to right column) on each 8x8 block.

3.3 Upsampling and Colourspace Conversion (Milestone 1 in Hardware)

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

Y

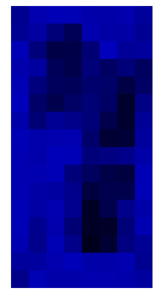


131	185	168	178	185	159	168	154
168	149	130	104	149	193	156	150
168	124	87	57	121	187	154	150
175	154	86	80	111	110	113	110
182	140	86	97	114	79	72	101
187	94	89	91	116	105	54	133
193	122	138	123	114	86	41	143
159	167	181	174	121	46	61	160
188	158	194	173	130	126	128	176
174	174	192	136	78	62	83	177
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

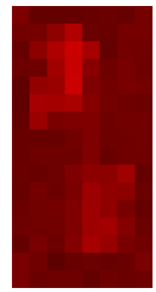
U

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 12. The output of IDCT is the downsampled YUV data.

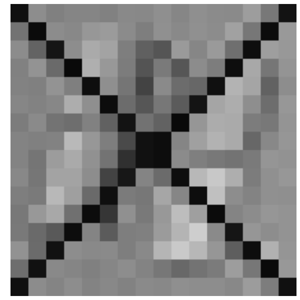
The output of IDCT is downsampled YUV data, where U and V are horizontally downsampled by 2, as shown in Figure 12. The downsampled data becomes the even columns in the upsampled image, and the odd columns are generated by horizontal interpolation. For each row (rows are processed independently), we interpolate with a 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. Given n samples in each row of the U and V planes, we produce $2n$ samples for the corresponding row of the reconstructed plane as follows:

$$U'[j] = \begin{cases} U\left[\frac{j}{2}\right] & j\ even \\ (int)\frac{1}{256}\left(21U\left[\frac{j-5}{2}\right] - 52U\left[\frac{j-3}{2}\right] + 159U\left[\frac{j-1}{2}\right] + 159U\left[\frac{j+1}{2}\right] - 52U\left[\frac{j+3}{2}\right] + 21U\left[\frac{j+5}{2}\right] + 128\right) & j\ odd \end{cases} \quad (10)$$

where U is the downsampled version and U' is the upsampled version. An analogous definition applies to V. In our example, with an image of size 16x16, n is 8. In the project, the image will be of size 320x240, so n will be 320/2 = 160. The borders of ***each row*** are handled by replicating the respective border sample, i.e. $U[0] = U[-1] = U[-2]$, and $U[n-1] = U[n] = U[n+1] = U[n+2]$. Note again the use of integer operations. These coefficients are borrowed from the ISO/IEC 13818-2 document on MPEG2 digital video decoding. Applying this filter to the U and V planes shows the result in Figure 13.

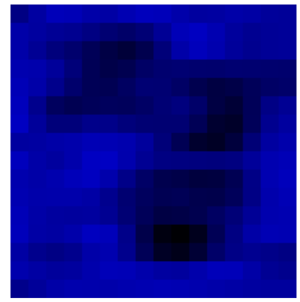
Having restored the complete Y, U and V planes, the only remaining step is the colourspace conversion from YUV back to RGB. As discussed above (Equation 1), the conversion from RGB to YUV is a matrix multiplication. In essence, the RGB values of a given pixel are viewed as a vector in 3-space, and the 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 nonzero, we may find its inverse, which will

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



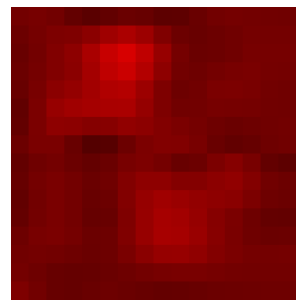
Y

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

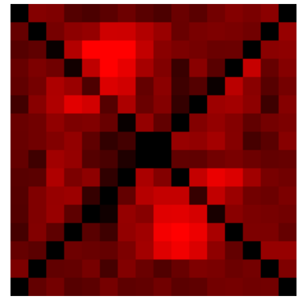
Figure 13. The result after horizontally upsampling U and V.

provide the transformation back to the RGB basis from the YUV basis. This is applied to each pixel independently. Accounting for the vector addition to first shift the basis, the inverse conversion of the one above is:

$$\begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} = (int)\frac{1}{65536}\left(\begin{bmatrix} 76284 & 0 & 104595 \\ 76284 & -25624 & -53281 \\ 76284 & 132251 & 0 \end{bmatrix}\left(\begin{bmatrix} \mathrm{Y} \\ \mathrm{U} \\ \mathrm{V} \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}\right)\right) \qquad (11)$$

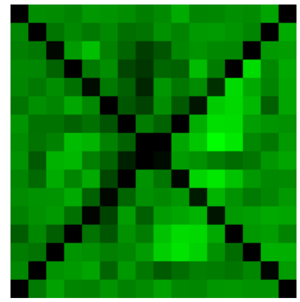
As in the case of IDCT, an identical clipping function is used to deal with the under/overflow errors. The outputs R, G, and B are each on 8 bits unsigned. The final result is shown in Figure 14.

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



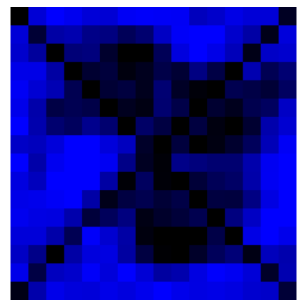
R

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 14. After colourspace conversion, we obtain the final decompressed RGB image.

The final decompressed image, as shown on the left (a) in Figure 15, requires 16x16x3 = 768 bytes (plus a header of 13 bytes), whereas the compressed image required only 214 bytes, giving a compression ratio of more than 3x. The original image appears on the right (c). As you can see, there is some reduction in the quality due to information lost during the compression and decompression process. The center image (b) shows the decompressed image for the case when the quantization matrix $\boldsymbol{Q_1}$ from Equation 3 is used (instead of $\boldsymbol{Q_0}$ 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 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.

	$\boldsymbol{Q_0}$ image	$\boldsymbol{Q_1}$ image	Original image
SNR (dB)	18.65	20.40	∞
File size (bytes, including header)	214	356	781
Compression ratio	3.65	2.19	1

Table 1. Quality versus compression tradeoff for the images using quantization matrices $\boldsymbol{Q_0}$ and $\boldsymbol{Q_1}$.

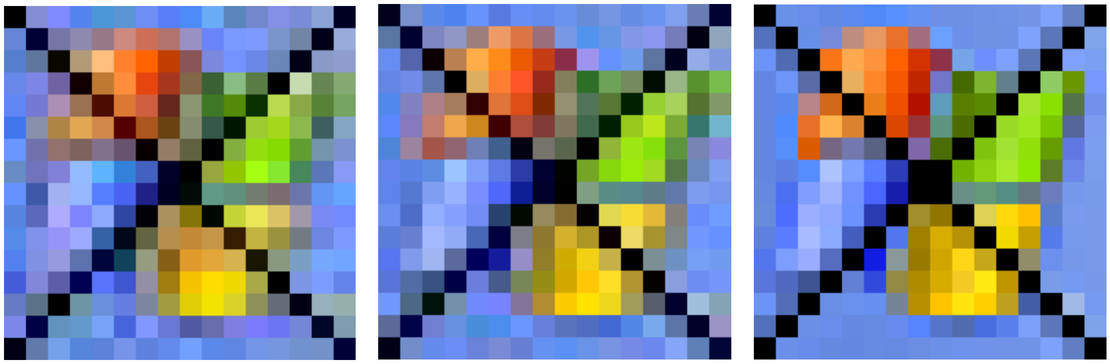


Figure 15. Final decompressed images using quantization matrices Q_0 (a) and Q_1 (b) and the original image (c).

