COE3DQ5 Project Description 2025
Hardware Implementation of an Image Decompressor

1 Introduction

The project provides a framework for developing and reinforcing core concepts in digital system design, and bridging the gap between theory and practice. Students will broaden their experience and deepen their understanding of digital systems while learning new methods by designing, implementing and verifying the decompression circuitry for the custom McMaster Image Compression (mic) revision 19 (`.mic19`) format in hardware. This format is based on the fundamental principles of the Joint Photographic Experts Group (JPEG) image compression standard [1], but it includes simplifications to make hardware implementation feasible within a five-week project timeline.

Compressed data, or intermediate data used for validation and debugging, for a 192 × 144 pixel image will be transferred to the DE2-115 board [2] from a Personal Computer (PC) via the Universal Asynchronous Receiver Transmitter (UART) interface and stored in the external Static Random-Access Memory (SRAM). The custom digital system, designed, modeled, and verified in SystemVerilog [3], will be synthesized for and mapped onto the Field-Programmable Gate Array (FPGA) on the DE2-115 board. The custom circuitry will read the compressed data from external memory, perform decompression in a sequence of steps, and store the raw pixel data back into SRAM, from which the Video Graphics Array (VGA) controller will read and display it on the monitor.

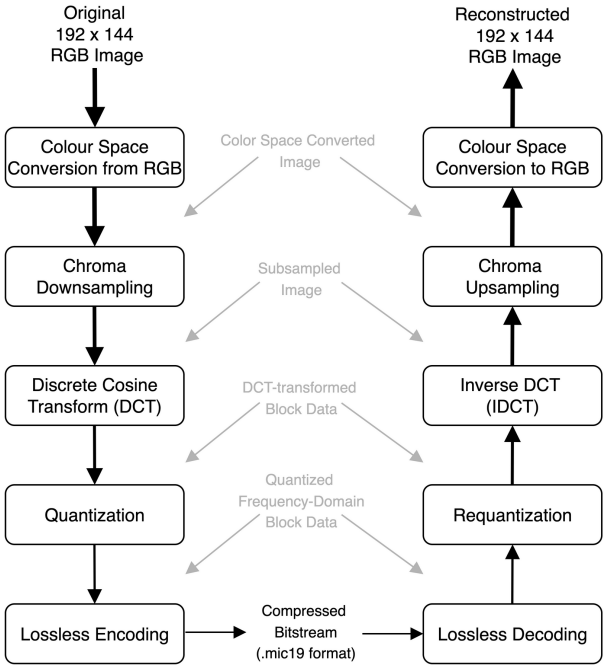


Figure 1: Flow of the coder-decoder (codec) for 192 × 144 images to and from the `.mic19` format.

The image compression pipeline (pipe) used for compressing and decompressing images of 192 × 144 resolution is shown in Figure 1. The encoding pipe, as detailed in Section 2.1, transforms a Portable Pixmap Format (PPM) image, which stores the data in uncompressed Red Green Blue (RGB) format, through five stages to the compressed bitstream in `.mic19` format: colour space conversion from RGB, chrominance (or chroma) downsampling, block-based Discrete Cosine Transform (DCT), quantization and lossless encoding. The decoding pipe, as detailed in Section 2.2, performs the computational steps that are the counterparts to the ones from the encoding stage: lossless decoding, requantization, Inverse Discrete Cosine Transform (IDCT), and chroma upsampling, and colour space conversion back to the RGB format. After decoding, the reconstructed image will not be bit-equivalent to the original one; nevertheless, it is generally perceived as indistinguishable.

From an implementation standpoint, the development of digital hardware for decoding should be organized into three milestones, as detailed in Section 3. The project guidelines begin with the software model in Section 3.1. The hardware milestones then proceed in the reverse order of the decoding pipeline: Milestone 1, which covers chroma upsampling and colour space conversion to RGB, is described in Section 3.2; Milestone 2, which covers the inverse block transform, is described in Section 3.3; and Milestone 3, which covers the requantizer and lossless decoder, is described in Section 3.4.

2 Codec Description

This section describes the codec specification, comprising two main parts: image compression and image decompression. These are detailed in Sections 2.1 and 2.2 respectively.

2.1 Image Compression

The purpose of image compression is to reduce storage and transmission requirements while maintaining indistinguishable visual quality. This subsection describes the five stages used to convert a raw 192 × 144 PPM image into the compressed `.mic19` format: (1) colour space conversion from RGB to Luminance (Y) and Chrominance (U and V) (YUV), (2) chroma downsampling, (3) block-based transformation using the two-dimensional (2D) DCT, (4) quantization, and (5) lossless encoding.

The RGB to YUV conversion separates luminance (or luma) (Y) from chroma (U, V), enabling independent processing of brightness and colour. Because the human vision is more sensitive to brightness than colour, the `.mic19` format reduces chroma resolution by downsampling the U and V channels along the horizontal axis. Each channel is then divided into fixed-size blocks and transformed using the 2D DCT, which concentrates signal energy into low-frequency coefficients [4]. The resulting coefficients are quantized, often producing long runs of zeros in the high-frequency region. These are efficiently compressed using run-length coding and a custom lossless encoding scheme, yielding the final bitstream.

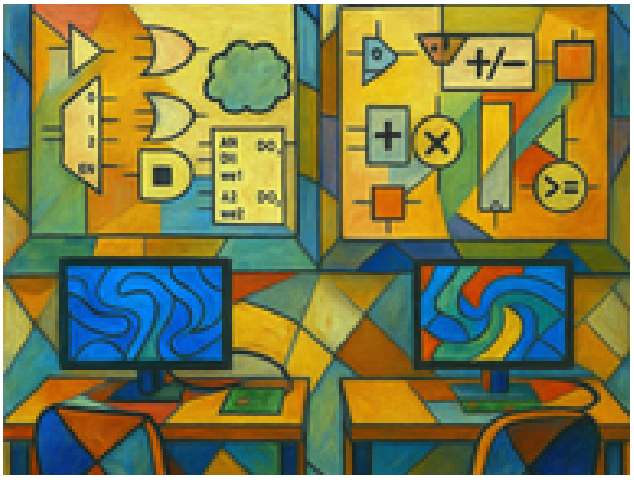


Figure 2: An example 192 × 144 image (144 rows and 192 columns) is used to illustrate the encoding and decoding process for the `.mic19` format.

On the left in Figure 2 is an artistic rendering of a computer lab. The 192 × 144 synthetic image serves as the reference example for illustrating the encoding and decoding stages of the `.mic19` codec. The source image used in the figure, generated with DALL·E in August 2025 and subsequently post-processed to achieve the final rendering, should not be interpreted as electrically accurate; the circuit symbols and diagrams are included purely as artistic elements.
A synthetic image with a diverse range of colours and fine features is useful for visually examining codec operations that act in the spatial domain, such as colour-space conversion, chroma downsampling, and the discrete cosine transform. In the frequency domain, a synthetic image can also help illustrate when and how quantization and run-length coding are more or less effective.

The following points summarize a few conventions used in this document.

- By convention, a two-dimensional array is referenced as *number of rows* × *number of columns*. The only **exception** in this document is the image resolution, which, consistent with common usage in the literature, is expressed as *width* × *height*.
- All constants are integers, and all arithmetic is performed in fixed-point 2's-complement format. Although some computational stages include a division operator for clarity in this document (for example, during scaling or quantization), the divisors are, in every case, powers of two, meaning that these operations can be implemented as right shifts.

2.1.1 Colour Space Conversion from RGB

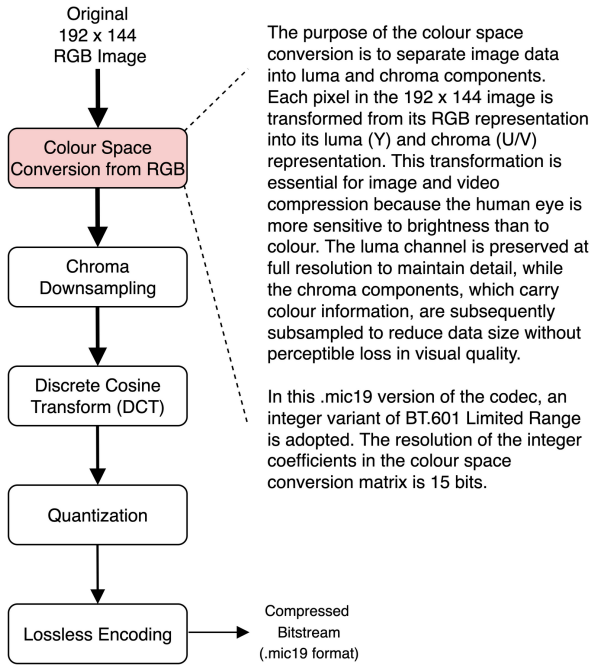


Figure 3: Colour space conversion from RGB to YUV.

The RGB to YUV conversion is the first stage of the encoding process, as illustrated in Figure 3. It separates image information into one luma component and two chroma components, allowing subsequent compression stages to exploit perceptual redundancy, since the human visual system is far less sensitive to chroma detail than to luma variations. In image and video coding standards such as JPEG [1], this transformation is formally defined in the Luma (Y), Blue-difference Chroma (Cb), Red-difference Chroma (Cr) (YCbCr) colour space; however, for simplicity and notational consistency, this document uses the YUV notation to describe the same process.
The visual effect of colour-space conversion is shown in Figure 4. To avoid ambiguity between chroma values obtained directly from colour-space conversion and those produced after chroma downsampling, the chroma components immediately after conversion are denoted by U' and V', and the downsampled chroma components by U and V.

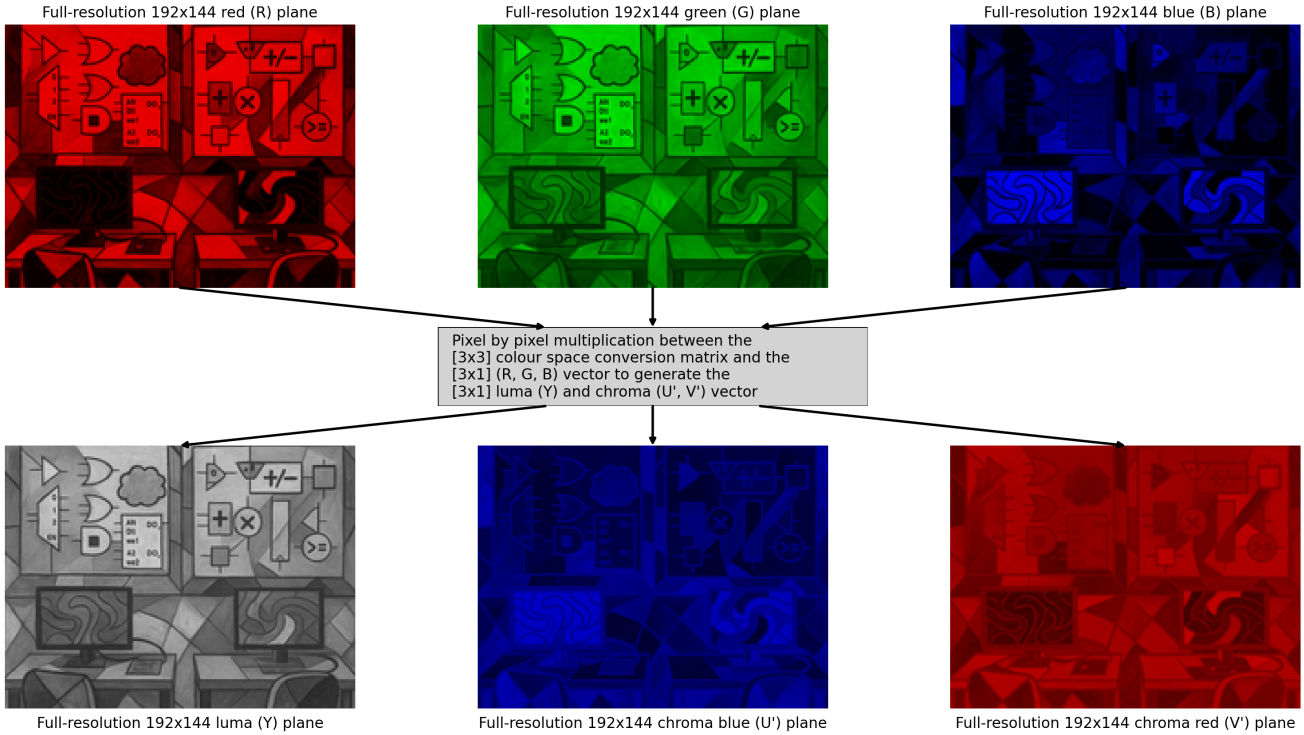


Figure 4: Conversion of the example 192 × 144 image from the RGB colour space to the YUV colour space. Each pixel's RGB triplet is transformed into a YU'V' triplet through a 3×3 matrix multiplication.

The conversion is applied pixel by pixel: each pixel's three RGB values are linearly transformed (using a 3×3 matrix and offsets) into one luma sample and two chroma samples. The equation for colour-space conversion is an integer variant of BT.601 Limited Range [5]. The real-valued coefficients are scaled by 15 bits to obtain their integer representation.

$$\begin{bmatrix} Y \\ U' \\ V' \end{bmatrix} = \frac{1}{32768} \left(\begin{bmatrix} 8421 & 16515 & 3211 \\ -4850 & -9535 & 14385 \\ 14385 & -12059 & -2326 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \right) + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \tag{1}$$

Consider, for example, the computation for the pixel at row 1 and column 86:

$$\begin{bmatrix} R_{278} \\ G_{278} \\ B_{278} \end{bmatrix} = \begin{bmatrix} 227 \\ 213 \\ 79 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} Y_{278} &= \left\lfloor \frac{8421 \cdot 227 + 16515 \cdot 213 + 3211 \cdot 79}{32768} \right\rfloor + 16 = 189 \\ U'_{278} &= \left\lfloor \frac{-4850 \cdot 227 - 9535 \cdot 213 + 14385 \cdot 79}{32768} \right\rfloor + 128 = 67 \\ V'_{278} &= \left\lfloor \frac{14385 \cdot 227 - 12059 \cdot 213 - 2326 \cdot 79}{32768} \right\rfloor + 128 = 143 \end{cases}$$

Equation 1 is applied to each of the 192×144 pixels to compute the Y, U', and V' planes shown in Figure 4, before proceeding to the next encoding step.

2.1.2 Chroma Downsampling

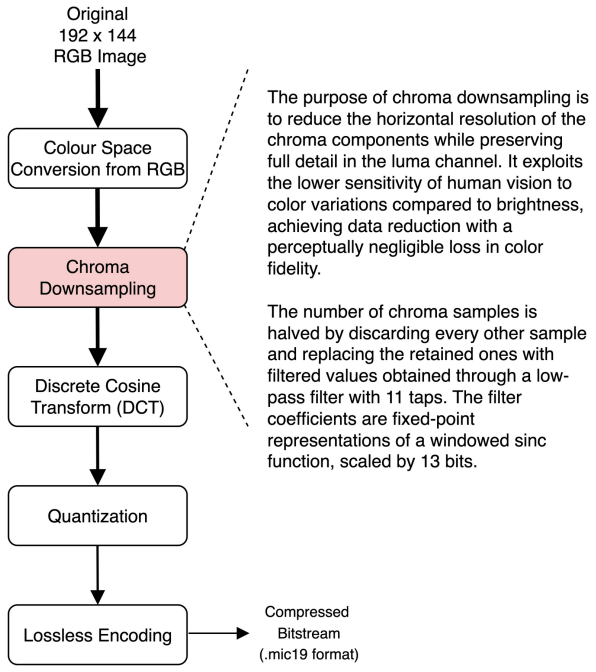


Figure 5: Chroma downsampling.

Chroma downsampling is the second stage of the encoding process, as illustrated in Figure 5. It reduces the spatial resolution of the chroma components U' and V' while preserving the full resolution of the luma component Y. In the `.mic19` standard, only horizontal downsampling is applied, where each pair of adjacent pixels in a row shares the same chroma value. This corresponds to a 4:2:2 sampling structure, which maintains full vertical resolution while halving the horizontal chroma resolution.

Although vertical chroma downsampling is commonly used in commercial image and video codecs to achieve higher data reduction, it is intentionally omitted in the `.mic19` codec to avoid additional implementation complexity. Excluding vertical filtering and its corresponding upsampling stage simplifies memory access patterns and makes the early hardware design phase more manageable to design, implement, and verify.

The equation for the downsampling filter is given below. It is based on a windowed sinc low-pass filter [6] that uses a Kaiser window [7]. The coefficients for 11 taps shown below are integer equivalents scaled by 13 bits.

$$\begin{aligned} U_j = \frac{1}{8192} \Big(& 71 \cdot U'_{2j-9} - 180 \cdot U'_{2j-7} + 360 \cdot U'_{2j-5} - 771 \cdot U'_{2j-3} + 2568 \cdot U'_{2j-1} + 4096 \cdot U'_{2j} \\ & + 2568 \cdot U'_{2j+1} - 771 \cdot U'_{2j+3} + 360 \cdot U'_{2j+5} - 180 \cdot U'_{2j+7} + 71 \cdot U'_{2j+9} \Big) \end{aligned} \tag{2}$$

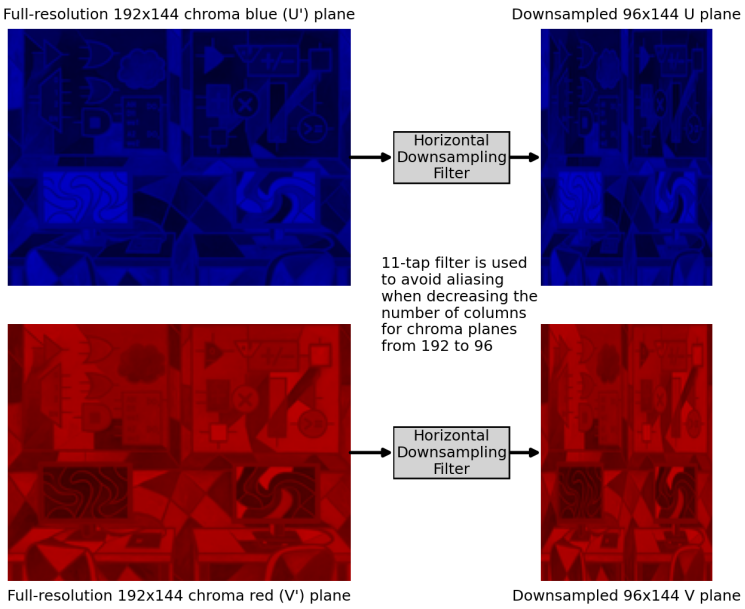


Figure 6: Downsampling from full-resolution U' and V' planes to the downsampled U and V planes.