Equation 12 defines the signal-to-noise ratio where N is the number of pixels, $error$ is the root mean squared error, and the 255 on the numerator is the maximum value of any pixel:

$$SNR\ (dB) = 20 \log_{10}\left(\frac{255}{\sqrt{\frac{error}{N}}}\right) \text{ where } error = \sum_{n=0}^{N-1}(reference\_pixel_n - image\_pixel_n)^2 \qquad (12)$$

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

4 Project Guidelines

The design task of implementing the overall decoder has been partitioned into three milestones, each of which can be completed within one week. As part of the validation methodology, implementation will proceed from the VGA display backward to the compressed input. Starting from the display end enables a visual progress check at each milestone, providing the framework for incremental integration of each newly designed module into the overall system.

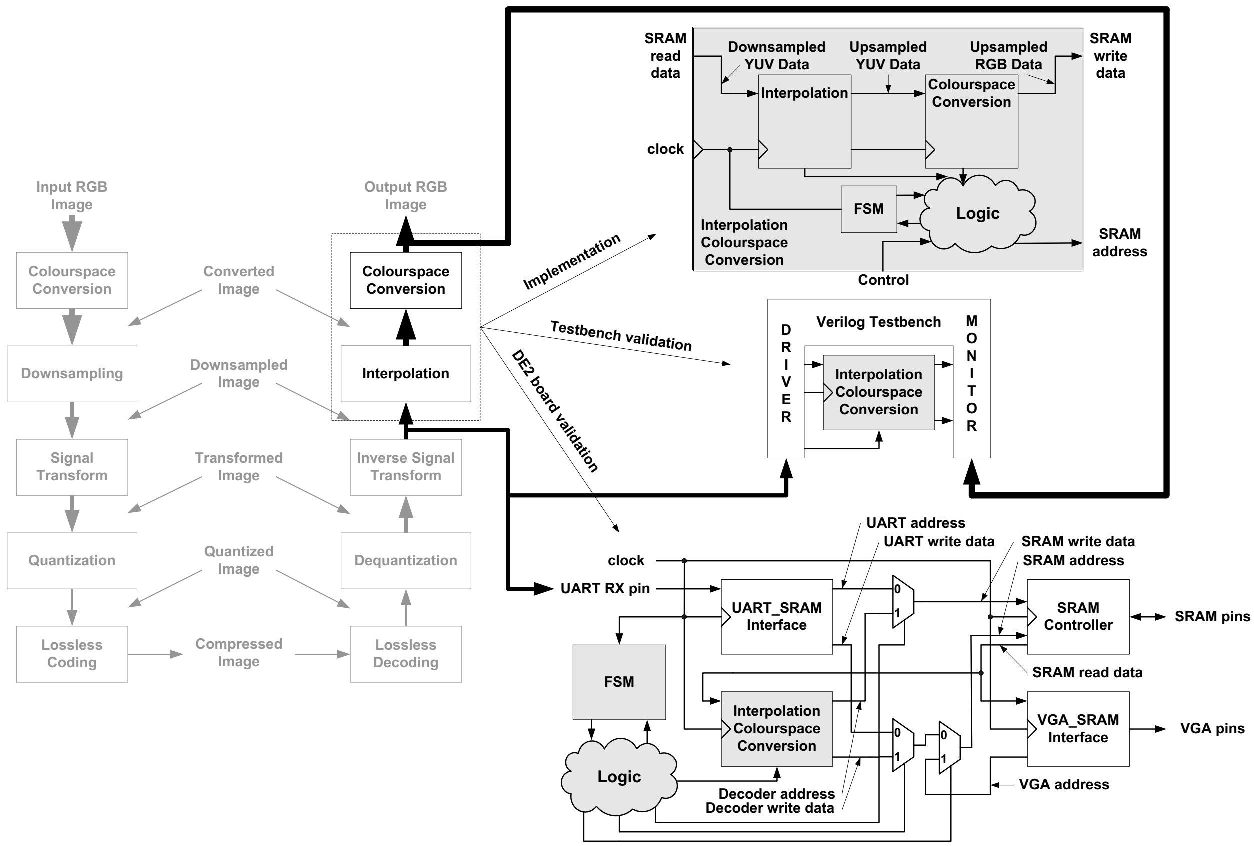


Figure 16. Overview of the upsampling and colourspace conversion hardware implementation (milestone 1).

4.1 The Software Model and its Integration in the Design Process

Starting with a .mic18 file, there are 3 separate source codes, 1 for each milestone. The input/output files are organized exactly like the SRAM in hardware, that is 2^{18} = 262144 locations with 16 bits per location. Each source code reads data from a software model of the SRAM, performs the necessary computation, and then writes the data back to the SRAM software model. The decoding process starts at milestone 3, and the source code “decode_m3” takes a .mic18 file and produces a .sram_d2 file. Continuing into milestone 2, the source code “decode_m2” takes a .sram_d2 file and produces a .sram_d1 file. Finally, for milestone 1, “decode_m1” takes a .sram_d1 file and produces a .sram_d0 file and a .ppm file.

By separating the decoding process, we can do part of the decoding in software and part of it in hardware. The decoding software handles the first part of the decoding task and the intermediate data can be passed to a testbench or to the board, where the hardware can take over the remainder of the decoding task to produce the image. 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. For example, to validate if milestone 1 is working properly in hardware, the .sram_d1 file can be used to initialize the SRAM, and after the hardware simulation is finished, the .sram_d0 file can be used to validate if the contents of SRAM are correct.

4.2 Milestone 1: Upsampling and Colourspace Conversion

Figure 16 shows the implementation and validation plan for milestone 1. In this first week, colourspace conversion and interpolation 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 decoder access to the external memory (by multiplexing). In fact, all the shaded portions of the figure will have to be built or modified. The SRAM Controller, UART SRAM Interface and VGA SRAM Interface shown in the figure have been provided in lab 5.

Figure 17 shows the memory layout which is used for milestone 1. The last portion of the memory holds the completed RGB image stored in {R0 G0}, {B0 R1}, {G1 B1} … format (as in lab 5 experiment 1). This is where the decompressed image will reside for all three milestones and, thus, for the completed project as well. Where it differs from the completed project is in the fact that downsampled YUV data is stored in the SRAM as decoder stimuli, since the earlier parts of the decoder are not yet implemented. By default (from the backbone in lab 5), the UART SRAM interface writes data in the SRAM starting at address 0 and the VGA SRAM interface starts reading from address 0. Clearly, the VGA SRAM interface must be modified to read the RGB data from the correct segment, as indicated in Figure 17. *Also, the UART receiver in lab 5 experiment 4 strips the header from the .ppm file, **which is no longer needed**.*

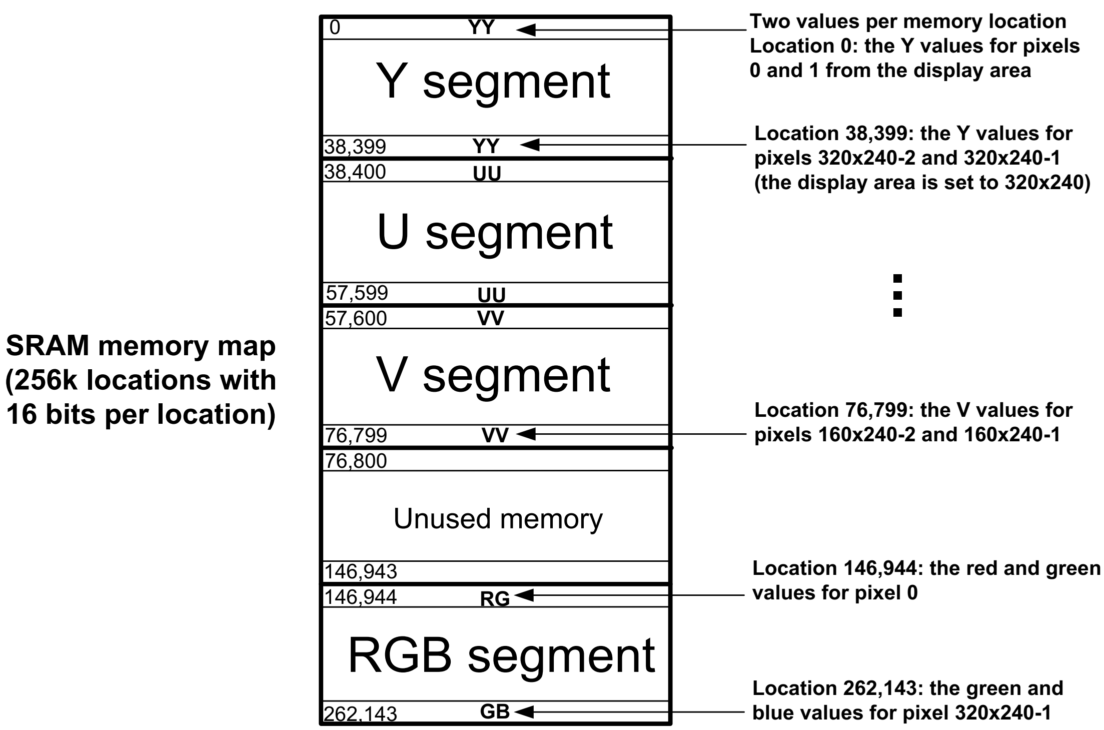
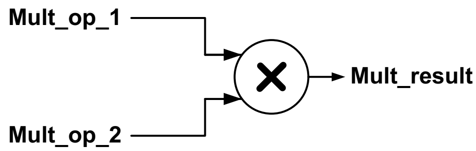


Figure 17. Memory layout for milestone 1.




```
logic [31:0] Mult_op_1, Mult_op_2, Mult_result;
logic [63:0] Mult_result_long;

assign Mult_result_long = Mult_op_1 * Mult_op_2;
assign Mult_result = Mult_result_long[31:0];
```

Figure 18. Instantiation of a single 32-bit multiplier.

Figure 18 shows how a single 32-bit integer multiplier can be instantiated in your design. As will be elaborated on soon, to end up with a resource-efficient implementation, multipliers must be shared because they are costly in terms of logic resources. Specifically, for this milestone, you are given the ***constraint of 3 multipliers (with input operands on 32 bits or less). Note that these 3 multipliers are shared among colourspace conversion and upsampling, including the symmetry in upsampling as summarized in Equation (13).*** *Thus,* there will be three instantiations of the form for Figure 18 (note that you are required to meet utilization constraints). Each of these tasks comprises more than one multiplication, so the multipliers for each task must be shared. For this reason, the instantiation given in Figure 18 should be used, and the signals Mult_op_1, Mult_op_2 and Mult_result should be multiplexed.

By explicitly instantiating multipliers, we are certain to have that many multipliers used in hardware. For example, in the following code snippet:

```
always_comb begin
    if (select == 1'b0) a = b*c;
    else a = d*e;
end
```

The multiplications happen at mutually exclusive times, so we could have 1 hardware multiplier with `b` and `d` multiplexed at one input to the multiplier, and `c` and `e` multiplexed at the other input to the multiplier. However, this not explicitly stated, so the computer-aided design (CAD) tool (in our case, Quartus) may choose to instantiate 2 hardware multipliers to compute both `b*c` and `d*e`, and then take these 2 results and multiplex them to obtain the output `a`. To avoid this problem, we can explicitly specify that 1 (and only 1) multiplier should be used by multiplexing the operands:

```
always_comb begin
    if (select == 1'b0) begin
        op1 = b;
        op2 = c;
    end
    else begin
        op1 = d;
        op2 = e;
    end
end
assign a = op1*op2;
```

Next, some architectural examples are provided regarding how the multipliers can be shared.

Figure 19 shows a straightforward implementation of a FIR filter, where the samples are stored in a shift-register 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 upsampled (U and V). Furthermore, we must consider if we can actually receive a new U and a new V sample on every clock cycle; if we cannot, then on some clock cycles, this hardware will be under-utilized. A similar argument applies to whether or not we can process one upsampled U and one upsampled V per clock cycle after the FIR filter.

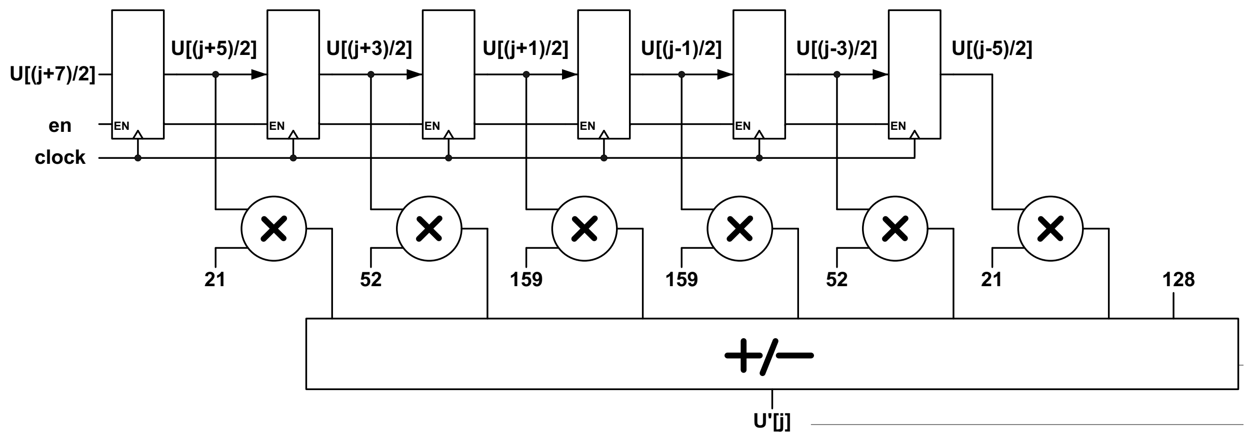


Figure 19. Resource inefficient FIR filter implementation.

Figure 20 shows how a single multiplier can be shared to accomplish the same computation as Figure 19, but at the expense of throughput. In this case, the boxed portion is known as a Multiply-Accumulate (MAC) unit, which works by always multiplying the two inputs, 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 the constant factor, as well as initialization, which in this case is to account for the +128 in the upsampling equations of Equation 10, also shown in Figure 19. 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.