Equation 2 is applied to all pixels in each row to compute the horizontally downsampled chroma values. The same operation is then repeated for all rows, producing the final downsampled chroma plane, as illustrated in Figure 6.
The boundary conditions are handled independently for every row, since the filter operates only along the horizontal direction. At the beginning of a row, indices $2j-9$, ..., $2j-1$ fall below the first valid position, while at the end of a row, indices $2j+1$, ..., $2j+9$ exceed the last valid position. In such cases, the out-of-range references are replaced by the nearest valid sample within that same row, effectively extending the boundary value to approximate the missing samples. The same filtering is applied to the V plane to ensure consistent chroma downsampling across both components.

Three examples are shown below. First, for a pixel in the middle of the row: U at row 1, column 43.

U'_{269}	U'_{271}	U'_{273}	U'_{275}	U'_{277}	U'_{278}	U'_{279}	U'_{281}	U'_{283}	U'_{285}	U'_{287}
69	69	66	64	66	67	62	129	140	153	163

$$U_{139} = \frac{1}{8192}\Big(71\cdot 69 - 180\cdot 69 + 360\cdot 66 - 771\cdot 64 + 2568\cdot 66 + 4096\cdot 67 + 2568\cdot 62 - 771\cdot 129 + 360\cdot 140 - 180\cdot 153 + 71\cdot 163\Big) = 61$$

Next, the boundary condition at the start of the row: U at row 1, column 0.

U'_{192}	U'_{192}	U'_{192}	U'_{192}	U'_{192}	U'_{192}	U'_{193}	U'_{195}	U'_{197}	U'_{199}	U'_{201}
124	124	124	124	124	124	146	152	151	152	106

$$U_{96} = \frac{1}{8192}\Big(71\cdot 124 - 180\cdot 124 + 360\cdot 124 - 771\cdot 124 + 2568\cdot 124 + 4096\cdot 124 + 2568\cdot 146 - 771\cdot 152 + 360\cdot 151 - 180\cdot 152 + 71\cdot 106\Big) = 128$$

Finally, the boundary condition at the end of the row: U at row 1, column 95.

U'_{373}	U'_{375}	U'_{377}	U'_{379}	U'_{381}	U'_{382}	U'_{383}	U'_{383}	U'_{383}	U'_{383}	U'_{383}
75	70	118	152	141	140	140	140	140	140	140

$$U_{191} = \frac{1}{8192}\Big(71\cdot 75 - 180\cdot 70 + 360\cdot 118 - 771\cdot 152 + 2568\cdot 141 + 4096\cdot 140 + 2568\cdot 140 - 771\cdot 140 + 360\cdot 140 - 180\cdot 140 + 71\cdot 140\Big) = 139$$

2.1.3 Discrete Cosine Transform

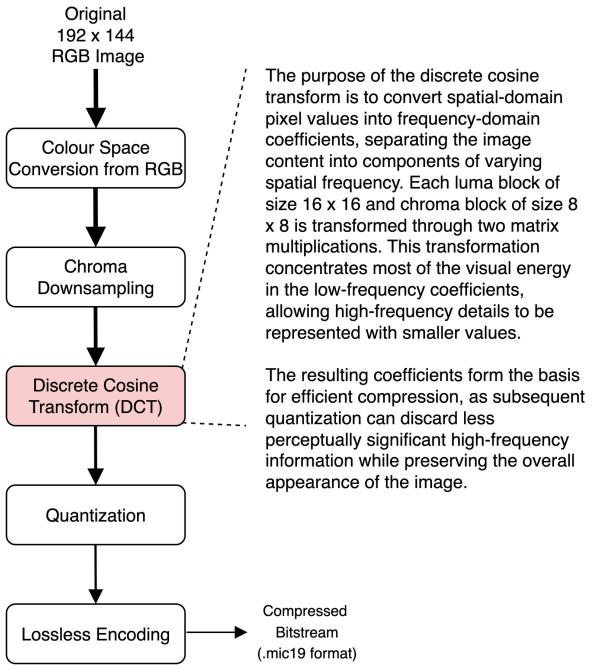


Figure 7: Discrete cosine transform.

The Discrete Cosine Transform (DCT) is the third stage of the encoding process, as illustrated in Figure 7. It converts spatial-domain data from the luma and chroma components into the frequency domain, exploiting its energy compaction property to concentrate most of the signal energy into a small number of low-frequency coefficients. This is particularly effective for natural images, where low-frequency components dominate.

In the `.mic19` codec, DCT is applied to 16 x 16 blocks for the luma plane and 8 x 8 blocks for the chroma plane. The separability of the DCT allows efficient two-dimensional computation by performing successive one-dimensional transforms along rows and columns.

The resulting concentration of energy in the low-frequency region aligns with human visual perception, providing the basis for the quantization step that follows, in which information is deliberately reduced in a controlled manner to achieve higher compression in the final stage of encoding.

The `.mic19` codec uses the DCT-II variant [4] for domain transformation between the spatial and frequency representations. The transform coefficient matrix $\mathbf{C}_N$, of dimension $N \times N$, is defined as:

$$[\mathbf{C}_N]_{i,j} = \begin{cases} (int)\left(\sqrt{\frac{1}{N}}\cos\left(\frac{\pi(j+0.5)i}{N}\right) \times 512\right) & \text{if } i = 0 \\ (int)\left(\sqrt{\frac{2}{N}}\cos\left(\frac{\pi(j+0.5)i}{N}\right) \times 512\right) & \text{otherwise} \end{cases} \tag{3}$$

where $i, j \in \{0, 1, \ldots, N-1\}$ represent the row and column indices.

In the above equation, the subscript N denotes the dimension of the square coefficient matrix. For an arbitrary input block S with N rows and M columns, the two-dimensional DCT can be expressed as:

$$S' = \mathbf{C}_N^T \times (S \times \mathbf{C}_M) \tag{4}$$

Here, $\mathbf{C}_N^T$ is the transpose of the square matrix $\mathbf{C}_N$ defined in Equation 3, and S' is the output block, also of dimension $N \times M$. The formulation from Equation 4 demonstrates the separability property of the DCT, which allows the two-dimensional transform to be implemented as two successive one-dimensional transforms. The input block S is first post-multiplied on the right by $\mathbf{C}_M$, applying the one-dimensional DCT across all rows. The resulting intermediate block is then pre-multiplied on the left by $\mathbf{C}_N^T$, which performs the transform along the columns. The total number of multiplications required is $N \times M \times M$ for the post-multiplication stage $(S \times \mathbf{C}_M)$ and $N \times N \times M$ for the pre-multiplication stage $(\mathbf{C}_N^T \times (S \times \mathbf{C}_M))$.

The analysis above shows that the arithmetic cost of the DCT is high, consistent with practice where the block transform is a major computational bottleneck in state-of-the-art codecs. Although fast DCT algorithms exist, often derived from trigonometric symmetries, this project adopts the matrix formulation from Equation 4 to emphasize the processing and transfer of two dimensional data blocks in and out of embedded memories, a concept that generalizes to other digital signal processing applications. The additional design effort required for fast DCT variants also lies beyond the project timeline.

In the `.mic19` standard, the real-valued coefficients obtained from the weighted `cos` function in Equation 3 are scaled by 9 bits to produce their integer representation. The luma plane is divided into blocks of size 16 x 16, as shown in Figure 8. Each of the 108 luma blocks is then processed by the following two-dimensional DCT operation during encoding:

$$S' = \frac{1}{8192}\left(\mathbf{C}_{16}^{T} \times \frac{1}{32}\left(S \times \mathbf{C}_{16}\right) + 4096\right) \tag{5}$$

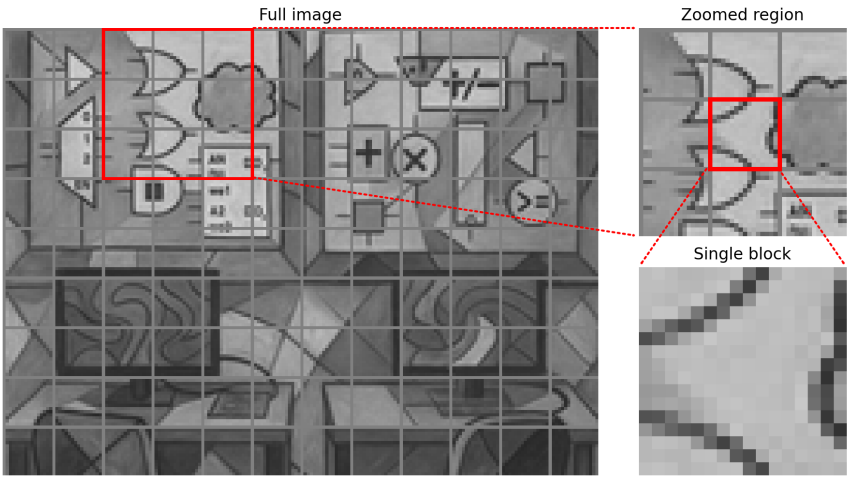


Figure 8: A luma plane partitioned into 16 × 16 blocks. For a luma plane with 144 rows and 192 columns, there are 9 rows and 12 columns of blocks, yielding a total of 108 luma blocks.

As we zoom into a specific region of the image, pixels with higher intensity appear brighter, while darker pixels represent lower intensity values. The variation in brightness between neighboring pixels within the zoomed region indicates the presence of fine spatial detail with high-frequency content, which is more difficult to compress after the transform. In contrast, areas with little variation are dominated by low-frequency content and can be represented more compactly.

The chroma planes are divided into blocks of size 8 x 8, as shown in Figure 9. Each of the 216 chroma blocks from each of the two two chroma planes is then processed by the following operation:

$$S' = \frac{1}{8192}\left(\mathbf{C}_{8}^{T} \times \frac{1}{32}\left(S \times \mathbf{C}_{8}\right) + 4096\right) \tag{6}$$

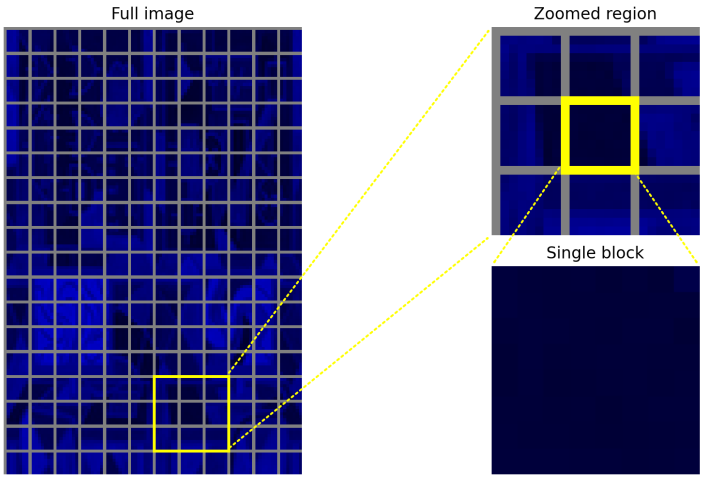


Figure 9: A chroma plane partitioned into 8 x 8 blocks. For a chroma plane with 144 rows and 96 columns, there are 18 rows and 12 columns of blocks, yielding a total of 216 chroma blocks for each of the two chroma components.

The cumulative downscaling from the two scale factors (32 and 8192) applied during post- and pre-multiplication in Equations 5 and 6 matches the upscaling from scaling each transform matrix coefficient by 9 bits.

Note that a smaller intermediate scale factor preserves higher precision, mitigating fixed-point precision loss. Note also that constant 4096 is added to each matrix element before final scaling to achieve rounding rather than truncation.

2.1.4 Quantization

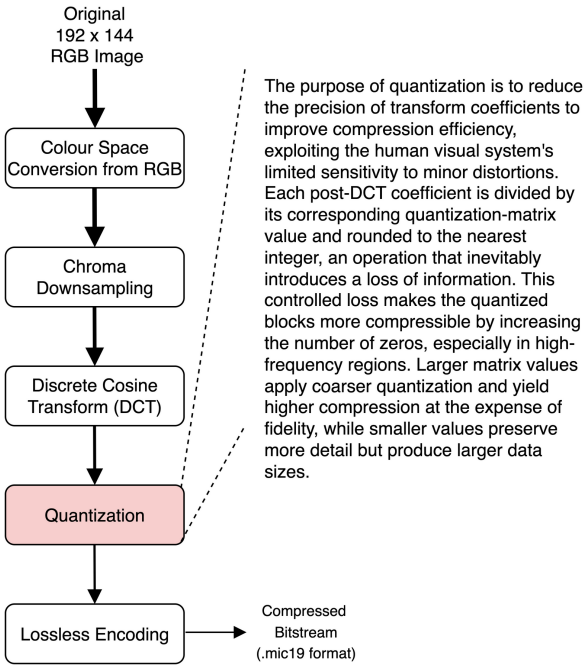


Figure 10: Quantization of post-DCT coefficients.

Quantization, illustrated in Figure 10, follows the block transform. Each luma and chroma transform coefficient is divided by the corresponding element of the quantization matrix and rounded to the nearest integer:

$$L_{i,j} = \text{round}\left(\frac{S'_{i,j}}{Q_{i,j}}\right) \tag{7}$$

where $S'_{i,j}$ denotes the post-DCT coefficients, $Q_{i,j}$ the quantization matrix, and $L_{i,j}$ the quantized coefficients. Larger $Q_{i,j}$ values, typically found in high-frequency positions, produce more zero coefficients after quantization, which can then be efficiently represented by a single escape code that encodes the variable-length run of zeros. The matrices are designed to exploit characteristics of human visual perception by prioritizing low-frequency components. Each channel uses two quantization matrices, shown for luma in Figure 11 and for chroma in Figure 12.

16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	32
16	16	16	16	16	16	16	16	16	16	16	16	16	16	32	32
16	16	16	16	16	16	16	16	16	16	16	16	16	32	32	32
16	16	16	16	16	16	16	16	16	16	16	16	32	32	32	32
16	16	16	16	16	16	16	16	16	16	16	32	32	32	32	32
16	16	16	16	16	16	16	16	16	16	32	32	32	32	32	32
16	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32
16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
16	16	16	16	16	16	16	32	32	32	32	32	32	32	32	32
16	16	16	16	16	16	32	32	32	32	32	32	32	32	32	32
16	16	16	16	16	32	32	32	32	32	32	32	32	32	32	32
16	16	16	16	32	32	32	32	32	32	32	32	32	32	32	32

(a) Luma quantization matrix 0.

16	16	16	16	16	16	32	32	32	32	32	32	32	32	32	32
16	16	16	16	16	32	32	32	32	32	32	32	32	32	32	32
16	16	16	16	32	32	32	32	32	32	32	32	32	32	32	32
16	16	16	32	32	32	32	32	32	32	32	32	32	32	32	32
16	16	32	32	32	32	32	32	32	32	32	32	32	32	32	32
16	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	64
32	32	32	32	32	32	32	32	32	32	32	32	32	32	64	64
32	32	32	32	32	32	32	32	32	32	32	32	32	64	64	64
32	32	32	32	32	32	32	32	32	32	32	32	64	64	64	64
32	32	32	32	32	32	32	32	32	32	32	64	64	64	64	64
32	32	32	32	32	32	32	32	32	32	64	64	64	64	64	64
32	32	32	32	32	32	32	32	32	64	64	64	64	64	64	64
32	32	32	32	32	32	32	32	64	64	64	64	64	64	64	64
32	32	32	32	32	32	32	64	64	64	64	64	64	64	64	64
32	32	32	32	32	32	64	64	64	64	64	64	64	64	64	64

(b) Luma quantization matrix 1.

Figure 11: Luma quantization matrices used in the `.mic19` codec.

8	8	8	8	8	8	8	16
8	8	8	8	8	8	16	16
8	8	8	8	8	16	16	16
8	8	8	8	16	16	16	16
8	8	8	16	16	16	16	32
8	8	16	16	16	16	32	32
8	16	16	16	16	32	32	32
16	16	16	16	32	32	32	32

(a) Chroma quantization matrix 0.

8	8	8	16	16	16	16	32
8	8	16	16	16	16	32	32
8	16	16	16	16	32	32	32
16	16	16	16	32	32	32	32
16	16	16	32	32	32	32	32
16	16	32	32	32	32	32	64
16	32	32	32	32	32	64	64
32	32	32	32	32	64	64	64

(b) Chroma quantization matrix 1.

Figure 12: Chroma quantization matrices used in the `.mic19` codec.

As illustrated in Figure 13, the quantization step reduces precision more aggressively at high-frequency positions (bottom-right region), where larger quantization factors produce more zeros and thereby increase compression efficiency. The resulting quantized coefficients, shown in the highlighted block on the right-hand side, represent the output of the quantization stage for channel U, block (15, 7) of size (8 x 8). The post-DCT block shown on the left-hand side of Figure 13 corresponds to the same block highlighted in Figure 9, where the pre-DCT spatial-domain pixel data were transformed into the frequency domain using Equation 6. This specific block will be used throughout the following subsections to drive the illustrative examples of the lossless encoding process (Section 2.1.5), its corresponding lossless decoding stage (Section 2.2.1), and the subsequent requantization step (Section 2.2.2).