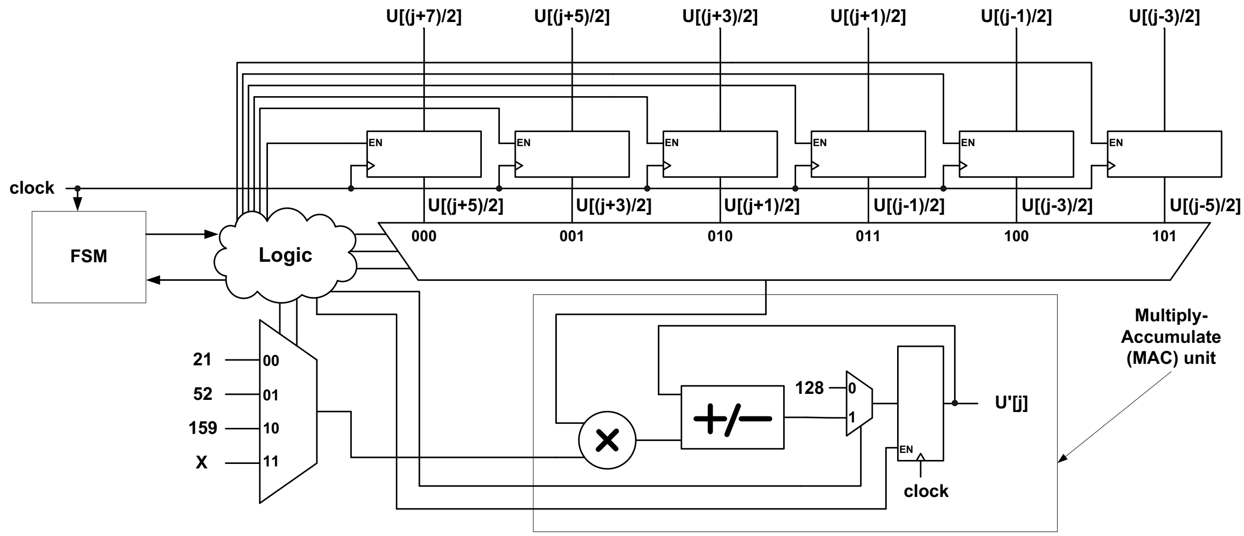


Figure 20. Resource-efficient FIR filter implementation.

Figure 21 shows a cost-effective FIR implementation similar to Figure 20, 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 majority of the 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 128), and during the next 6 clock cycles, the samples are circularly shifted, each time the contents of the rightmost register being accumulated. At the end of these 6 clock cycles, each data returns to its original register, and the accumulation is complete (a new reconstructed sample has been produced). Although identical to Figure 20 in throughput, eliminating the large sample selector mux makes this design substantially more efficient.

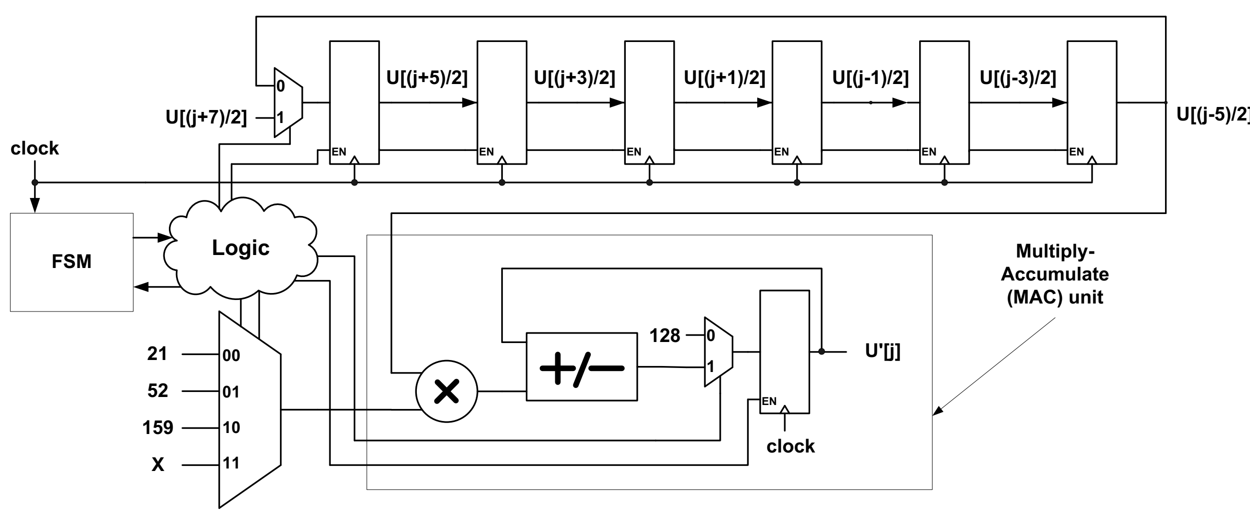


Figure 21. Resource efficient FIR filter implementation.

In addition to the FIR filter implementation, we are faced with the challenge of sharing a multiplier for colourspace conversion. Figure 22 shows the straightforward way to implement Equation 11, with the advantage of high throughput (one full RGB pixel per clock cycle). However, due to the constraints of using only **three multipliers** for both interpolation and colourspace conversion, this solution is not suitable for our implementation.

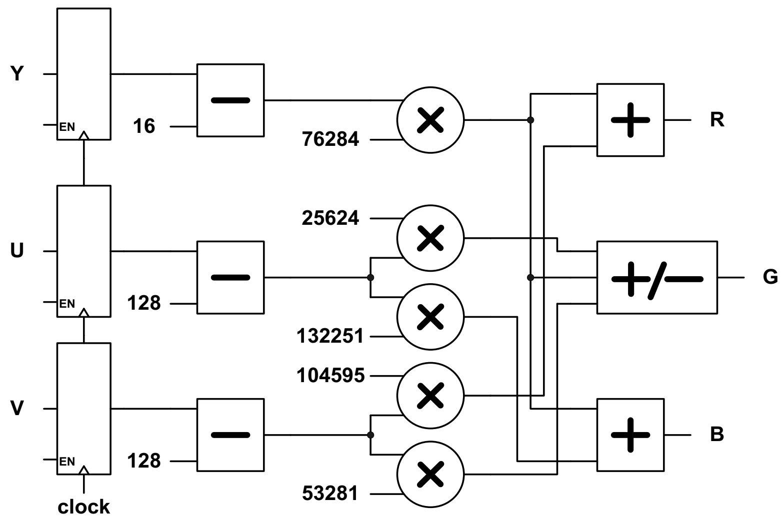


Figure 22. Resource inefficient colourspace conversion implementation.

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. The data path for this is shown in Figure 23. In this case, using 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.

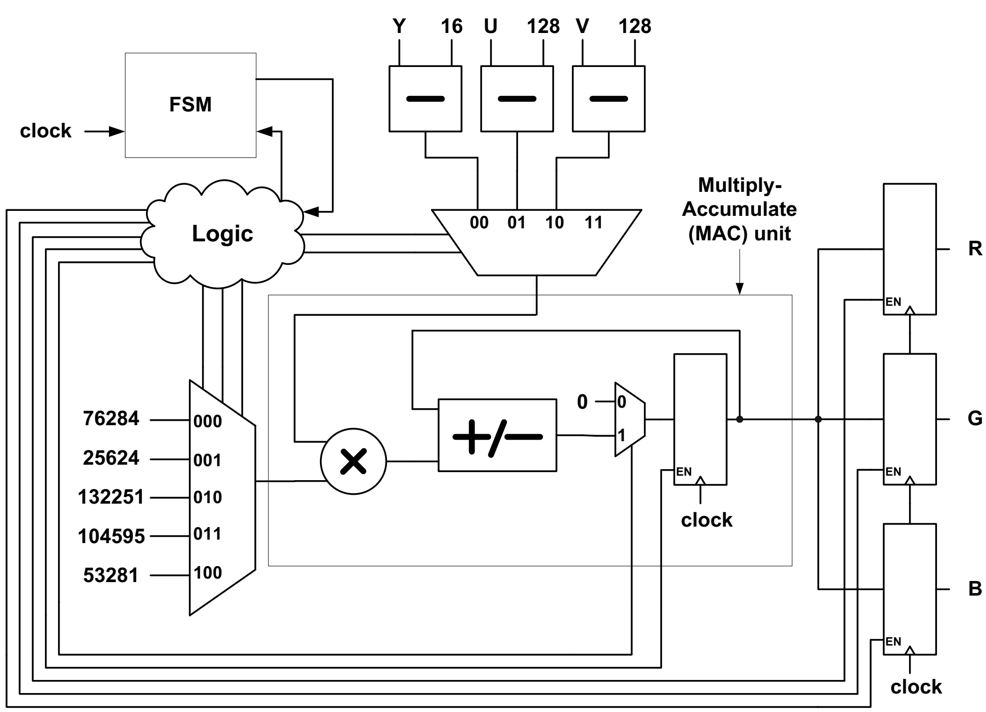


Figure 23. Resource efficient colourspace conversion implementation.

Table 2 provides a good starting point for a state table that can be used for milestone 1. You are strongly urged to use state tables throughout development, verification and debugging.

State code / clock cycle	0	1	2	3	4	5	…
SRAM_address							
SRAM_read_data							
SRAM_write_data							
SRAM_we_n							
R							
G							
B							
Y							
U’							
U[(j+5)/2]							
U[(j-5)/2]							
V’							
V[(j+5)/2]							
V[(j+5)/2]							
…							

Table 2. A starting point for the state table in milestone 1.

The average utilization of the 3 multipliers for milestone 1 must be at least 75%. The utilization of a multiplier is defined as the number of clock cycles a multiplier is used to perform calculations divided by the total number of clock cycles for the respective milestone. For clarification purposes, consider the following example on a hypothetical milestone whose duration is 100 clock cycles. Let’s assume we use two multipliers. The first multiplier performs calculations for 70 clock cycles hence its utilization is 70%. The second multiplier performs calculations for 90 clock cycles hence its utilization is 90%. For this *hypothetical* milestone, the average utilization is 80%.

In addition to the above-mentioned utilization constraint, exploiting **the symmetry in the interpolation filter is mandatory**. The symmetry of the filter’s coefficients from Equation 10 can be exploited as:

$$21\left(U\left[\frac{j-5}{2}\right]+U\left[\frac{j+5}{2}\right]\right)-52\left(U\left[\frac{j-3}{2}\right]+U\left[\frac{j+3}{2}\right]\right)+159(U\left[\frac{j-1}{2}\right]+U\left[\frac{j+1}{2}\right]) \qquad (13)$$

Note the 256 scale factor and the 128 offset from Equation (10) are not shown in the equation above, whose purpose is **only** to illustrate how symmetry can be exploited. Naturally, the 256 scale factor and the 128 offset from Equation (10) must be used when interpolating the odd samples for the U and V. Based on the above, it can be concluded that the interpolation for each of the odd samples from the U and V planes can be done with **only 3, rather than 6, multiplications**.

Note also that interpolated U’ and V’ can take values above 255 and below 0. These interpolated values should **not** be clipped before passing them to the colourspace converter (only the final R, G, and B values are clipped to 8 bits in the unsigned format in milestone 1).

Finally, as is the case for any digital design project, hardware resources should not be wasted. In your project report, you must justify your design decisions, e.g., the need for registers and their purpose.

4.3 Milestone 2: IDCT

Figure 24 shows the overview for milestone 2 which is the implementation of IDCT. As before, the shaded portions of the figure indicate what needs to be built or modified. In particular, notice 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 decoder circuitry (all of which will eventually reside in the block currently labelled "Partial Decoder") to access the SRAM.

As discussed in Section 4.1, the software model can generate the necessary initialization and verification data for the SRAM. The testbench uses this data for validation, and it can be sent to the board with a working design to get visual confirmation that the IDCT, upsampling, and colourspace conversion are working together.

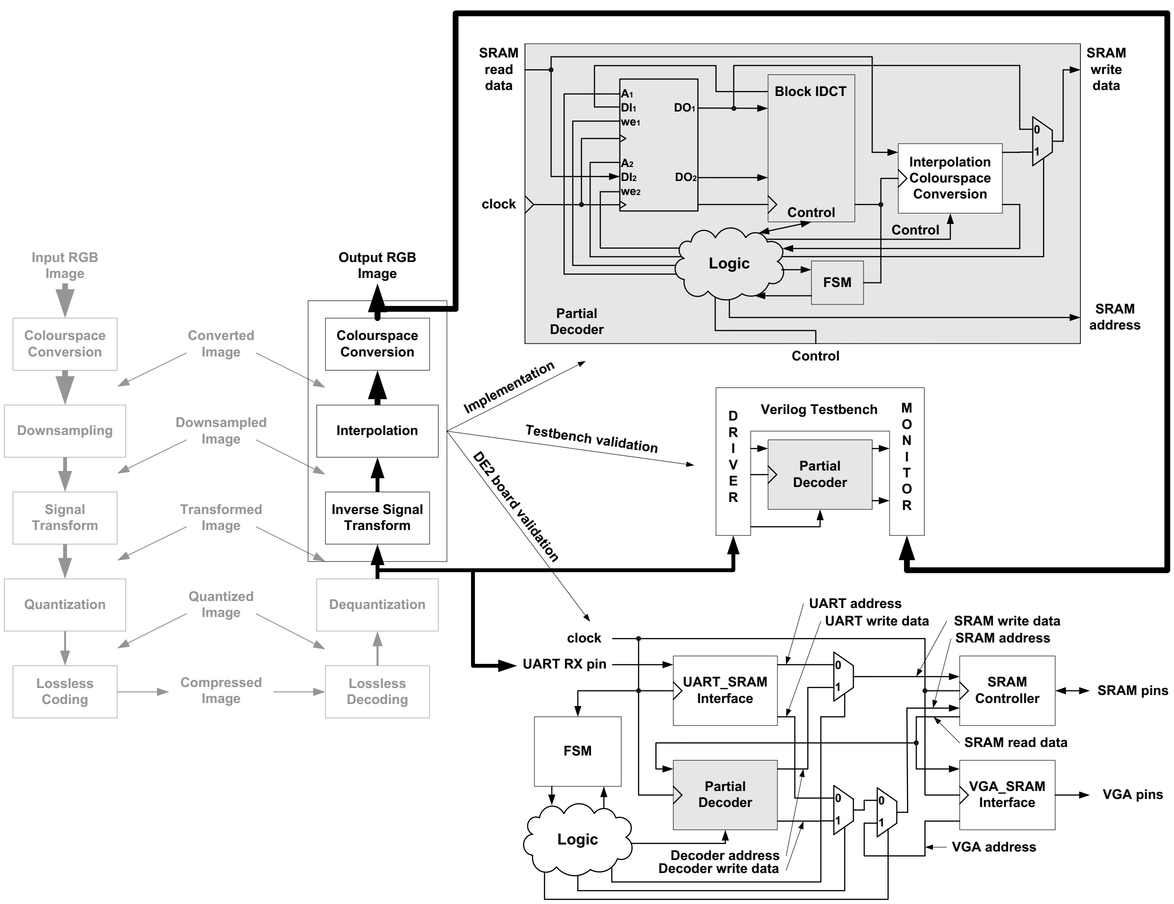


Figure 24. Overview of the IDCT implementation (milestone 2).

In Figure 25, the memory map for milestone 2 is provided, similar to that for milestone 1. The pre-IDCT data is stored as 1 value per memory location, whereas the post-IDCT data (YUV) is stored as 2 values per memory location. After IDCT is complete, the RGB data computed in milestone 1 will overwrite some pre-IDCT data (used to initialize the SRAM for milestone 2).