455	0	7	-2	1	-2	0	-3
3	-1	3	-1	2	-1	2	-2
7	0	5	-4	6	1	4	0
-1	-5	6	-2	4	-1	3	-3
3	-3	0	-3	3	-5	1	-2
0	-2	1	-2	1	0	2	-1
3	-1	1	-2	-1	1	1	-1
-1	0	0	0	-1	-1	1	0

(a) Post-DCT block

8	8	8	8	8	8	8	16
8	8	8	8	8	8	16	16
8	8	8	8	8	16	16	16
8	8	8	8	16	16	16	16
8	8	8	16	16	16	16	32
8	8	16	16	16	16	32	32
8	16	16	16	16	32	32	32
16	16	16	16	32	32	32	32

(b) Quantization matrix

57	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0
0	-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	0
0	0	0	0	0	0	0	0

(c) Quantized block

Figure 13: Quantization process for channel U, block (15, 7) of size (8 x 8).

Figure 14 presents a complementary example from the Y channel, block (1, 3) of size (16 x 16), showing a dense distribution of post-DCT coefficients. In such blocks, a larger number of high-magnitude coefficients appear across both low- and high-frequency regions, indicating areas of the image that contain fine spatial detail or sharp transitions. The post-DCT block shown on the left-hand side of Figure 14 corresponds to the same block highlighted in Figure 8, where the pre-DCT spatial-domain pixel data were transformed into the frequency domain using Equation 5. After quantization, as shown on the right-hand side of Figure 14, the resulting block contains fewer and shorter zero runs, which reduces the effectiveness of run-length compression in the lossless stage. This illustrates that dense blocks representing regions rich in detail are generally less compressible than sparse ones.

2632	44	-139	160	-170	94	-28	-1	-22	19	4	-20	8	4	-7	1
19	23	18	23	-34	39	-38	41	-54	48	-45	35	-26	18	-8	1
-8	-24	246	-17	-34	33	26	2	-57	73	-48	17	-16	19	-11	4
-17	36	-43	-9	2	-14	45	-54	66	-53	49	-34	20	-10	2	1
117	122	100	-130	-49	8	63	-70	13	1	8	-35	22	-2	-3	3
-19	-40	-38	-7	-12	36	-42	25	-28	29	-21	11	2	-11	7	-5
24	-50	-61	-141	38	65	37	-81	18	18	-11	-9	7	14	-19	11
-10	-42	24	21	58	-1	-6	-11	16	-3	-6	12	-8	6	-10	4
-42	-22	-64	25	70	69	-47	-53	40	21	-2	-32	47	-23	11	-7
36	38	64	29	7	-42	-21	-1	15	-15	-3	-7	1	-3	5	3
2	22	27	50	35	-10	-66	-2	31	23	-49	4	6	1	-9	6
14	-22	-3	-60	-43	-56	26	7	21	-22	13	-3	4	-5	1	1
7	6	12	-5	-15	-45	-23	20	36	-17	-30	16	17	-4	-12	8
-25	-23	-32	-10	5	43	41	16	-18	-18	12	3	7	-8	4	-1
-2	0	2	-7	-8	-9	15	28	10	-23	-16	18	6	-10	-6	6
3	14	13	28	31	39	17	-2	-22	-4	2	5	-4	-1	0	3

(a) Post-DCT block

16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	32
16	16	16	16	16	16	16	16	16	16	16	16	16	16	32	32
16	16	16	16	16	16	16	16	16	16	16	16	16	32	32	32
16	16	16	16	16	16	16	16	16	16	16	16	32	32	32	32
16	16	16	16	16	16	16	16	16	16	16	32	32	32	32	32
16	16	16	16	16	16	16	16	16	16	32	32	32	32	32	32
16	16	16	16	16	16	16	16	16	32	32	32	32	32	32	32
16	16	16	16	16	16	16	16	32	32	32	32	32	32	32	32
16	16	16	16	16	16	32	32	32	32	32	32	32	32	32	32
16	16	16	16	32	32	32	32	32	32	32	32	32	32	32	32

(b) Quantization matrix

165	3	-9	10	-11	6	-2	0	-1	1	0	-1	1	0	0	0
1	1	1	1	-2	2	-2	3	-3	3	-3	2	-2	1	0	0
0	-1	15	-1	-2	2	2	0	-4	5	-3	1	-1	1	-1	0
-1	2	-3	-1	0	-1	3	-3	4	-3	3	-2	1	-1	0	0
7	8	6	-8	-3	1	4	-4	1	0	1	-2	1	0	0	0
-1	-2	-2	0	-1	2	-3	2	-2	2	-1	1	0	-1	0	0
2	-3	-4	-9	2	4	2	-5	1	1	-1	-1	0	0	-1	0
-1	-3	2	1	4	0	0	-1	1	0	0	1	0	0	0	0
-3	-1	-4	2	4	4	-3	-3	3	1	0	-1	1	-1	0	0
2	2	4	2	0	-3	-1	0	1	-1	0	0	0	0	0	0
0	1	2	3	2	-1	-4	0	2	1	-2	0	0	0	0	0
1	-1	0	-4	-3	-3	2	0	1	-1	0	0	0	0	0	0
0	0	1	0	-1	-3	-1	1	1	-1	-1	1	1	0	0	0
-2	-1	-2	-1	0	3	1	1	-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	-1	0	1	0	0	0	0
0	1	1	2	1	1	1	0	-1	0	0	0	0	0	0	0

(c) Quantized block

Figure 14: Quantization process for a denser block in channel Y, block (1, 3) of size (16 x 16).

2.1.5 Lossless Encoding

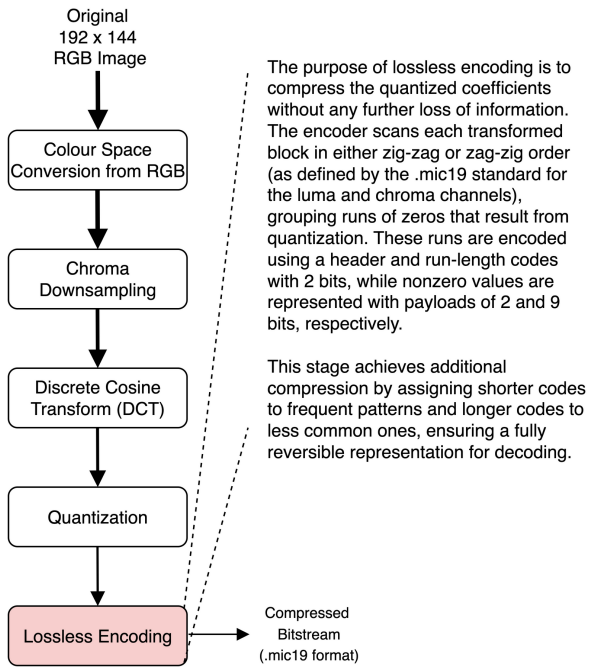


Figure 15: Lossless encoding of quantized coefficients.

As shown in Figure 15, lossless encoding is the final encoding step. Quantized transform blocks for both luma and chroma components are traversed according to a predefined order. In the `.mic19` codec, each channel uses either the *zig-zag* or the *zag-zig* scan order: *luma* channel blocks of size 16 x 16, use the *zig-zag* order, as shown on the left-hand side of Figure 16; *chroma* channel blocks of size 8 x 8, use the *zag-zig* order, as shown on the right-hand side of Figure 16. The purpose of traversing the quantized coefficients in a zig-zag or zag-zig pattern is to group low-frequency terms, which contain most of the visual detail, before the high-frequency terms. High-frequency coefficients, concentrated in the bottom-right region of the block, often quantize to zero. This ordering places long zero runs near the end of the block, making them easier to encode. After traversal, the coefficients are serialized and mapped onto variable-length codewords according to the custom encoding scheme described in Table 1.

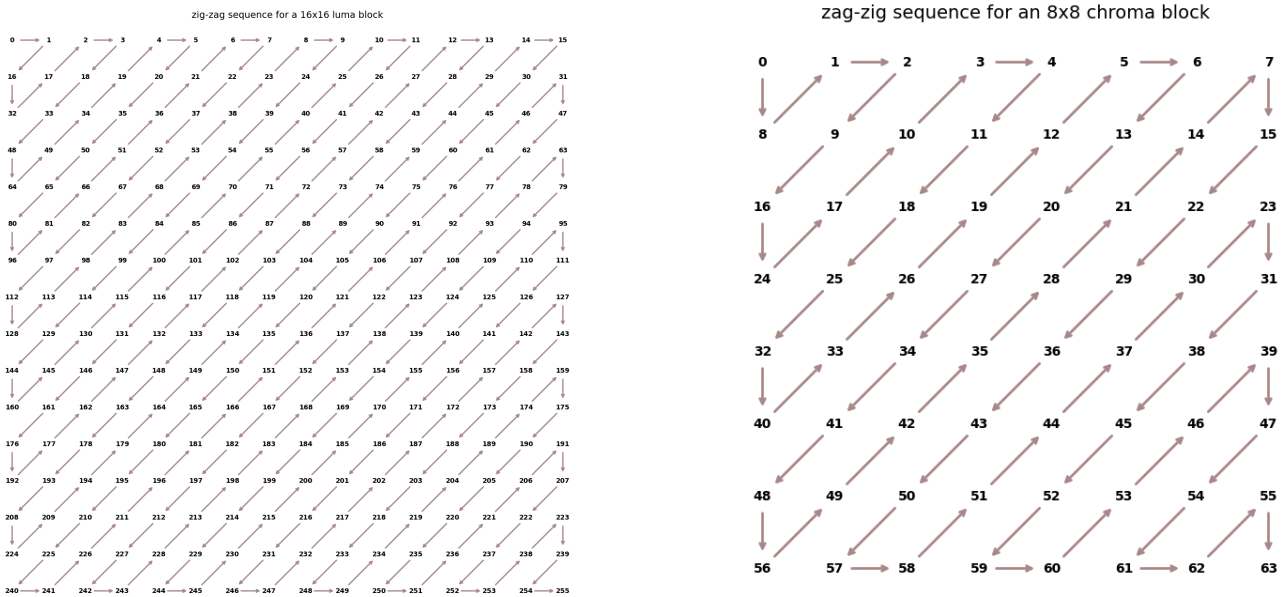


Figure 16: Traversal patterns for quantized transform blocks in the `.mic19` codec.

The lossless encoder used in the `.mic19` format relies on a header prefix to distinguish among four symbol types: runs of zeros, small coefficients, large coefficients, and an end-of-block (EOB) marker. Each header is followed by a variable-length payload field. As summarized in Table 1, the zero-run symbols use a 2-bit payload field that encodes the run length. Small coefficients use 2 payload bits, covering the range from -2 to 1, while large coefficients use 9 bits, covering -256 to 255. The EOB marker contains no

payload bits and signals that all samples until the end of the current block are zeros.

Symbol Type	Header Code	Payload Bits	Description
Run of Zeros	00	2	Payload encodes run length; value 0 means 4 zeros
Small Value	01	2	Small coefficient from -2 to 1
Large Value	10	9	Large coefficient from -256 to 255
End of Block	11	0	Run of zeros until the end of the current block

Table 1: Lossless encoding codeword mapping for the `.mic19` codec.

The following example illustrates the encoding process for channel U, block (15, 7) of size (8 x 8) in the `.mic19` codec. The input to the encoding process is the quantized block from the right-hand side of Figure 13. The block is traversed in the zag-zig order defined by the `.mic19` standard for the chroma channel. The bitstream generation steps are shown in Table 2.

Description	Header (bits)	Data (bits)	Encoded bits
Large value 57 from position (0, 0)	Header 10 on 2 bits	Data 000111001 on 9 bits	Encoded 10000111001 on 11 bits
Zero run of length 2 starting at (1, 0)	Header 00 on 2 bits	Data 10 on 2 bits	Encoded 0010 on 4 bits
Small value 1 from position (0, 2)	Header 01 on 2 bits	Data 01 on 2 bits	Encoded 0101 on 4 bits
Zero run of length 1 starting at (1, 1)	Header 00 on 2 bits	Data 01 on 2 bits	Encoded 0001 on 4 bits
Small value 1 from position (2, 0)	Header 01 on 2 bits	Data 01 on 2 bits	Encoded 0101 on 4 bits
Zero run of length 4 starting at (3, 0)	Header 00 on 2 bits	Data 00 on 2 bits	Encoded 0000 on 4 bits
Zero run of length 2 still from (3, 0)	Header 00 on 2 bits	Data 10 on 2 bits	Encoded 0010 on 4 bits
Small value 1 from position (2, 2)	Header 01 on 2 bits	Data 01 on 2 bits	Encoded 0101 on 4 bits
Small value -1 from position (3, 1)	Header 01 on 2 bits	Data 11 on 2 bits	Encoded 0111 on 4 bits
Zero run of length 3 starting at (4, 0)	Header 00 on 2 bits	Data 11 on 2 bits	Encoded 0011 on 4 bits
Small value 1 from position (3, 2)	Header 01 on 2 bits	Data 01 on 2 bits	Encoded 0101 on 4 bits
Zero run of length 4 starting at (2, 3)	Header 00 on 2 bits	Data 00 on 2 bits	Encoded 0000 on 4 bits
Zero run of length 1 still from (2, 3)	Header 00 on 2 bits	Data 01 on 2 bits	Encoded 0001 on 4 bits
Small value 1 from position (2, 4)	Header 01 on 2 bits	Data 01 on 2 bits	Encoded 0101 on 4 bits
EOB at position (3, 3)	Header 11 on 2 bits	No data to be encoded	Encoded 11 on 2 bits

Table 2: Bitstream generation steps for channel U, block (15, 7) of size (8 x 8).

The bitstreams for all encoded blocks (first all **Y**, then all **U** and **V**) are concatenated sequentially to form the final `.mic19` bitstream. This bitstream is placed immediately after a fixed-size header, which provides the decoder with essential information about the image and encoding parameters.

Byte Offset	Field	Length (bytes)	Description
0–1	`Year`	2	Year of the codec specification (2025).
2	`Version`	1	Codec version number (19, lower 6 bits are used).
3	`Quant_index`	1	Quantization index (defined during encoding, lowest bit is used).
4–5	`Height`	2	Encoded image height in pixels (144, big-endian).
6–7	`Width`	2	Encoded image width in pixels (192, big-endian).
8–19	`Reserved`	12	Reserved (not relevant to this project).

Table 3: Structure of the `.mic19` file header.

Only one field from the header is required during decoding: the `Quant_index`. It is located at **byte offset 3**, **bit 0**. All other fields are used for validation and metadata identification by the software model and are inconsequential for the hardware implementation.

2.2 Image Decompression

The purpose of image decompression is to reconstruct a visually accurate image from its compressed representation while maintaining consistency with the original scene content. This subsection describes the five computational stages used to recover an image of resolution 192 × 144 from a `.mic19` bitstream: (1) lossless decoding, (2) requantization, (3) inverse block transform using the IDCT, (4) chroma upsampling, and (5) colour space conversion from YUV back to RGB.

The process begins with *lossless decoding*, which reconstructs the quantized transform coefficients exactly as they were before lossless encoding. These coefficients are then passed to the *requantization* stage, where they are rescaled by the corresponding quantization matrices to approximate the original transform-domain values. In the next step, the *inverse block transform* converts these frequency-domain coefficients back into the spatial domain, recreating the pixel-level structure for each luma and chroma block. Once the blocks are reconstructed, *chroma upsampling* restores the full horizontal resolution of the colour components, aligning their width with the luma channel. Finally, the *colour space conversion* transforms the YUV pixel representation back into the RGB domain using the inverse of the matrix and offset parameters applied during encoding. Together, these stages produce an approximation of the original image, with differences arising primarily from the irreversible information loss introduced during chroma downsampling and upsampling, and during quantization and requantization.

In the following, each of these five stages is elaborated in its own subsection.

2.2.1 Lossless Decoding

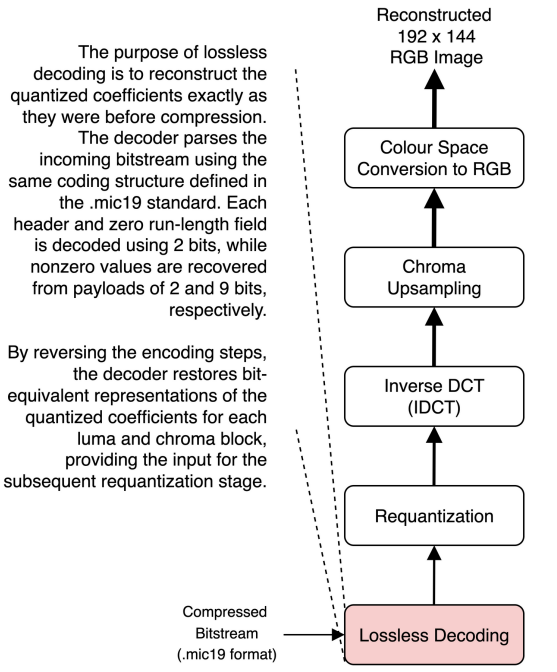


Figure 17: Lossless decoding of quantized block without information loss.

As shown in Figure 17, lossless decoding is the first decoding step and reverses the operations performed during lossless encoding. Variable-length codewords from the compressed bitstream are parsed and mapped back to quantized coefficient values according to the inverse of the custom encoding scheme described in Table 1. These decoded coefficients are then deserialized and placed into transform blocks following the same traversal patterns used during encoding: *luma* channel blocks of size 16 x 16, use the *zig-zag* order; *chroma* channel blocks of size 8 x 8, use the *zag-zig* order. All luma blocks are processed first, followed by the chroma blocks.

This reconstruction process recovers the quantized transform coefficients in their original block structure without any information loss, preparing them for subsequent requantization and inverse transform operations.