Like milestone 1, the blocks are divided into segments of Y, segments of (downsampled) U, and segments of (downsampled) V. However, the way the data is accessed is different. In milestone 2, we want to fetch an entire 8x8 block of pre-IDCT data, perform IDCT and write it back to the YUV memory space. To determine the addressing for writing post-IDCT samples back to the memory, observe Figure 12 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 320x240 image, and noting that we have two samples per address, our write addressing for block 0 should be: 0 … 3, …, 1120 … 1123 (as shown in Figure 26). For the second block of the first block row (Y 0,1), we have: 4 … 7, …, 1124 … 1127. 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 (38400+): 0 … 3, …, 560 … 563. The last block V 29,19 would reside in addresses (57600+): 18636 .. 18639, …, 19196 ... 19199.

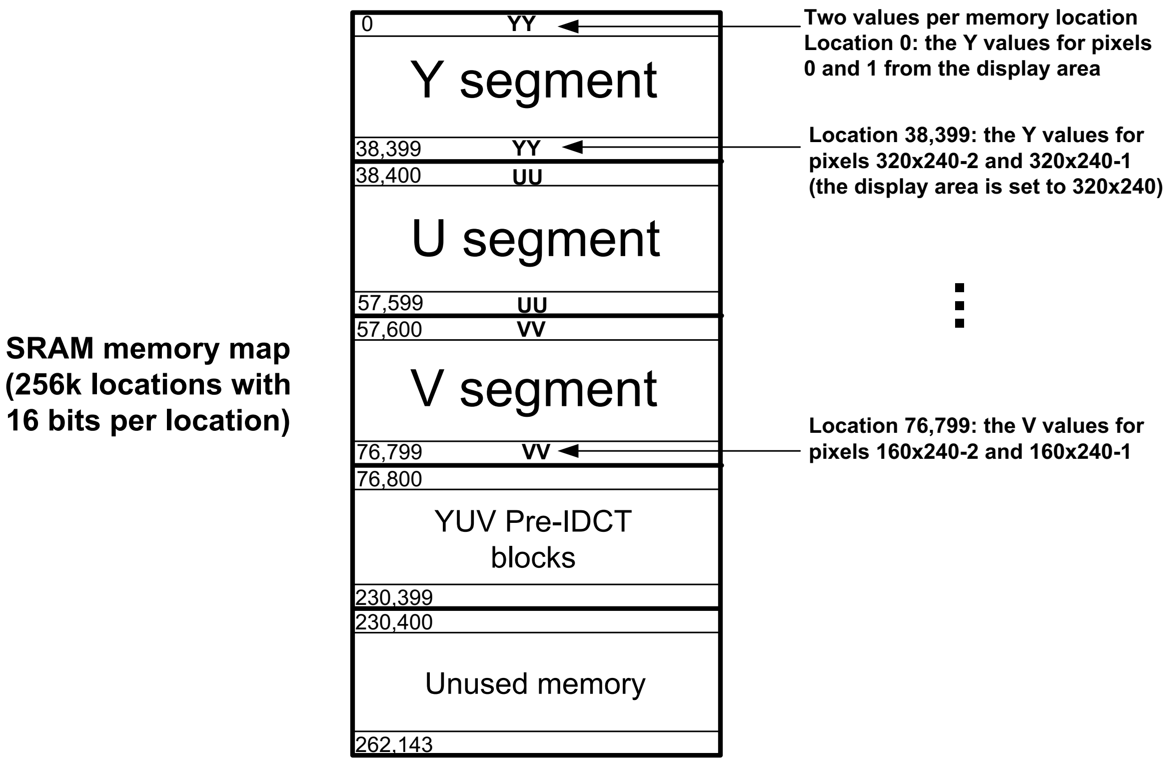


Figure 25. Memory layout for milestone 2.

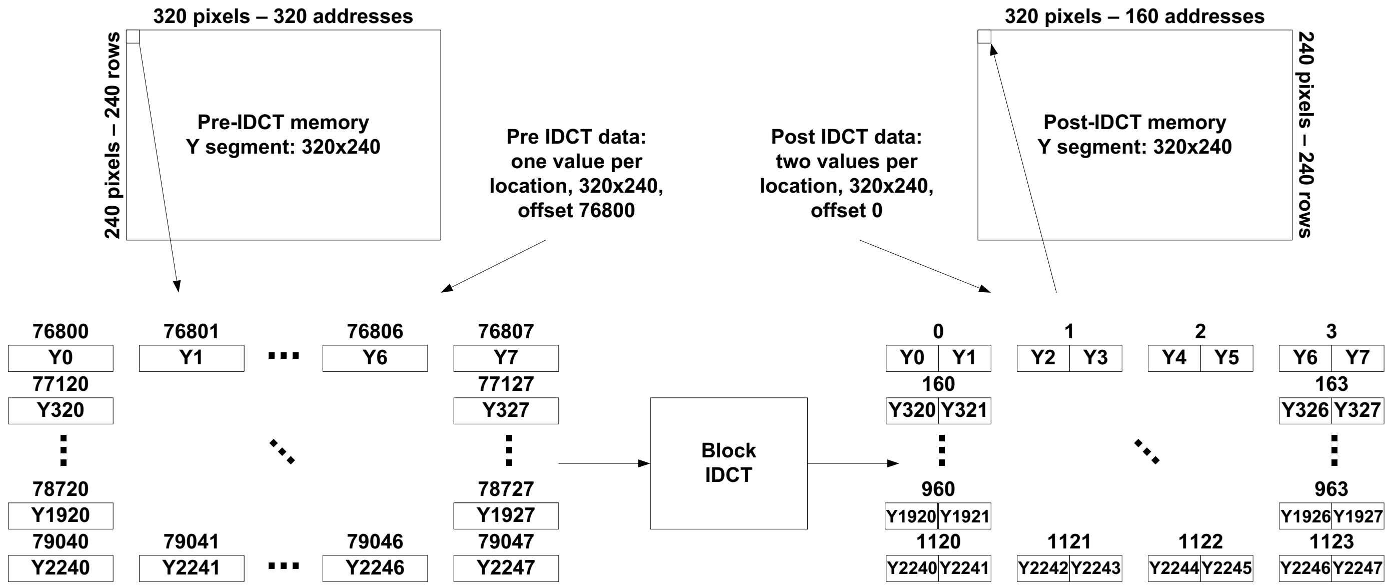


Figure 26. Reading pre-IDCT values and writing back YUV data to the external SRAM.

Like writing, the read addressing will also jump as we move from one pixel row to another within the same 8x8 block. However, now only 1 value is stored per memory location. To go across a row, the address simply increments. Within the same pixel column, to go down a pixel row (still within the same 8x8 block), the address increases by 320 (for Y, but only by 160 for U and V). Finally, we add the appropriate offset depending on our colour segment. Thus to read block Y 0,0 we use addresses (76800+): 0 … 7, …, 2240 … 2247 (as in Figure 26). Likewise to read block Y 0,1 we use (76800+): 8 … 15, …, 2248 … 2255. On the next block row, to read block Y 1,0 we use (76800+): 2560 … 2567, …, 4800 … 4807. For the block U 0,0 we use (153600+): 0 … 7, …, 1120 … 1127 (please note for U and V we have half the columns, hence there are half the number of blocks per row).