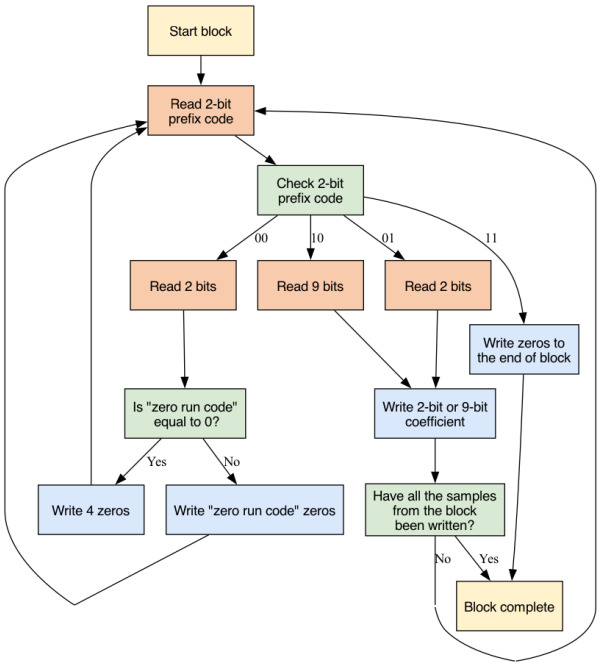


Figure 18: Lossless decoding process for the `.mic19` codec.

The decoder reconstructs the quantized coefficients from the variable-length bitstream using the same mapping as the encoder (see Table 1). Each header (prefix code) determines whether the subsequent bits represent a coefficient (small or large) or a zero run. The procedure is illustrated in Figure 18.
After skipping the 20-byte header in the `.mic19` bitstream, all luma blocks are processed before the chroma blocks. The losslessly decoded samples are *inserted* into the decoded block at the next position according to the traversal order: *zig-zag* for the luma blocks of size 16 x 16, and *zag-zig* for the chroma blocks of size 8 x 8. A new block begins processing after all samples in the current block have been inserted (256 for luma blocks and 64 for chroma blocks).

Using the diagram from Figure 18, Table 4 shows the lossless decoding for the quantized block highlighted on the right-hand side of Figure 13 and for which lossless encoding was illustrated in Table 2.

Description	**Header (bits)**	**Data (bits)**	**Decoded output**
Large value 57	Header 10 (2 bits)	Data 000111001	Large value: 57 - inserted at (0, 0)
Zero run of length 2	Header 00 (2 bits)	Data 10	Inserted 2 zeros starting at (1, 0)
Small value 1	Header 01 (2 bits)	Data 01	Small value: 1 - inserted at (0, 2)
Zero run of length 1	Header 00 (2 bits)	Data 01	Inserted 1 zeros starting at (1, 1)
Small value 1	Header 01 (2 bits)	Data 01	Small value: 1 - inserted at (2, 0)
Zero run of length 4	Header 00 (2 bits)	Data 00	Inserted 4 zeros starting at (3, 0)
Zero run of length 2	Header 00 (2 bits)	Data 10	Inserted 2 zeros starting at (0, 4)
Small value 1	Header 01 (2 bits)	Data 01	Small value: 1 - inserted at (2, 2)
Small value -1	Header 01 (2 bits)	Data 11	Small value: -1 - inserted at (3, 1)
Zero run of length 3	Header 00 (2 bits)	Data 11	Inserted 3 zeros starting at (4, 0)
Small value 1	Header 01 (2 bits)	Data 01	Small value: 1 - inserted at (3, 2)
Zero run of length 4	Header 00 (2 bits)	Data 00	Inserted 4 zeros starting at (2, 3)
Zero run of length 1	Header 00 (2 bits)	Data 01	Inserted 1 zeros starting at (1, 5)
Small value 1	Header 01 (2 bits)	Data 01	Small value: 1 - inserted at (2, 4)
Zero run of length 40	Header 11 (2 bits)	No data	Inserted 40 zeros starting at (3, 3)

Table 4: Mapping of bitstream codewords to decoded samples for channel U, block (15, 7) of size (8 x 8).

For the luma channel, the total number of blocks is given by:

$$\frac{144 \times 192}{16 \times 16} = 108$$

For each chroma channel (U and V), the total number of blocks is:

$$\frac{144 \times 96}{8 \times 8} = 216$$

After all 108 luma blocks and 216 $\times$ 2 chroma blocks have been processed, the lossless decoder should reach the end of the `.mic19` bitstream.

2.2.2 Requantization

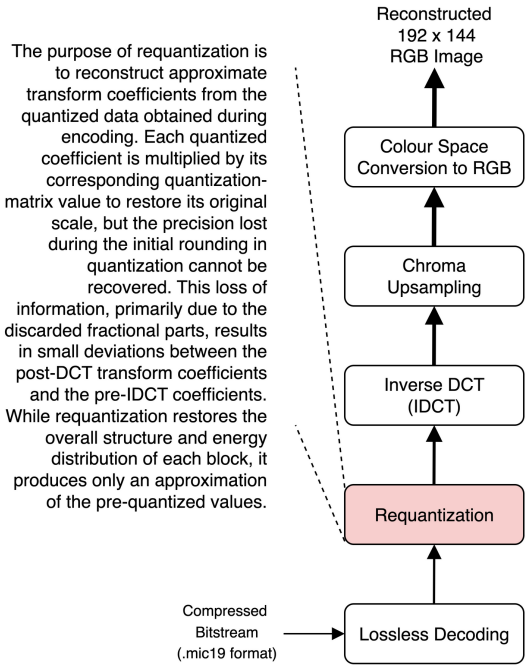


Figure 19: Requantization.

Requantization, illustrated in Figure 19, is the inverse operation of quantization and precedes the inverse block transform. Each decoded coefficient is multiplied by the corresponding element of the quantization matrix to reconstruct an approximation of the original post-DCT coefficient:

$$S'_{i,j} = L_{i,j} \times Q_{i,j} \tag{8}$$

where $L_{i,j}$ denotes the losslessly decoded quantized coefficients, $Q_{i,j}$ the quantization matrix, and $S'_{i,j}$ the requantized coefficients. The quantization matrices for both luma and chroma are specific to the `.mic19` standard and are identical to those used during encoding (see Figures 11 and 12). One particular point worth noting is that, although the 20-byte header from the `.mic19` file is not used during lossless decoding, the quantization index stored in the header (**byte offset 3**, **bit 0**, as documented at the end of Section 2.1.5), must be extracted to select the correct quantization matrix during requantization.

Using channel U, block (15, 7) of size (8 x 8) as an example, as illustrated in Figure 20, the lossless decoding stage reconstructs the quantized post-DCT coefficients shown in the highlighted block on the left-hand side of the figure. These coefficients correspond exactly to the quantized block produced during the encoding process (see Figure 13). The same quantization matrix is then applied to restore the approximate pre-IDCT values by multiplying each coefficient by its corresponding quantization factor, producing the requantized block on the right-hand side.

It is important to note that the requantized block represents only an approximation of the original post-DCT coefficients before quantization. The differences between the pre-IDCT coefficients (shown below on the right-hand side of Figure 20) and the original post-DCT values (shown on the left-hand side of Figure 13) reflect the irreversible information loss introduced during the quantization stage. Although these numerical discrepancies may appear noticeable in the coefficient domain, once the inverse block transform is applied to reconstruct each block, the resulting visual artifacts are typically imperceptible to the human eye.

57	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0
0	-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	0
0	0	0	0	0	0	0	0

(a) Lossless decoded block

8	8	8	8	8	8	8	16
8	8	8	8	8	8	16	16
8	8	8	8	8	16	16	16
8	8	8	8	16	16	16	16
8	8	8	16	16	16	16	32
8	8	16	16	16	16	32	32
8	16	16	16	16	32	32	32
16	16	16	16	32	32	32	32

(b) Quantization matrix

456	0	8	0	0	0	0	0
0	0	0	0	0	0	0	0
8	0	8	0	8	0	0	0
0	-8	8	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

(c) Requantized (pre-IDCT) block

Figure 20: Requantization process for channel U, block (15, 7) of size (8 x 8).

2.2.3 Inverse Discrete Cosine Transform

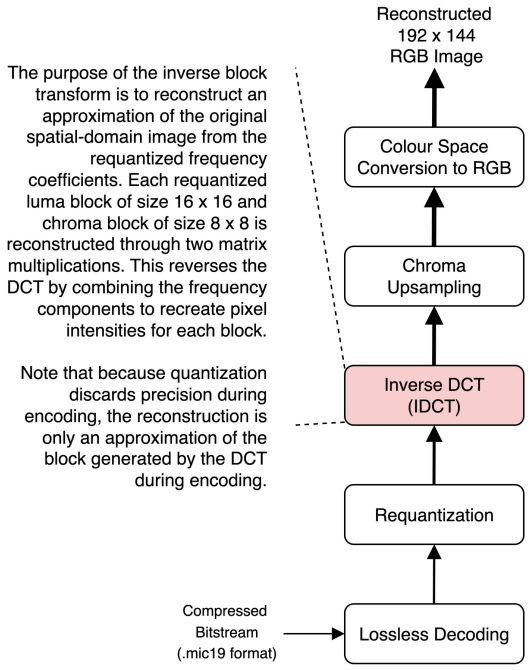


Figure 21: Inverse discrete cosine transform.

The Inverse Discrete Cosine Transform (IDCT) is the third stage of the decoding process, as illustrated in Figure 21. It reverses the DCT by redistributing the energy from the frequency domain back into the spatial domain. In the `.mic19` codec, the IDCT operates on blocks of size 16 x 16 for the **luma** plane, as shown in Equation 9:

$$S = \frac{1}{8192}\left(\mathbf{C}_{16}^{T} \times \frac{1}{32}\left(S' \times \mathbf{C}_{16}\right) + 4096\right) \tag{9}$$

For the **chroma** plane, the IDCT operates on blocks of size 8 x 8, as shown in Equation 10:

$$S = \frac{1}{8192}\left(\mathbf{C}_{8}^{T} \times \frac{1}{32}\left(S' \times \mathbf{C}_{8}\right) + 4096\right) \tag{10}$$

The input to the IDCT consists of frequency-domain blocks S' produced by the requantizer. For the luma plane, both the input S' and the reconstructed spatial-domain block S have dimensions 16 x 16. For the chroma planes, S' and S have dimensions 8 x 8.

The coefficient matrices used in Equations 9 and 10 correspond to those defined for the DCT in Equation 4 and are provided below for convenience.

$$\mathbf{C}_{16} = \begin{bmatrix}
128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 \\
180 & 173 & 159 & 139 & 114 & 85 & 52 & 17 & -17 & -52 & -85 & -114 & -139 & -159 & -173 & -180 \\
177 & 150 & 100 & 35 & -35 & -100 & -150 & -177 & -177 & -150 & -100 & -35 & 35 & 100 & 150 & 177 \\
173 & 114 & 17 & -85 & -159 & -180 & -139 & -52 & 52 & 139 & 180 & 159 & 85 & -17 & -114 & -173 \\
167 & 69 & -69 & -167 & -167 & -69 & 69 & 167 & 167 & 69 & -69 & -167 & -167 & -69 & 69 & 167 \\
159 & 17 & -139 & -173 & -52 & 114 & 180 & 85 & -85 & -180 & -114 & 52 & 173 & 139 & -17 & -159 \\
150 & -35 & -177 & -100 & 100 & 177 & 35 & -150 & -150 & 35 & 177 & 100 & -100 & -177 & -35 & 150 \\
139 & -85 & -173 & 17 & 180 & 52 & -159 & -114 & 114 & 159 & -52 & -180 & -17 & 173 & 85 & -139 \\
128 & -128 & -128 & 127 & 128 & -127 & -127 & 127 & 127 & -127 & -127 & 127 & 128 & -127 & -128 & 127 \\
114 & -159 & -52 & 180 & -17 & -173 & 85 & 139 & -139 & -85 & 173 & 17 & -180 & 52 & 159 & -114 \\
100 & -177 & 35 & 150 & -150 & -35 & 177 & -100 & -100 & 177 & -35 & -150 & 150 & 35 & -177 & 100 \\
85 & -180 & 114 & 52 & -173 & 139 & 17 & -159 & 159 & -17 & -139 & 173 & -52 & -114 & 180 & -85 \\
69 & -167 & 167 & -69 & -69 & 167 & -167 & 69 & 69 & -167 & 167 & -69 & -69 & 167 & -167 & 69 \\
52 & -139 & 180 & -159 & 85 & 17 & -114 & 173 & -173 & 114 & -17 & -85 & 159 & -180 & 139 & -52 \\
35 & -100 & 150 & -177 & 177 & -150 & 100 & -35 & -35 & 100 & -150 & 177 & -177 & 150 & -100 & 35 \\
17 & -52 & 85 & -114 & 139 & -159 & 173 & -180 & 180 & -173 & 159 & -139 & 114 & -85 & 52 & -17
\end{bmatrix}$$

$$\mathbf{C}_{8} = \begin{bmatrix}
181 & 181 & 181 & 181 & 181 & 181 & 181 & 181 \\
251 & 212 & 142 & 49 & -49 & -142 & -212 & -251 \\
236 & 97 & -97 & -236 & -236 & -97 & 97 & 236 \\
212 & -49 & -251 & -142 & 142 & 251 & 49 & -212 \\
181 & -181 & -181 & 181 & 181 & -181 & -181 & 181 \\
142 & -251 & 49 & 212 & -212 & -49 & 251 & -142 \\
97 & -236 & 236 & -97 & -97 & 236 & -236 & 97 \\
49 & -142 & 212 & -251 & 251 & -212 & 142 & -49
\end{bmatrix}$$

Based on the same principle detailed in Section 2.1.3 for the DCT, each of the 108 pre-IDCT luma blocks is processed using Equation 9, and each pre-IDCT chroma block from the two sets of 216 chroma blocks (corresponding to the U and V planes) is processed using Equation 10.

2.2.4 Chroma Upsampling

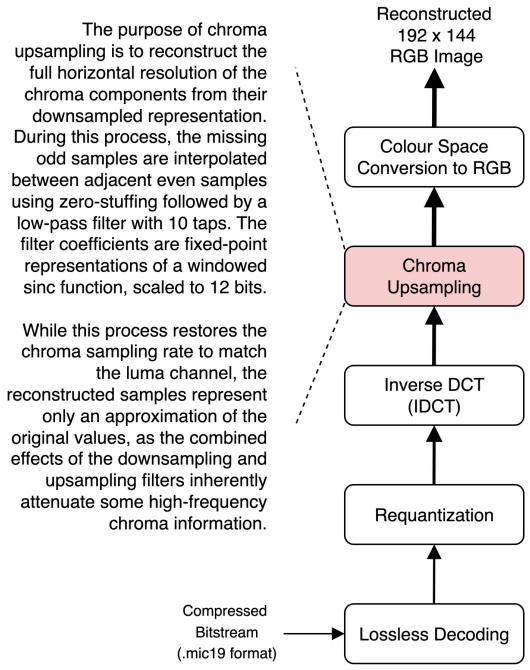


Figure 22: Chroma upsampling.

Chroma upsampling is the second last stage of the decoding process, as illustrated in Figure 22. It restores the full horizontal resolution of the chroma components U and V to match the luma component Y, reversing the 4:2:2 subsampling applied during encoding. Each reconstructed odd chroma sample is derived from its neighboring even values using interpolation.

At the conceptual level, the upsampling process begins by inserting zero-valued odd samples between the existing even chroma values in the downsampled plane, effectively doubling the horizontal sampling rate of the U and V components. This zero insertion increases the sample density but also introduces unwanted high-frequency replicas of the original spectrum. To suppress these artifacts, the sequence is processed by a finite impulse response interpolation filter with a lowpass response that preserves only the frequency range of the original chroma signal.

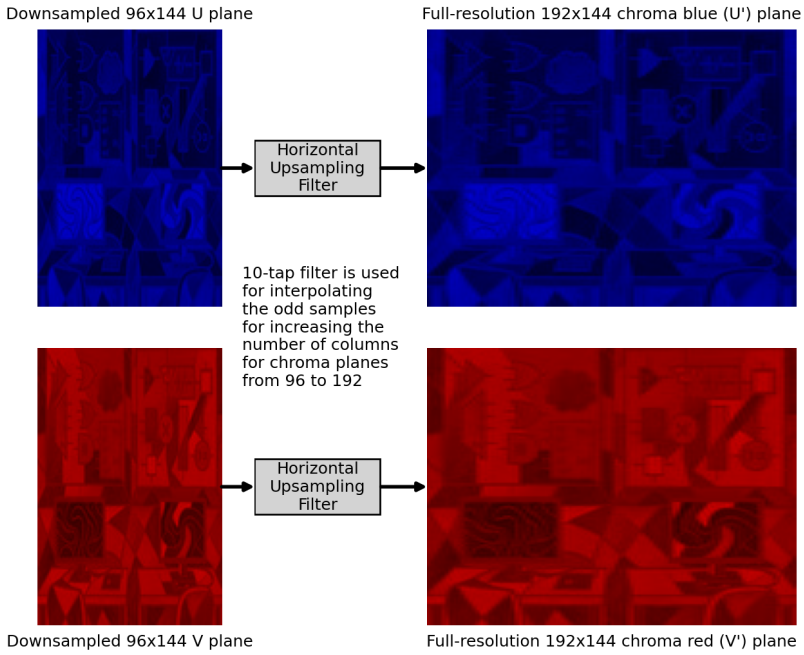


Figure 23: Upsampling from downsampled U and V planes to full-resolution U' and V' planes.

Equation 11 is evaluated for every pixel in each row to generate the horizontally interpolated chroma values. The procedure is then applied to all rows, yielding the complete upsampled chroma plane, as shown in Figure 23.

Boundary handling is performed separately for every row because the interpolation filter acts only in the horizontal direction. At the start of a row, indices $\frac{j-9}{2}$, ..., $\frac{j-1}{2}$ reference positions before the first valid sample, whereas at the end, indices $\frac{j+1}{2}$, ..., $\frac{j+9}{2}$ extend beyond the last. In these situations, the missing samples are substituted with the nearest valid value within the same row, effectively extending the boundary region. The same interpolation process is applied to the V plane to maintain consistent chroma reconstruction across both components.

The equation for the upsampling filter is given below. It follows the same design principles as the downsampling filter, using a windowed sinc formulation to realize a lowpass interpolation filter. The coefficients for 10 taps shown below are integer representations of the real values derived from the sinc function, scaled by 12 bits.

$$U'[j] = \begin{cases} U\left[\frac{j}{2}\right] & j \text{ even} \\ \frac{1}{4096}\Big(36 \cdot U\left[\frac{j-9}{2}\right] - 98 \cdot U\left[\frac{j-7}{2}\right] - 233 \cdot U\left[\frac{j-5}{2}\right] + 528 \cdot U\left[\frac{j-3}{2}\right] + 1815 \cdot U\left[\frac{j-1}{2}\right] \\ \qquad + 1815 \cdot U\left[\frac{j+1}{2}\right] + 528 \cdot U\left[\frac{j+3}{2}\right] - 233 \cdot U\left[\frac{j+5}{2}\right] - 98 \cdot U\left[\frac{j+7}{2}\right] + 36 \cdot U\left[\frac{j+9}{2}\right] + 2048\Big) & j \text{ odd} \end{cases} \tag{11}$$

Three examples are shown below. Since the even chroma samples correspond directly to the values from the downsampled plane, no computation is required for them. The interpolation filter is applied only to the odd samples to reconstruct the missing values, and the examples below illustrate this process. First, for a pixel located in the middle of a row: U at row 1, column 91.

U_{137}	U_{138}	U_{139}	U_{140}	U_{141}	U_{142}	U_{143}	U_{144}	U_{145}	U_{146}
67	69	64	87	137	139	159	143	68	70

$$U'_{283} = \frac{1}{4096}\Big(36 \cdot 67 - 98 \cdot 69 - 233 \cdot 64 + 528 \cdot 87 + 1815 \cdot 137 + 1815 \cdot 139 + 528 \cdot 159 - 233 \cdot 143 - 98 \cdot 68 + 36 \cdot 70 + 2048\Big) = 140$$

Next, the boundary condition at the beginning of a row: U at row 1, column 1.

U_{96}	U_{96}	U_{96}	U_{96}	U_{96}	U_{97}	U_{98}	U_{99}	U_{100}	U_{101}
132	132	132	132	132	150	150	158	133	93

$$U'_{193} = \frac{1}{4096}\Big(36 \cdot 132 - 98 \cdot 132 - 233 \cdot 132 + 528 \cdot 132 + 1815 \cdot 132 + 1815 \cdot 150 + 528 \cdot 150 - 233 \cdot 158 - 98 \cdot 133 + 36 \cdot 93 + 2048\Big) = 140$$

Finally, the boundary condition at the end of a row: U at row 1, column 191.

U_{187}	U_{188}	U_{189}	U_{190}	U_{191}	U_{191}	U_{191}	U_{191}	U_{191}	U_{191}
74	83	139	147	142	142	142	142	142	142

$$U'_{383} = \frac{1}{4096}\Big(36 \cdot 74 - 98 \cdot 83 - 233 \cdot 139 + 528 \cdot 147 + 1815 \cdot 142 + 1815 \cdot 142 + 528 \cdot 142 - 233 \cdot 142 - 98 \cdot 142 + 36 \cdot 142 + 2048\Big) = 144$$

Because the RGB and YUV data in the spatial domain are interpreted as 8-bit unsigned values, any resulting value that falls outside the valid range of 0 to 255 is clipped to the nearest limit. Values below 0 are set to 0, and values above 255 are set to 255. During decoding, the same rule applies to the output of the IDCT, the output of the upsampler, and the output of the colour space converter.

2.2.5 Colour Space Conversion to RGB

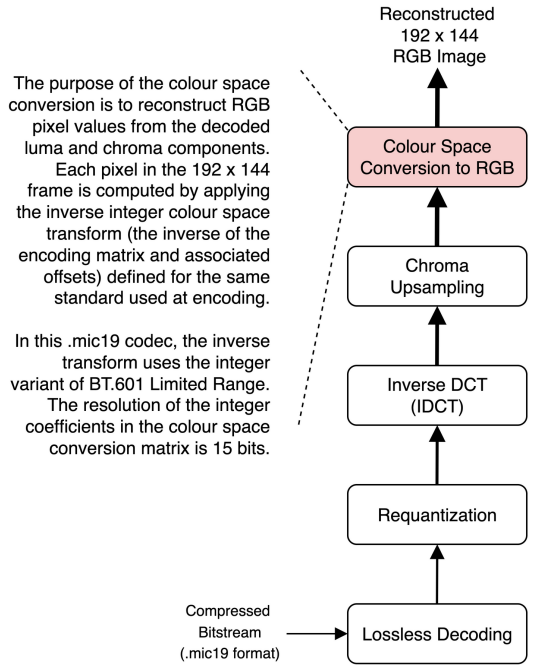


Figure 24: Colour space conversion from YUV to RGB.

The final stage of image reconstruction is the conversion from the YUV colour space back to the RGB domain, as illustrated in Figure 24. This operation reverses the colour-space transformation applied during encoding by recombining the luma and chroma components into the red, green, and blue channels of the output image. Each pixel's YU'V' triplet is converted to an RGB triplet through a linear 3×3 matrix multiplication using the integer-valued inverse of the matrix employed during encoding.

The colour space conversion is performed independently for each pixel, with all arithmetic executed in fixed-point (integer) precision according to Equation 12. After matrix multiplication, the resulting RGB values are scaled to integer form and clipped to the valid 8-bit range of 0 to 255. The reconstructed colour planes closely resemble the original ones, as illustrated in Figure 25.

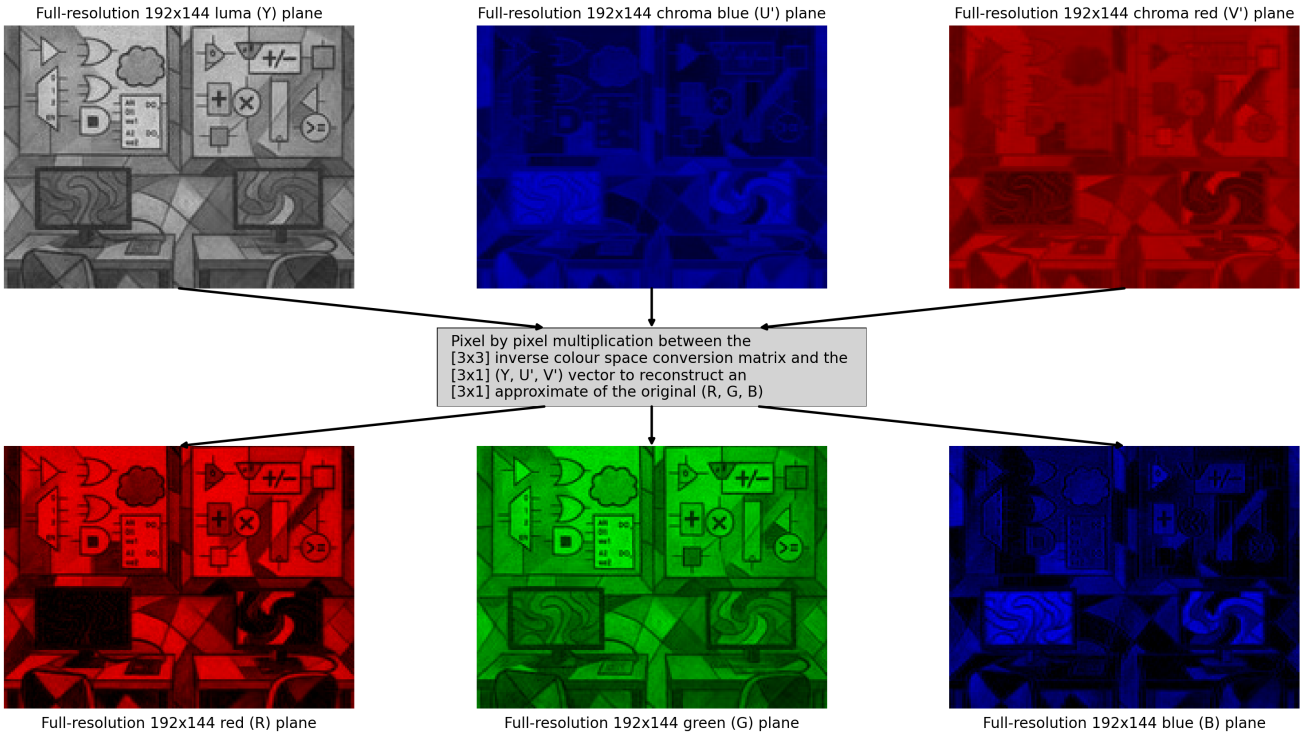


Figure 25: Conversion from the YUV colour space to the RGB colour space. Each pixel's YU'V' triplet is transformed into a RGB triplet through a 3×3 matrix multiplication.

The conversion is applied independently to each pixel. For every pixel, the three YU'V' components are linearly combined through a 3×3 matrix to reconstruct the corresponding RGB values. The transformation uses the real-valued inverse of the matrix applied during encoding, scaled by 15 bits to obtain the integer form used in the equation below.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{32768} \left(\begin{bmatrix} 38142 & 0 & 52298 \\ 38142 & -12845 & -26640 \\ 38142 & 66093 & 0 \end{bmatrix} \left(\begin{bmatrix} Y \\ U' \\ V' \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right) + \begin{bmatrix} 16384 \\ 16384 \\ 16384 \end{bmatrix} \right) \tag{12}$$

It is worth noting that the colour space conversion matrix has a characteristic structure that reflects the properties of the YUV representation. The first column, corresponding to the luma component, contains identical coefficients across all three RGB equations, ensuring that each channel receives the same luminance contribution. The remaining elements include two zeros, one in the first row and one in the last, indicating that the red channel is independent of U and the blue channel is independent of V, while the green channel is reconstructed from all three components. This structure is inherent to the definition of the luma-chroma representation.

To illustrate how the colour space conversion to RGB operates numerically, below is the reconstruction for the pixel at row 1 and column 1.

$$\begin{bmatrix} Y_{193} \\ U'_{193} \\ V'_{193} \end{bmatrix} = \begin{bmatrix} 118 \\ 140 \\ 103 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} R_{193} &= \left\lfloor \frac{38142 \cdot (118-16) + 52298 \cdot (103-128) + 16384}{32768} \right\rfloor = 79 \\ G_{193} &= \left\lfloor \frac{38142 \cdot (118-16) - 12845 \cdot (140-128) - 26640 \cdot (103-128) + 16384}{32768} \right\rfloor = 134 \\ B_{193} &= \left\lfloor \frac{38142 \cdot (118-16) + 66093 \cdot (140-128) + 16384}{32768} \right\rfloor = 143 \end{cases}$$

As with the output of the IDCT and the upsampler, **clipping** may be required. To illustrate the effect of clipping, see below the reconstruction for the pixel at row 2 and column 93.

$$\begin{bmatrix} Y_{477} \\ U'_{477} \\ V'_{477} \end{bmatrix} = \begin{bmatrix} 51 \\ 152 \\ 99 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} R_{477} &= \left\lfloor \frac{38142 \cdot (51-16) + 52298 \cdot (99-128) + 16384}{32768} \right\rfloor = 0 \\ G_{477} &= \left\lfloor \frac{38142 \cdot (51-16) - 12845 \cdot (152-128) - 26640 \cdot (99-128) + 16384}{32768} \right\rfloor = 55 \\ B_{477} &= \left\lfloor \frac{38142 \cdot (51-16) + 66093 \cdot (152-128) + 16384}{32768} \right\rfloor = 89 \end{cases}$$

In the above example, the R value evaluates to a negative number, which must be clipped to the lowest valid pixel intensity, namely R = 0.

To illustrate clipping at the upper limit, see below the reconstruction for the pixel at row 13 and column 120.

$$\begin{bmatrix} Y_{2616} \\ U'_{2616} \\ V'_{2616} \end{bmatrix} = \begin{bmatrix} 211 \\ 63 \\ 150 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} R_{2616} &= \left\lfloor \frac{38142 \cdot (211-16) + 52298 \cdot (150-128) + 16384}{32768} \right\rfloor = 255 \\ G_{2616} &= \left\lfloor \frac{38142 \cdot (211-16) - 12845 \cdot (63-128) - 26640 \cdot (150-128) + 16384}{32768} \right\rfloor = 235 \\ B_{2616} &= \left\lfloor \frac{38142 \cdot (211-16) + 66093 \cdot (63-128) + 16384}{32768} \right\rfloor = 96 \end{cases}$$

In the above example, the R value exceeds the maximum valid range and must be clipped to the highest pixel intensity, namely R = 255.

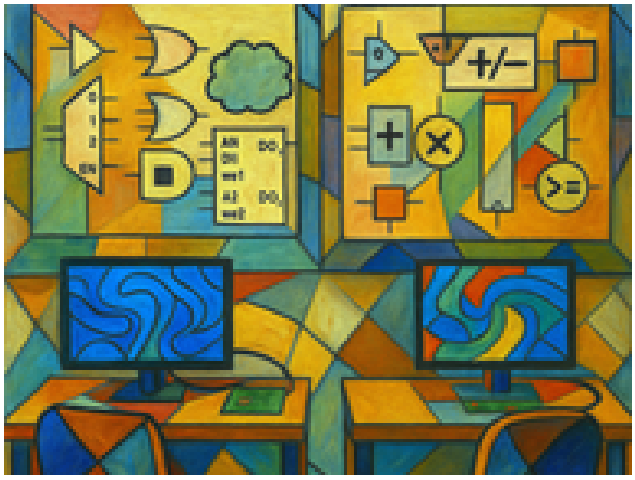


Figure 26: The original image from Figure 2.

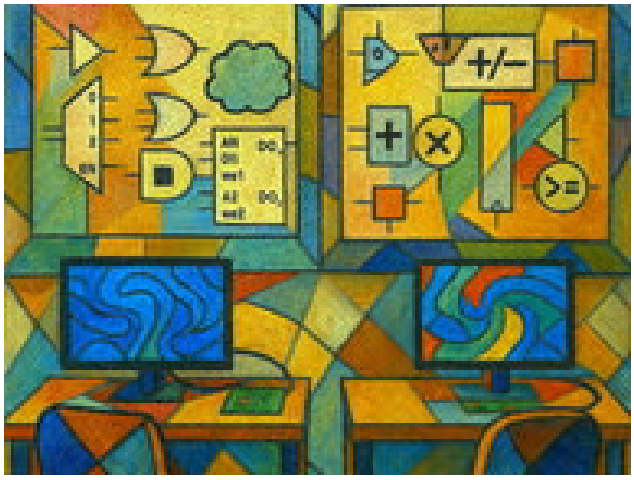


Figure 27: The reconstructed image recovered from the `mic.19` bitstream.

Figure 26 shows the original image before compression, while Figure 27 shows the image reconstructed from the `.mic19` bitstream after decoding. The two images are not bit equivalent, and the perceptible differences are primarily due to the loss of fine high-frequency details introduced by chroma downsampling and upsampling, as well as by quantization.

To quantitatively assess the fidelity of the reconstructed image, the Peak Signal-to-Noise Ratio (PSNR) metric is employed. It measures how closely the reconstructed image matches the original by comparing corresponding pixel intensities. The metric is derived from the mean squared error (MSE), which represents the average squared difference between the pixel values of the original image $I(i, j)$ and the reconstructed image $\hat{I}(i, j)$. The MSE is formally defined as:

$$\text{MSE} = \frac{1}{M \times N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i, j) - \hat{I}(i, j)]^2 \tag{13}$$

where M and N denote the image dimensions. Using this relationship, the PSNR is given by:

$$\text{PSNR} = 10 \log_{10} \left(\frac{255^2}{\text{MSE}} \right) \tag{14}$$

where 255 corresponds to the maximum possible pixel intensity for 8-bit images. Because PSNR represents a power ratio, it is expressed on a logarithmic scale and measured in decibels (dB).

A higher PSNR value indicates better reconstruction quality, although this usually comes at the expense of a lower compression ratio. For example, when quantization matrix 1 is selected during encoding, the larger quantization steps generate more zero coefficients, improving compression efficiency but reducing image fidelity. The measured results for the example image in this document (size 192 × 144, compressed in the `.mic19` format) confirm this trade-off. Increasing the quantization step size, by changing the quantization matrices shown in Figures 11 and 12 from Section 2.1.4, improves the compression ratio from approximately 3.75x to 5.45x, while reducing the PSNR from 26.31 dB to 24.92 dB.

3 Project Guidelines

This section provides a structured overview of the project across all milestones. It begins with the software model, which serves as the golden reference for validation, and continues with the design approach and the specific constraints defined for each milestone in the hardware implementation.

3.1 Software Model

The software model assists hardware developers by generating stimuli such as compressed bitstreams and intermediate data, along with golden responses including reconstructed images and SRAM memory snapshots for use in testbenches. In the context of this project, examples include the `.sram_d0`, `.sram_d1`, and `.sram_d2` files, which are used to verify the hardware implementation at the output of each project milestone. The software model, written in Python for the `.mic19` codec, is located in the `sw` sub-folder.

3.1.1 Command Line Usage

The primary entry point for the model is `main.py`, which supports three operations: **encoding**, **decoding**, and **comparison**. All input and output files should be stored in the `../data/` subdirectory to maintain consistency. To encode an image, run the following Python script in the `sw` folder:

```
python3 main.py -encode ../data/image.ppm
```

The above encodes the input image `../data/image.ppm` and it generates the compressed file in the `.mic19` format: `../data/image_0.mic19`. The suffix `_0` indicates the use of the default quantization matrix 0. If the input image dimensions differ from 192x144, it is automatically rescaled to produce `../data/image_192x144.ppm` as the encoding source. To decode a previously compressed `.mic19` file, run:

```
python3 main.py -decode ../data/image_0.mic19
```

This generates `../data/image_0_sw.ppm`, matching the spatial dimensions of `image_192x144.ppm`. Due to the lossy compression, pixel values may differ from the original image. To compare the original and reconstructed images, use:

```
python3 main.py -compare ../data/image_192x144.ppm ../data/image_0_sw.ppm
```

The above calculates the PSNR in dB between the two images, as defined in Equation 14.