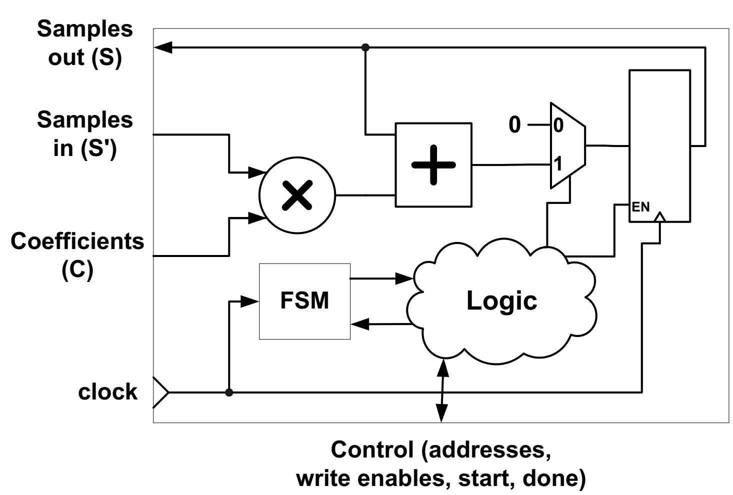


Figure 27. MAC unit for block IDCT.

Figure 27 shows a MAC unit, which may be used to calculate part of the IDCT on a single block of samples with a single multiplier. Recall that the IDCT 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 (the number of multipliers for this milestone is clarified below).

To facilitate a simple hardware implementation, we can read from dual-port memories both the fixed-point cosine coefficients (Equation 9) and the data value (pre-IDCT values or intermediate values after the first matrix multiplication). It is your task to determine a suitable memory layout for the dual-port RAMs. Because you are given this freedom, you must provide the appropriate .hex or .mif file to specify the dual-port memory initialization, particularly for the fixed-point cosine coefficients.

To maximize the utilization of the embedded memories, it is recommended that both ports are used and that the memory is organized on 32 bits per location (the maximum width allowed). It is your choice whether the results after the second matrix multiplication are written directly to the SRAM or buffered first in a dual-port RAM.

In this milestone, ***you may use 3 dual-port RAMs (of 4Kb capacity and not more than 32 bits per location), and only 3 multipliers are allowed (with input operands on 32 bits or less).*** Note also that ***the average utilization of the 3 multipliers for milestone 2 must be at least 85%.***

You are encouraged to use a state table such as the one in Table 3 during your development for each matrix multiplication for the first few and the last few clock cycles.

Clock cycle	0	1	2	3	4	5	…
SRAM_address							
SRAM_read_data							
SRAM_write_data							
SRAM_we_n							
Address_0							
Data_in_0							
Write_en_0							
Data_out_0							
Address_1							
…							
S							
S’							
C							
…							

Table 3. A starting point for the state table in milestone 2.

4.4 Milestone 3: Lossless Decoding and Dequantization

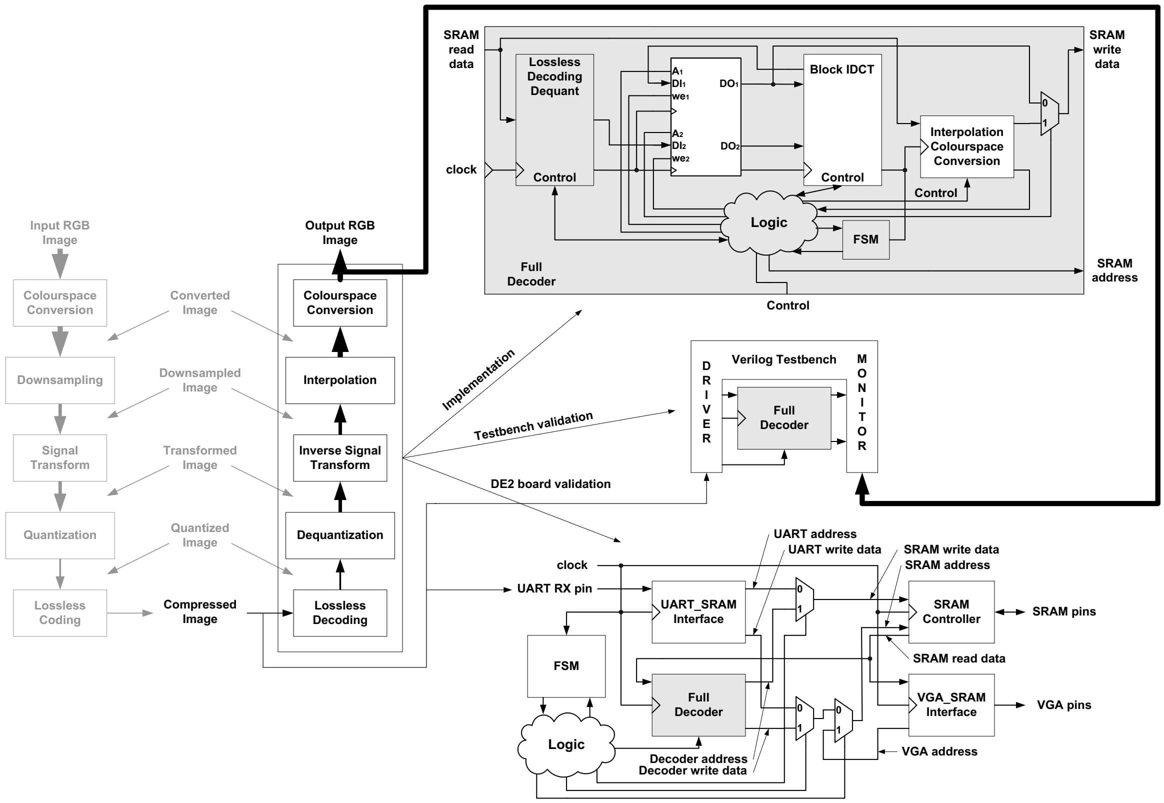


Figure 28. Lossless decoding, dequantization, and complete integration (milestone 3).

Finally, once colourspace conversion, interpolation and IDCT are complete, we can move on to lossless decoding, where the compressed bitstream itself is parsed, and the raw blocks are extracted and dequantized before being passed to the block IDCT, which completes the IDCT and writes the complete block of YUV data (belonging to the downsampled image of Figure 1) back to the SRAM. Once this milestone is complete, you will have a working image decoder.

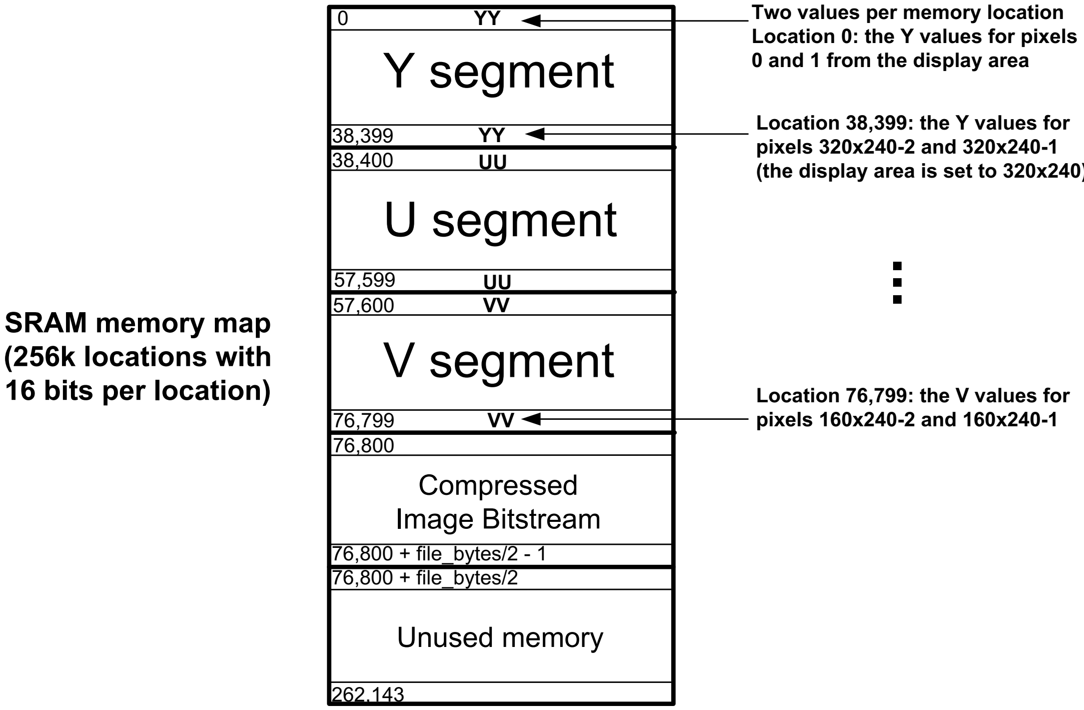


Figure 29. Memory layout for milestone 3.