3.1.2 Verification Data

During decoding, as illustrated in Figure 28, intermediate files are generated to facilitate verification and debugging both during simulation and on the hardware platform. These files capture snapshots that must be consistent with the contents of the external SRAM between different stages (milestones) of the hardware implementation.

- `image_0.sram_d0` — data at the output of milestone 1; this is the golden reference data for the R, G, and B values written to the external SRAM.
- `image_0.sram_d1` — data at the input of milestone 1; this data serves as stimuli for colour space conversion and upsampling, and at the same time, is golden reference data for the IDCT.
- `image_0.sram_d2` — data at the input of milestone 2; this data serves as stimuli for the IDCT and, at the same time, as golden reference data for validating the correctness of the hardware implementation of the lossless decoder and requantizer.

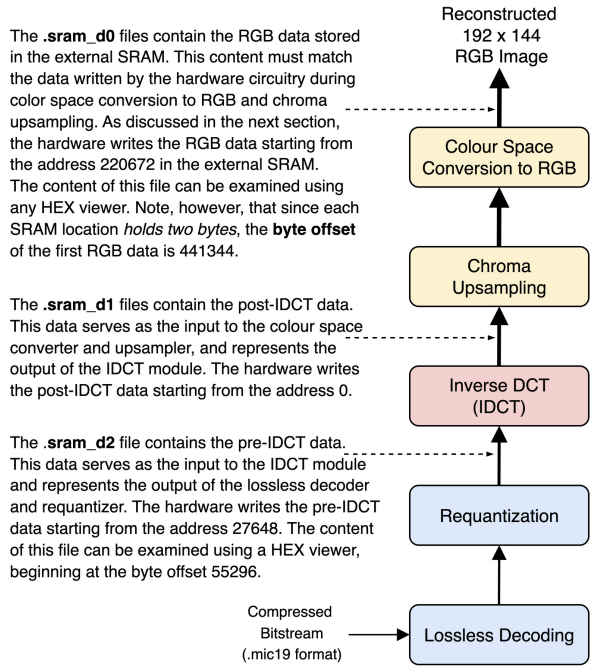


Figure 28: Snapshots from the software model used to provide both stimuli and golden reference data for validating the hardware implementation.

Each `.sram_d` file is 2^{18} bytes in size, corresponding to the 18-bit address space of the SRAM in the hardware implementation. The external SRAM layout at the input and output of each milestone will be detailed in the following sections. For now, it is important to note that these memory dumps are binary files needed to validate the intermediate data produced at the output of each milestone.
From a hardware design perspective, the goal is to create digital circuitry that precisely replicates the software model's behaviour bit for bit, using the software model as the golden reference while meeting the specified resource and utilization targets for each milestone. Memory dumps from the `.sram_d` files play a key role during verification, where checks are restricted to the memory regions written by each milestone. This separation of concerns allows the software modelling phase, already completed since the software model is provided, to focus on the application, while the hardware design phase, which is the primary focus moving forward, is concerned with a correct implementation within the defined constraints.

3.1.3 Module Overview

Module	Description
`encoder.py`	Performs encoding: RGB-to-YUV conversion, downsampling, block transform, quantization, and lossless coding.
`decoder.py`	Performs decoding: lossless decoding, requantization, inverse transform, upsampling, and YUV-to-RGB conversion.
`converter.py`	Implements color space conversion, both RGB-to-YUV and YUV-to-RGB.
`xsampler.py`	Implements downsampling and upsampling filters for horizontal chroma subsampling.
`transform.py`	Defines DCT/IDCT matrices and block transforms: a 16×16 transform for the luma plane and a 8×8 transform for the chroma planes, both fixed-point DCT-II variants.
`quantizer.py`	Provides quantization matrices and methods for quantizing and requantizing luma and chroma data blocks.
`lossless.py`	Encodes and decodes quantized coefficient blocks using a custom mapping scheme tailored to the `.mic19` format.
`support.py`	Offers utility functions for block extraction, PSNR calculation, and binary memory dump generation.
`reader.py`	Manages reading and writing of `.ppm` and `.mic19` file formats, including rescaling PPM images that do not match the expected dimensions of 192x144.
`parser.py`	Handles parsing of command-line arguments.

3.2 Milestone 1

The project guidelines are not step-by-step instructions but a framework designed to help students develop a way of thinking about digital hardware design in the context of a real-world application. The specific challenges and trade-offs introduced by the constraints of this year's version of the project will be examined in detail during classes as the project progresses.

The project consists of three milestones, each presented and discussed in its own section. The first milestone includes an acclimatization period during the transition week when the final lab is completed. During this transition week, students are expected to develop a foundational understanding of the application described in Section 2. The aspects of the application relevant to the hardware implementation will be explored in greater depth during lectures as the course progresses through the project phase. This initial stage focuses on building familiarity with fundamental hardware concepts such as digital filters, fixed-point arithmetic, and Multiply-and-Accumulate (MAC) units. Taking into account the learning curve rather than the design complexity for Milestone 1, and assuming steady progress in design, implementation, and verification, the normal expectation is to complete Milestone 1 within a two-week timeframe following the transition week. The additional time allocated to Milestone 1 is also intended to help students become familiar with the verification flow and gain experience in troubleshooting issues that inevitably arise in complex hardware implementations.

As a final introductory note applicable to all three milestones, the memory layouts provided in the Project Guidelines section are consistent with the SRAM debug files generated by the software reference model summarized in Section 3.1, with all source code available in the `sw` sub-folder. Neither the memory layouts nor the arithmetic defined in the software model should be changed, since together they establish the functional constraints that the hardware implementation must satisfy. In addition to these functional constraints, each milestone also includes non-functional requirements such as limits on multiplier resources and utilization, all of which must be met strictly.

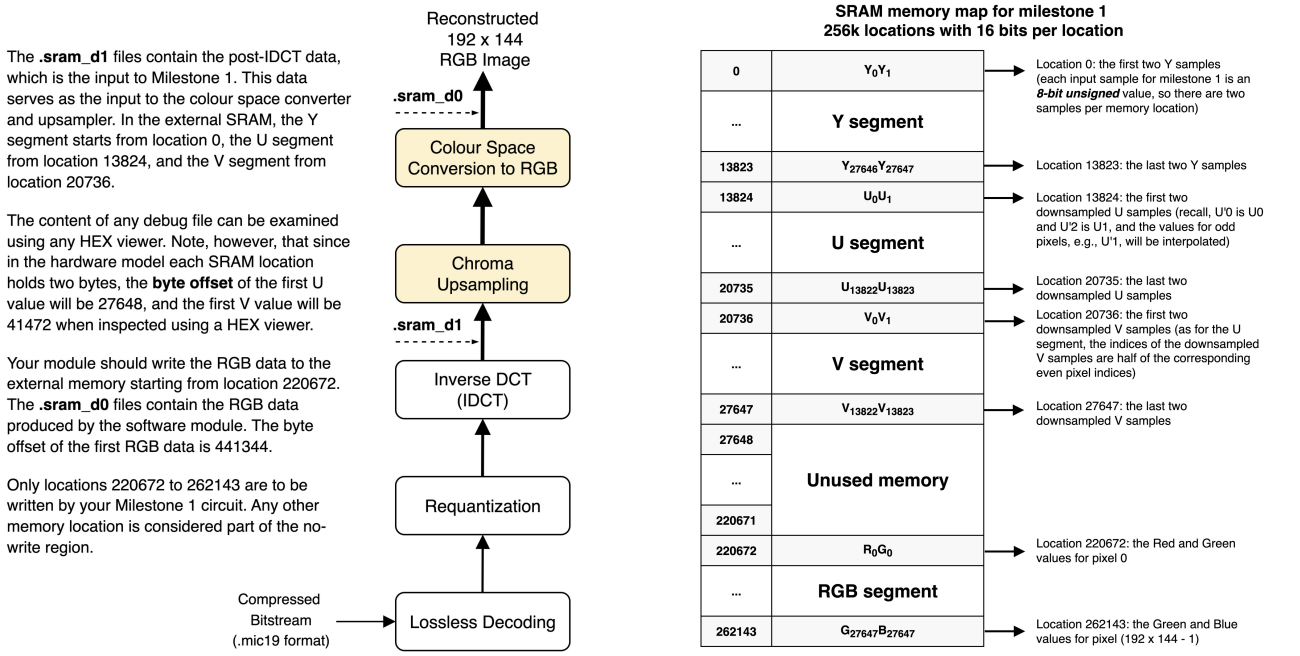
3.2.1 Overview

The hardware decoder design is divided into three milestones. Implementation proceeds backward, from the stage that reconstructs the 192 x 144 image to the one that processes the compressed `.mic19` bitstream. This approach enables visual progress checks at each milestone and provides a framework for the incremental design, integration, and verification of each new hardware module.

Figure 29 shows where the Milestone 1 hardware fits within the decoding process. As introduced in Section 3.1.2, the `.sram_d` files act as both stimuli and golden responses for verification. Until the full hardware system is complete, part of the decoding runs in software and part in hardware, with the `.sram_d1` file linking the two for Milestone 1. Figure 29a indicates where the memory snapshots for Milestone 1 are captured in software: `.sram_d1` is the output of the IDCT and the input to upsampling and colour space conversion, while `.sram_d0` is the RGB output of Milestone 1.

The data layout in `.sram_d1` is shown in Figure 29b. The Milestone 1 input data are YUV samples from the IDCT, arranged in separate segments for Y, U, and V (see Figure 29b for segment boundaries). Within each segment, data is stored in raster-scan order: across the first row, then proceeding to the next row, and so on. Each sample is one byte interpreted as an 8-bit *unsigned* number, so two samples are stored per external SRAM location. The Milestone 1 output consists of R, G, and B samples in the same format as in experiment 4 from lab 5, as summarized in Figure 29b. Note that the RGB segment is generated by the Milestone 1 logic and is therefore not part of `.sram_d1`; its content, stored in `.sram_d0` by the software model, is used to validate the hardware output.

Finally, memory dumps from the `.sram_d` files can be inspected with any `HEX` viewer. Because each SRAM location stores two bytes, the byte offset equals the memory location multiplied by two. This relationship is exemplified in Figure 29a.



(a) Image decoding flow. (b) Memory map.

Figure 29: Milestone 1 in hardware - positioning within the image decoding flow and its memory map.

3.2.2 Upsampling

The decoding stages implemented in M1 are upsampling (Section 2.2.4) and colour space conversion (Section 2.2.5). Both of these stages require the use of fixed-point multipliers. In practice, there are usually fewer available multipliers than the number of multiplications required in the application, often necessitating a time-sharing mechanism in which multiple operations share the same multiplier across different clock cycles. The instantiation shown in Figure 30 should be used, with the signals `Mult_op_1` and `Mult_op_2` multiplexed by the logic of your design.

It is important to note that multipliers must be instantiated explicitly to control their usage. Consider the code snippet shown in Figure 31a. Although the multiplications occur at mutually exclusive times, suggesting that one hardware multiplier could be shared, with `b` and `d` multiplexed at one input and `c` and `e` multiplexed at the other, this is not explicitly specified in the code. As a result, the synthesis tools may instantiate two hardware multipliers to compute both `b * c` and `d * e`, and then multiplex the two results to produce `a`. To prevent this, explicitly specify that only one multiplier is used by multiplexing the operands before the multiplication, as shown in Figure 31b. This approach ensures that a single hardware multiplier is instantiated and time-shared between operations.

Note that regardless of the sample size, all multiplications are performed on 32 bits, requiring zero or sign extension of the operands depending on how the samples are interpreted in different contexts.

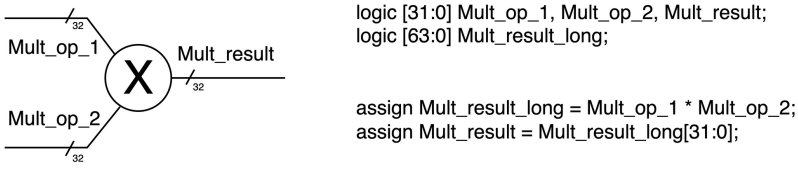


Figure 30: Hardware multiplier instantiation.

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

(a) Implicit multiplier instantiation.

```
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;
```

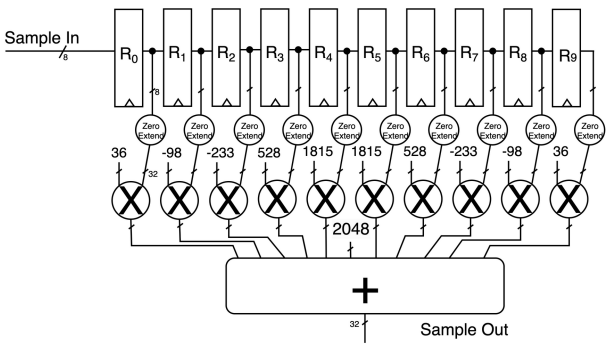
(b) Explicit multiplier instantiation.

Figure 31: Comparison of multiplier instantiation approaches.

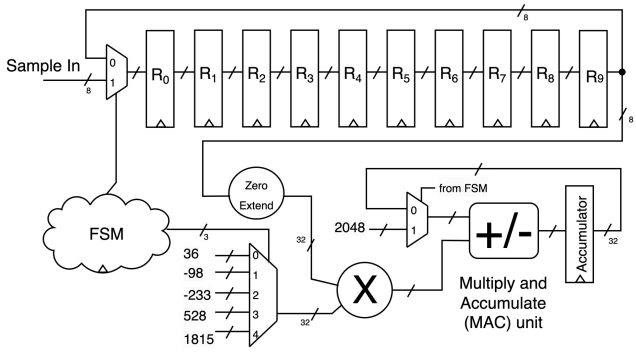
Having established how multipliers can be instantiated, we now focus on implementing upsampling with fixed-point multipliers. Equation 2, introduced in Section 2.2.4, is reproduced below for convenience.

$$V'[j] = \begin{cases} V\left[\frac{j}{2}\right] & j \text{ even} \\ \frac{1}{4096}\Big(36 \cdot V\left[\frac{j-9}{2}\right] - 98 \cdot V\left[\frac{j-7}{2}\right] - 233 \cdot V\left[\frac{j-5}{2}\right] + 528 \cdot V\left[\frac{j-3}{2}\right] + 1815 \cdot V\left[\frac{j-1}{2}\right] \\ \qquad + 1815 \cdot V\left[\frac{j+1}{2}\right] + 528 \cdot V\left[\frac{j+3}{2}\right] - 233 \cdot V\left[\frac{j+5}{2}\right] - 98 \cdot V\left[\frac{j+7}{2}\right] + 36 \cdot V\left[\frac{j+9}{2}\right] + 2048\Big) & j \text{ odd} \end{cases}$$

Figure 32a shows a straightforward implementation of the interpolating filter, where samples are stored in a 10-tap shift-register structure, each multiplied by a constant in the same clock cycle, and the partial results are combined through an adder tree. Although this structure achieves high throughput, since one filtered sample is produced every clock cycle, it requires significant hardware resources when the number of filter taps is high. Moreover, unless new input samples are available on every clock cycle, the hardware utilization will be low. Figure 32b illustrates how a single multiplier can be shared to perform the same computation as in Figure 32a, at the cost of reduced throughput. In this design, a MAC unit accumulates the partial product of two inputs in a register. An Finite State Machine (FSM) controls the selection of the constant coefficient and initialization. With this architecture, only one multiplier is required, but each filtered sample requires at least 10 clock cycles to produce. This represents a typical architectural trade-off between hardware resources, their utilization, and the overall throughput.



(a) 10 multipliers (one per tap).



(b) Single time-shared multiplier with MAC unit.

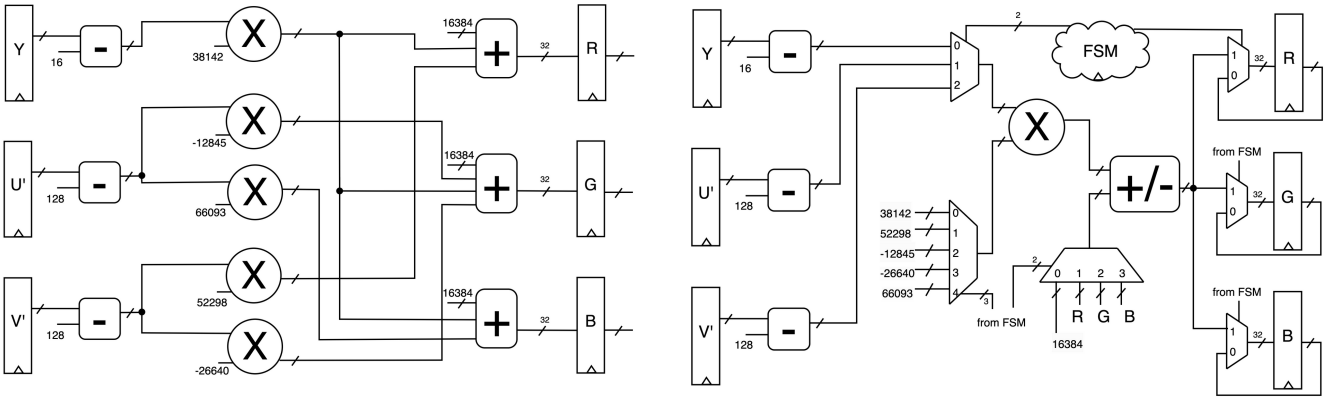
Figure 32: Alternative standalone interpolation filter architectures: high throughput with more resources, or few resources with reduced throughput. Scaling and clipping logic omitted.