Because the decoding process works on blocks of 8x8, after a block is decoded, it is immediately passed to block IDCT (milestone 2) by buffering into an embedded memory written to by milestone 3 and read from by milestone 2. The SRAM read address logic from milestone 2 will no longer be needed when milestones 2 and 3 work concurrently. Nevertheless, it is very important to note that you are strongly recommended to get milestone 3 working independently before integrating it to work concurrently with milestone 2. This will give you a chance to understand the functionality and work out the implementation details for the lossless decoder and the dequantizer, and only then can you focus on dealing with concurrency-related issues that will arise while integrating with milestone 2.

The memory layout for milestone 3 is shown above in Figure 29. No memory is allocated for the pre-IDCT data. Rather, this is processed by milestone 2 before being written back to the SRAM. After this, the RGB data generated by milestone 1 may overwrite the compressed image bitstream.

The basic decoding process is captured in Figure 8 at the beginning of the image decompression example. This flow chart can be used to construct a state table to guide your state machine implementation. Part of the challenge of lossless decoding is the implementation of a serializer, which enables the extraction of coefficient data by reading 16-bit words stored in the SRAM and delivering them to the decoder in a serial fashion. Figure 30 shows the basic architecture of the serializer. A custom FSM is required to drive the SRAM address and to increment it only if all the bits have been extracted from the current memory location. A 32-bit shift register can buffer the data on-chip, as codes and values may span two memory locations. In each clock cycle, the mux network will choose which bits from the location will be processed and shifted out of the 32-bit shift register. Note the shifting of bits in this 32-bit buffer may happen over more than one clock cycle, depending on your FSM (for example, we may shift out 6 bits by doing a shift by 3 on two consecutive clock cycles).

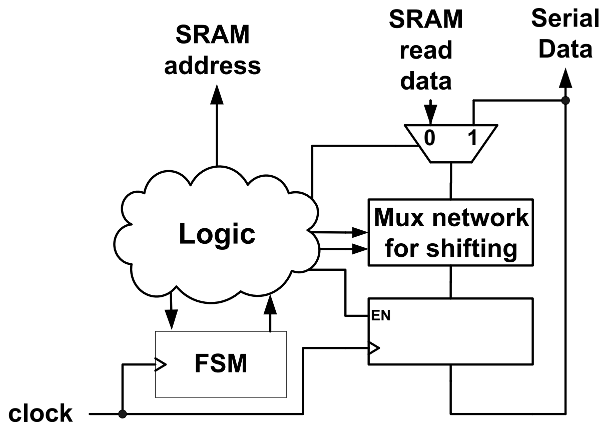


Figure 30. SRAM data is serialized for extracting coefficients.

Once the coefficients have been extracted, they must be dequantized before being placed in the dual-port RAM to await the block IDCT. Due to the particular structure of the two possible quantization matrices (as in Equation 3), which contain only powers of two, a barrel shifter structure can be used to implement the dequantization (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 values are delivered from the bitstream to the correct locations in the 8x8 block.

To facilitate easier integration between block IDCT and lossless decoding and quantization, ***you are allowed one extra dual-port RAM (of 4Kb capacity and not more than 32 bits per location). Thus a total of 4 dual-port RAMs can be used in milestones 2 and 3 together***. This extra RAM can be used for loading the out-of-order pre-IDCT values, and then during the clock cycles when milestone 2 does not need to fetch pre-IDCT values, the content from this extra embedded RAM can be loaded into the embedded RAM shared between milestones 2 and 3. You may also write directly to the embedded RAM shared between milestones 2 and 3 (without the additional embedded RAM). However, this requires more design effort to ensure no scheduling conflicts between writing from milestone 3 and reading from milestone 2.

To handle a run of zeros, there are two basic approaches. One method is to write a sequence of zeros when it happens (in different memory locations and thus over several clocks). The other method is to initialize all values to zero, and then when a run of zeros is decoded, we can skip over that many memory locations.

Finally, successful implementation of lossless decoding and dequantization relies heavily on getting your state machine to work correctly and synchronizing the signals between it and the dual-port RAMs (used for IDCT), the serializer, the top-level state machine, etc.

ENJOY!

Appendix A: Summary of Data Flow and Computation

The 3DQ5 project must compile in Quartus and it is expected to operate correctly at the 50 MHz clock frequency.

Milestone 1

Input: data is represented as ***8 bits unsigned***. Each SRAM location contains 2 values. There are separate sections for Y, U, and V. Within each section, data is stored in raster-scan order (across first, then down).

Computation: it is advised to do ***all arithmetic on 32 bits signed***. Upsample the odd values of U and V (no clipping here). Each pixel is independently converted from YUV to RGB; after this, clipping to ***8 bits unsigned*** is done. In this milestone, you **must exploit the symmetry in the interpolation filter** and use only ***3 multipliers*** with at least 75% utilization.

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

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

Output: each of R, G and B are ***8 bits unsigned***. Data for 2 pixels requires 3 SRAM locations to store. The memory layout of the RGB segment resembles lab 5 experiment 1 (see Figure 17 for starting address).

Milestone 2

Input: data is represented as ***16 bits signed***.

Computation: it is advised to do ***all arithmetic on 32 bits signed***. There are 2 matrix multiplications per output 8x8 block, after which each value is clipped to ***8 bits unsigned***. Take note of the SRAM access pattern since data is processed in blocks of 8x8. ***Up to 3 dual-port memories*** (of 4Kb capacity and not more than 32 bits per location) can be used and it is your choice on how these are organized. In this milestone, you must use ***3 multipliers*** with at least 85% utilization.

$$\boldsymbol{S} = (int)\frac{1}{65536}\left(\boldsymbol{C}^T \times (int)\frac{1}{256}(\boldsymbol{S}'\boldsymbol{C})\right)$$

Output: ***8 bits unsigned*** values are written to the SRAM, 2 values per SRAM location.

Milestone 3

Input: the .mic18 file is a ***bitstream***; to get the next bit, we go from the most significant bit to the least significant bit within the same SRAM location, then only go to the following SRAM location.

Computation: decode each 8x8 block independently. Each prefix code may produce several values. Values are processed in the zigzag order for each block. Once the position in the 8x8 block is known for each value, dequantization is applied. The quantization matrix to be used must be extracted from the .mic18 header. ***One extra dual-port memory*** (same as the 3 memories from milestone 2) may be used.

Output: each dequantized value is represented as ***16 bits signed***. Before you integrate milestone 3 into milestone 2, it is strongly recommended that you implement it separately, in which case, take note of the SRAM access pattern since data is processed in blocks of 8x8.