3.2.3 Colour Space Conversion

Once chroma upsampling reconstructs the missing odd samples, these together with the even samples available in the downsampled plane are passed, along with the corresponding luma samples, to the colour space converter to generate the R, G, and B samples. The colour space conversion to RGB, detailed in Section 2.2.5, follows Equation 12, which is reproduced below for convenience.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{32768} \left(\begin{bmatrix} 38142 & 0 & 52298 \\ 38142 & -12845 & -26640 \\ 38142 & 66093 & 0 \end{bmatrix} \left(\begin{bmatrix} Y \\ U' \\ V' \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right) + \begin{bmatrix} 16384 \\ 16384 \\ 16384 \end{bmatrix} \right)$$

Figure 33a shows a straightforward implementation of the colour space conversion, which achieves high throughput, producing one triplet of RGB values per clock cycle. However, because the allocated multipliers must be shared between interpolation and colour space conversion, this approach is not suitable for our implementation. In the same way that the MAC concept is used to share multiplications in the interpolation filters, it can also be applied to colour space conversion. The corresponding data path is shown in Figure 33b.



(a) One RGB triplet per clock cycle.

(b) Single time-shared multiplier.

Figure 33: Alternative standalone colour space conversion architectures: high throughput with greater resource use, or low throughput with fewer resources. Scaling and clipping logic is omitted.

3.2.4 Resource and Utilization Constraints

Neither of the design choices used to illustrate the trade-offs for the upsampler (Figure 32) and the colour space converter (Figure 33) is a direct fit for this project. For the hardware implementation of the `.mic19` decoder in Milestone 1, the design is limited to 4 multipliers, each with input operands of 32 bits. These 4 multipliers are **shared** between the colour space converter and the upsampler. Accordingly, there will be 4 instances of the form shown in Figure 30.

The average utilization of the 4 multipliers for Milestone 1 must be at least 80%. The utilization of a multiplier is defined as the ratio between the number of clock cycles during which the multiplier performs calculations and the total number of clock cycles for the respective milestone. For illustration, consider a hypothetical milestone that lasts 100 clock cycles and uses two multipliers. The first multiplier performs calculations for 70 clock cycles (utilization of 70%), while the second performs calculations for 90 clock cycles (utilization of 90%). The average utilization across both multipliers in this case is 80%.

Understanding the trade-offs between the alternative design approaches illustrated in Figures 32 and 33, together with the reasoning process used to evaluate design decisions that meet the specific resource and utilization constraints of the hardware implementation of the `.mic19` decoder, will be **discussed in detail during classes held throughout the Milestone 1 phase of the project**.

3.2.5 Design and Verification Strategy

Milestone 1 should begin with the use of a state table, as it is well suited for planning and structuring this type of design before implementation. Although state tables are not recommended for Milestone 2 due to the increased complexity of block transfers and matrix calculations, and are not feasible for Milestone 3, they remain valuable at this stage for organizing the design, particularly when the number of clock cycles per pair of pixels is fixed. In general, state tables are effective in systems with static schedules and data-independent latency. As more experience is gained, reliance on state tables will gradually diminish.

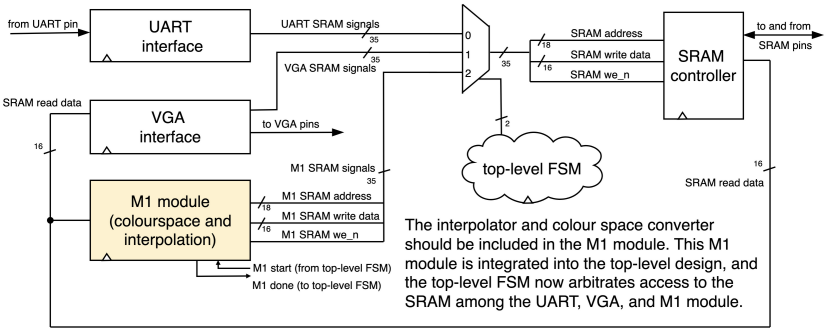
State code (clock cycle)	**Lead-In (LI)**			**Common Case (CC)** (capture three iterations)			**Lead-Out (LO)**		
	$\mathbf{LI}_0$	**...**	$\mathbf{LI}_{i-1}$	$\mathbf{CC}_0$	**...**	$\mathbf{CC}_{j-1}$	$\mathbf{LO}_0$	**...**	$\mathbf{LO}_{k-1}$
SRAM_address									
SRAM_read_data									
SRAM_write_data									
SRAM_we_n									
Y_even									
Y_odd									
U_regs[...]									
V_regs[...]									
...									
U_odd									
V_odd									
R_even									
R_odd									
...									

Table 6: Skeleton state table for Milestone 1 implementation.

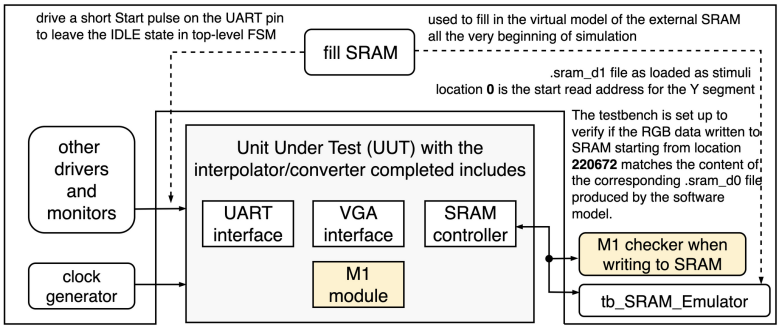
Table 6 provides a useful starting point for a state table that can be applied to Milestone 1. The main groups of signals include memory interface signals, value-holding registers, and accumulators. Add as many groups or signals as needed to organize your design. The state transitions can be divided into three phases: Lead-In, Common Case, and Lead-Out. Start with the Common Case, and it is recommended to process a pair of pixels (or a multiple of a pair) in one iteration of this phase. Focus on at least three iterations of the Common Case to identify dependencies between memory reads and writes, as well as the interactions among value-holding registers and accumulators.

Once the state table provides confidence that the design constraints for this milestone are satisfied, you can proceed to the implementation phase. The Milestone 1 circuit should be integrated into the top-level design as a standalone unit, as shown in Figure 34a. This modular approach enables flexible verification and future scalability. During verification, `tb_project_v0` can be configured to verify the logic of each milestone independently from the rest of the system, as summarized in Figure 34b for Milestone 1. This not only simplifies debugging and validation but also supports cleaner code organization and smoother integration as additional milestones are implemented. By default, the checker and the no-write regions in `tb_project_v0` are set up to verify Milestone 1, using `.sram_d1` as the input and `.sram_d0` as the golden reference. Comments in the source code explain how the testbench can be extended to verify other milestones independently from the rest of the design.

Once the design has been integrated into the top-level system and `tb_project_v0` passes, `tb_project_v1` should be used to verify that control from Milestone 1 is properly handed off to the VGA interface. This is necessary because, by default (as implemented in the starter code reused from lab 5), the UART interface writes data to address 0, while the VGA interface also begins reading from address 0. The VGA interface must be modified to read the RGB data from the correct memory segment.



(a) Integration flow.



(b) Verification approach.

Figure 34: Milestone 1 integration and verification.

The starter code for the project is lab 5, experiment 4. Figure 34 summarizes both the integration into this starter code and the verification approach to be used for Milestone 1.

After all testbenches pass, the `.sram_d1` file should be transferred from the host computer to the external SRAM on the board through the UART interface. During simulation, the UART interface is bypassed to shorten the simulation time, allowing the testbench to begin exercising the design as early as possible. This bypass is not possible when running on the actual board. Therefore take note that, the UART interface from lab 5, experiment 4 *strips the header* from the `.ppm` file, which **is no longer needed** (see the transition from the idle state in `UART_SRAM_interface.sv` in the `rtl` sub-folder).

As a final but important point, once the implemented design successfully displays the decoded image on the VGA monitor, it should also be capable of receiving and processing a new image.

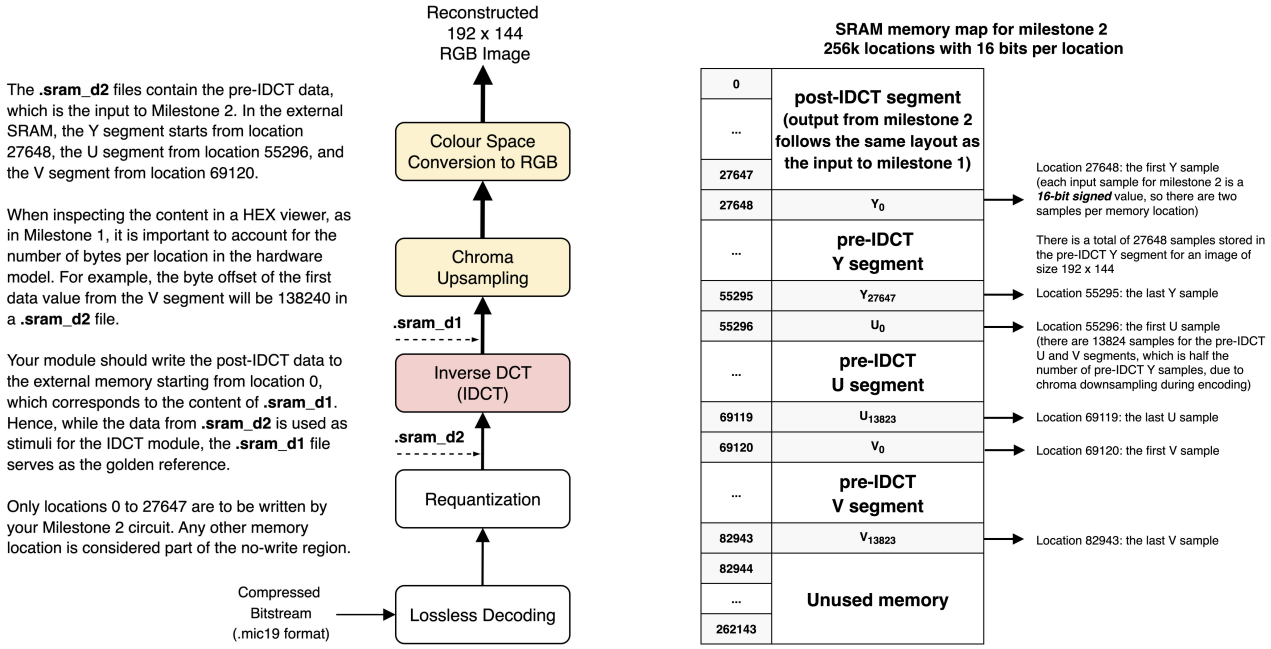
3.3 Milestone 2

Milestone 2 builds upon the foundations established in the first milestone and introduces two essential concepts that extend beyond the scope of this project: the transfer of data between embedded memories and external memory, and the implementation of arithmetic circuitry for processing multi-dimensional arrays stored in embedded memories. In this case, the focus is on two-dimensional arrays, as required by the IDCT module.

It is important to stress that **Milestone 2 should not be attempted before Milestone 1 has been fully completed.** The knowledge and skills developed earlier, such as the understanding of fixed-point arithmetic and MAC units, are critical prerequisites for meaningful progress at this stage of the project. Once those fundamentals are firmly established, students can focus on new challenges, specifically block transfers between external and embedded memories and the design of arithmetic circuitry for matrix-based calculations.

Although the design complexity increases in Milestone 2, steady advancement along the learning curve makes it feasible to complete this stage within approximately one week. This expectation **assumes** that the concepts from Milestone 1 have been thoroughly understood and that students have gained substantial experience in both design and verification during the preceding stage of the project.

3.3.1 Overview



(a) Image decoding flow. (b) Memory map.

Figure 35: Milestone 2 in hardware - positioning within the image decoding flow and its memory map.

Figure 35 shows where Milestone 2 fits within the decoding flow (Figure 35a) and the corresponding memory map to be used (Figure 35b). The input (stimuli) to Milestone 2 (pre-IDCT data) is obtained from the software model and stored in the `.sram_d2` file. The output (golden response) from Milestone 2 (post-IDCT data) serves as the input to Milestone 1 and is stored in the `.sram_d1` file by the software model. The input samples are 16-bit values interpreted as *signed* integers. Hence, there is only one sample per external memory location in the pre-IDCT segments. The output samples, which form the input to Milestone 1, are 8-bit values interpreted as *unsigned* integers. As with the input to Milestone 1, the output of Milestone 2 stores two samples per external memory location.

3.3.2 Matrix Multiplication

The main purpose of Milestone 2 is to perform block transforms, i.e., the inverse discrete cosine transform (IDCT). At its core, this involves matrix multiplications. The IDCT operates on different block sizes depending on the plane type, as shown in the equations below, reproduced from Section 2.2.3.

Luma: input block (S') and output block (S) are of size 16 x 16

$$S = \frac{1}{8192}\left(\mathbf{C}_{16}^{T} \times \frac{1}{32}\left(S' \times \mathbf{C}_{16}\right) + 4096\right)$$

Chroma: S' and S are of size 8 x 8

$$S = \frac{1}{8192}\left(\mathbf{C}_{8}^{T} \times \frac{1}{32}\left(S' \times \mathbf{C}_{8}\right) + 4096\right)$$

The coefficient matrices used in the above equations are listed in Section 2.2.3 and will be abstracted away for the remainder of this section, as the same principles apply regardless of matrix size. We will ignore scaling and clipping operations, which must be implemented but are not relevant to the main challenges summarized below.

The IDCT can be expressed as a sequence of two matrix multiplications: $T = S \times \mathbf{C}$ and $S' = \mathbf{C}^{T} \times T$, where T is the intermediate matrix having the same dimensions as S' and S.

Based on the above, for each block of pre IDCT data, regardless of whether it belongs to the luma or chroma component or its specific dimensions, the following four tasks are performed:

- **Fetch S':** Move a block of pre IDCT data (S') from the external SRAM to the embedded RAM.
- **Compute T:** Compute the intermediate matrix (T) by post multiplying S' with $\mathbf{C}$.
- **Compute S:** Compute the output block of post IDCT data (S) by pre multiplying T with $\mathbf{C^T}$.
- **Write S:** Move the block of post IDCT data (S) from the embedded RAM to the external memory.

These four tasks (**Fetch** S', **Compute** T, **Compute** S, and **Write** S) are repeated for all luma and chroma blocks. As analyzed in Section 2.1.3, for an image with 144 rows and 192 columns processed by the `.mic19` codec, there are 108 luma blocks of size 16 x 16 and 216 chroma blocks of size 8 x 8 for each of the two chroma channels (U and V).

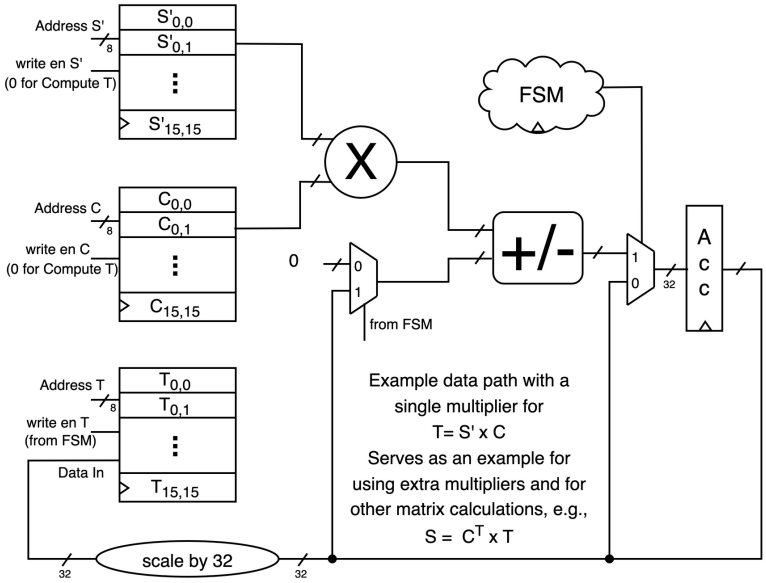


Figure 36: Example datapath for matrix-matrix multiplication using three embedded memories and a single multiplier for the intermediate matrix (T) in the luma plane.

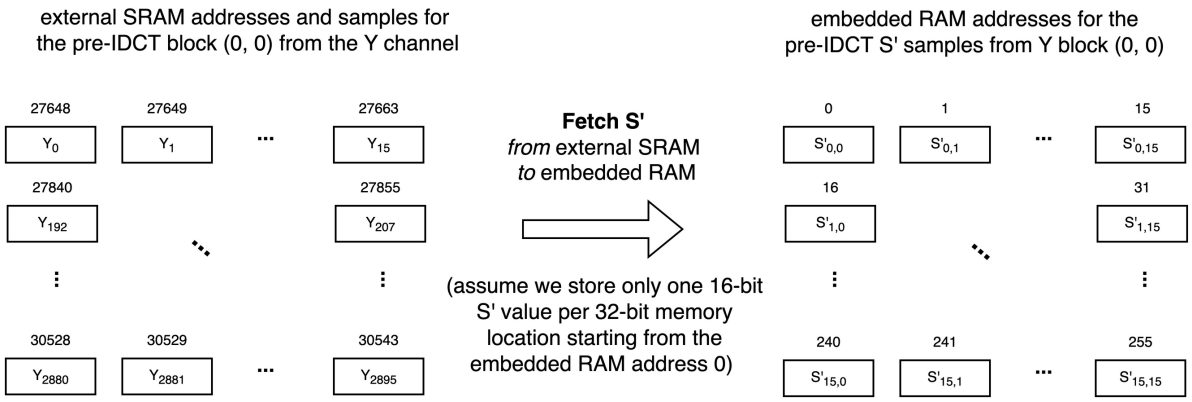
Using the same principle of MAC units introduced in Milestone 1, Figure 36 illustrates how to implement matrix matrix multiplication using three embedded memories and a single multiplier. For simplicity, we assume three independent embedded memory blocks for storing the matrices. For the specific constraints regarding the number of embedded memories to be used throughout Milestone 2, refer to Section 3.3.4.

Matrix multiplication **must** be performed with data stored in embedded memories. No direct bypasses from external SRAM are permitted.

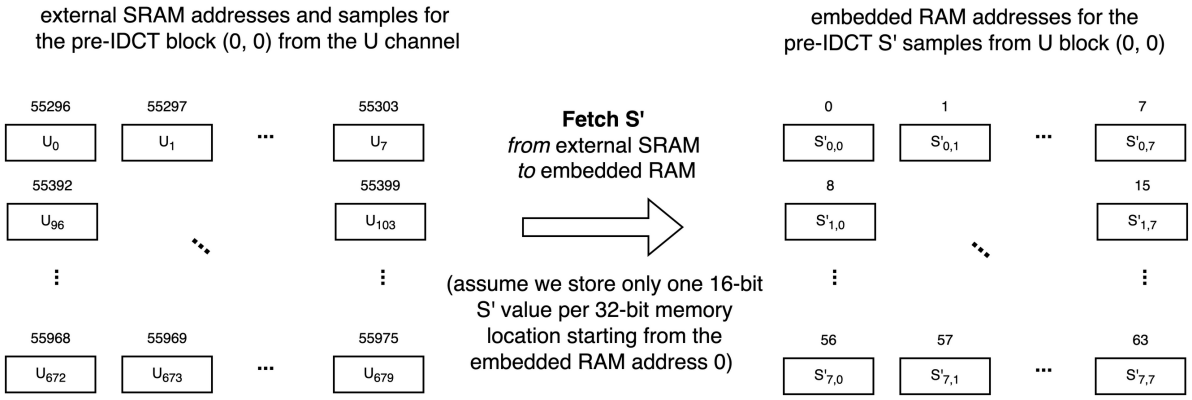
3.3.3 Block Addressing

As discussed in the previous section, in Milestone 2 we need to fetch an entire 16 × 16 luma block of pre-IDCT data before performing matrix calculations for that block. The same applies to chroma blocks of size 8 × 8. This corresponds to the **Fetch** S' task.

The blocks are stored in separate segments for Y, U, and V, as shown in Figure 35b. Although there is only one 16-bit S' sample per external memory location, as in Milestone 1, they are stored in raster-scan order (all samples from the first row, followed by all samples from the second row, and so on). Figure 37 shows how the data is accessed for one block in the external memory to bring it to the embedded RAM. Although embedded RAMs have 32 bits per location, for simplicity, we assume in this figure that only one sample of S' is stored in the embedded RAM (right-hand side). While the layout of the external memory is fixed, as part of your design choices you can decide whether to store one or two S' samples per embedded RAM location.



(a) Fetching luma (Y) block from external memory.



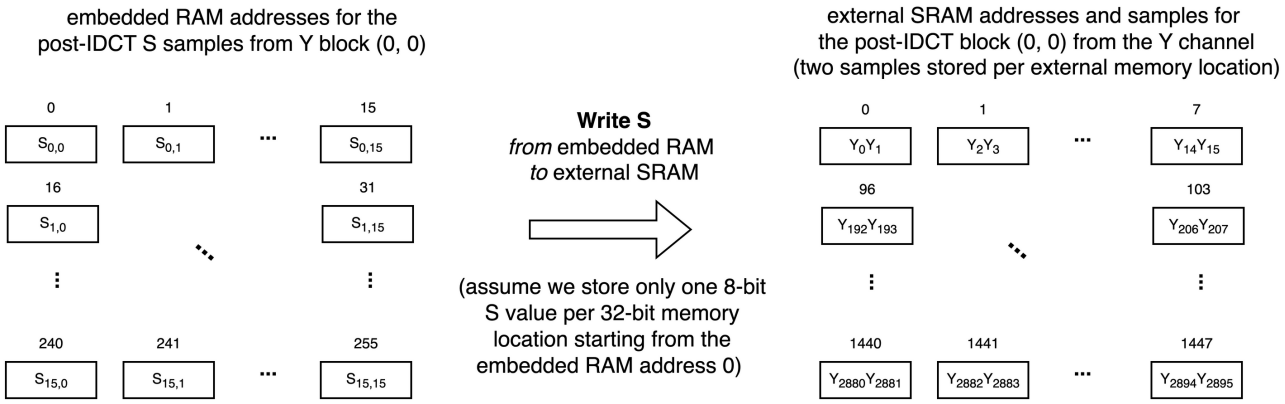
(b) Fetching chroma (U) block from external memory.

Figure 37: Data block fetching from external memory to embedded memories for matrix multiplication.

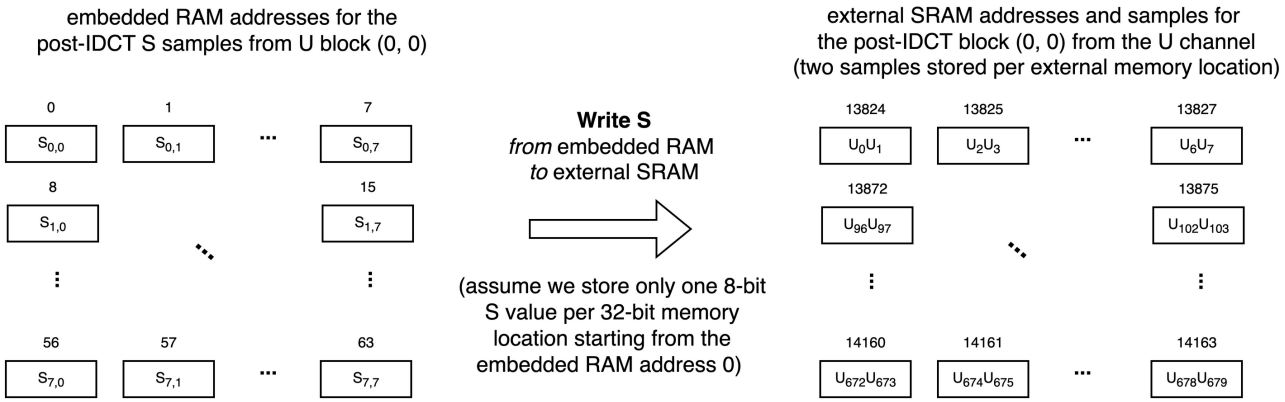
Pay particular attention to the differences in addressing between the luma blocks (Figure 37a) and the chroma blocks (Figure 37b). One difference lies in the block shape, 16×16 for luma versus 8×8 for chroma; the other difference lies in the image line width, 192 for luma and 96 for chroma. How the address generators for varying block shapes and block widths can be designed in a systematic way will be discussed in class during the Milestone 2 phase of the project.

After all calculations for a block S' have been completed, that is, both **Compute** T and **Compute** S, the post-IDCT data S for that block must be written back to the external memory. This is the **Write** S task. Since the output of Milestone 2 will serve as the input to Milestone 1, the post-IDCT data, after clipping, will contain 8-bit samples interpreted as *unsigned* values. Because the external memory is 16 bits wide, two samples (an even and an odd sample from the same row) are stored together in one external SRAM location.

Figure 38 shows how the data is accessed to write a block from the embedded RAM to the external SRAM. In this example, for simplicity, we assume that only one sample of S is stored in the embedded RAM (left-hand side). As part of your design choices, you may decide whether to store one, two, or four S samples per embedded RAM location. Similar to the **Fetch** S' task, there are conceptual differences in addressing patterns between the luma (Figure 38a) and chroma blocks (Figure 38b), which stem from their different block shapes and line widths. However, compared to **Fetch** S', the **Write** S task has an additional constraint on external memory addressing caused by storing two 8-bit samples in one external memory location.



(a) Writing luma (Y) block to external memory.



(b) Writing chroma (U) block to external memory.

Figure 38: Data block writing from embedded memories to external memory after matrix multiplication.

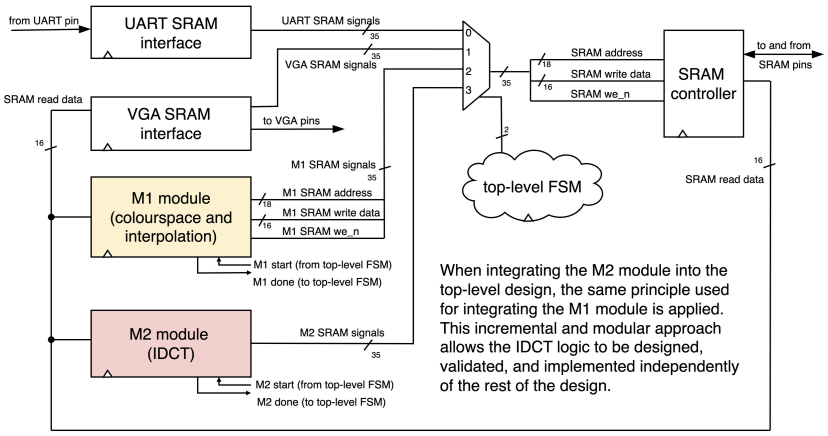
3.3.4 Resource and Utilization Constraints

While the project guidelines for Milestone 2 emphasize the use of MAC units for matrix multiplication, the specific hardware constraints for the `.mic19` codec are defined in this section. You must use exactly 3 multipliers for all processing in Milestone 2, including both **Compute** T and **Compute** S. The average utilization of these multipliers over the entire milestone should be 85.0%.

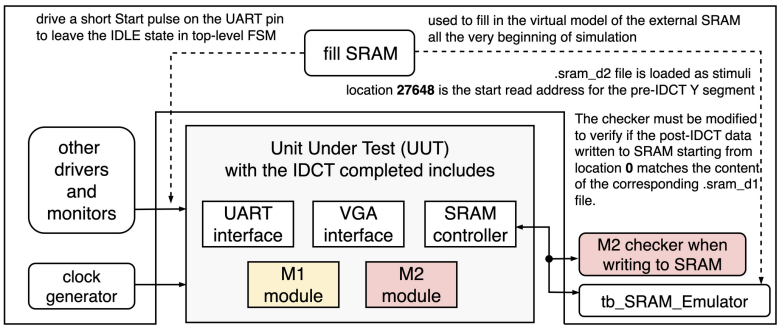
In addition to the constraints on the multipliers and their utilization, Milestone 2 imposes limits on the number of embedded memories you may use. For this project, you may use 4 embedded memories, each with an 8-kbit capacity, organized as 2^9 locations $\times$ 32 bits per location. Note that T samples must be stored as 32-bit values in the embedded RAMs because the intermediate matrix samples may exceed the range supported by 16 bits. However, S' samples are 16-bit, S samples are 8-bit (after clipping), and C samples can be represented with up to 16 bits. Therefore, there are several possible layouts for organizing data in the embedded RAMs. Both S' and C values can be stored as up to two samples per embedded RAM location, whereas S can have up to four clipped samples per embedded memory location.

A key challenge is that if the four tasks (**Fetch** S', **Compute** T, **Compute** S, **Write** S) are processed serially, the multipliers remain idle during **Fetch** S' and **Write** S, which lowers multiplier utilization. Therefore, a concurrent approach across consecutive blocks is needed. For example, while **Compute** S is performed for the current block, **Fetch** S' can be initiated for the next block; similarly, while **Compute** T is performed for the current block, **Write** S can be completed for the previous block. This concurrent approach will be **discussed in class throughout the Milestone 2 phase of the project**.

3.3.5 Design and Verification Strategy



(a) Integration flow.



(b) Verification approach.

Figure 39: Milestone 2 integration and verification.

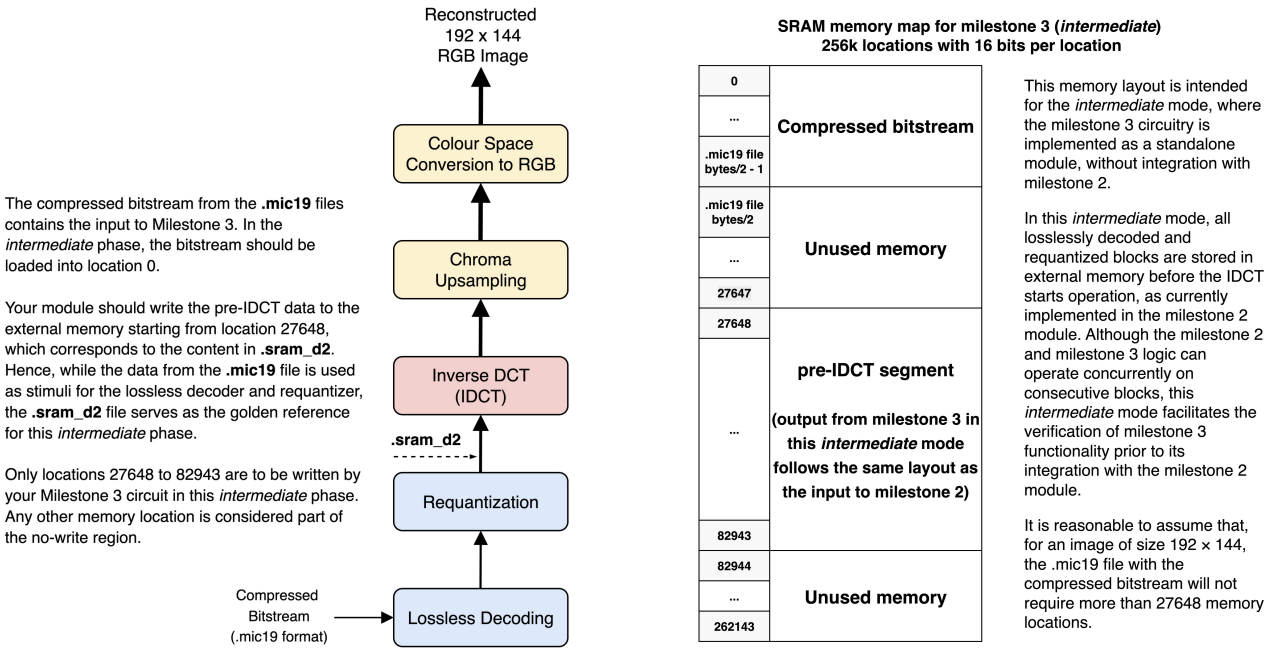
Due to the modular and incremental nature of the project, the conceptual integration and verification process for Milestone 2 is similar to that of Milestone 1, as illustrated in Figure 39.
From a design standpoint, a key difference is that the duration and size of the block transfers and calculations make a state table approach impractical. While it is still useful to visualize, in a state table-like format, what happens during transitions between consecutive matrix calculations (between **Compute** T and **Compute** S), the block addressing for both embedded and external RAMs must be implemented using hierarchical modulo counters. The circuit state and the values of these counters determine when to increment or hold the addresses, and when to accumulate or assert write enables for both the embedded and external memories.

3.4 Milestone 3

Milestone 3 introduces a design challenge that differs fundamentally from the previous two stages. The hardware developed for this milestone does not have a predictable latency for completing all its computations. This variability arises because the time required to process a block during lossless decoding depends on the number of codewords contained in that block, which differ from one block to another in the bitstream. The primary learning objective in this stage is to understand how to manage data-dependent variable latencies in a hardware design.

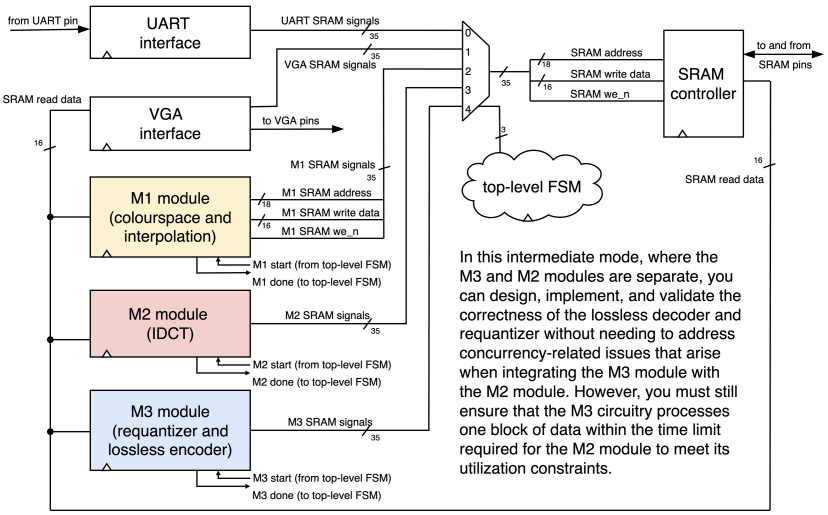
Since each block can require a different amount of processing time, the design must accommodate the worst-case scenario when the logic from Milestone 3 is integrated with Milestone 2. The last two milestones can operate concurrently: while the circuit developed for Milestone 3 processes the current block, the Milestone 2 circuit can operate simultaneously on the preceding block. However, to simplify development and verification, the implementation for this milestone must be carried out in **two** modes. First, the lossless decoding hardware should be designed and verified in isolation while accounting for the worst-case latency of any block (Figures 40 and 41). Only then it should be integrated with the Milestone 2 module to enable concurrent operation (Figures 42 and 43).

It is essential to acknowledge that **Milestone 3 should not be attempted before Milestone 2 has been completed.** A deep understanding of the concepts developed in the earlier milestones is required before addressing the additional challenges introduced in this final stage of the project.

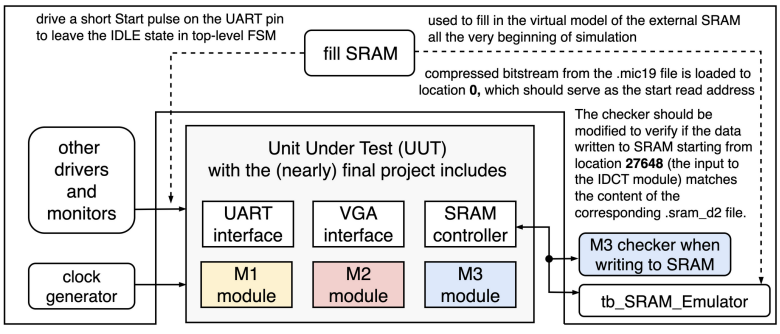


(a) Image decoding flow. (b) Memory map.

Figure 40: Milestone 3 in hardware - intermediate mode where Milestone 3 is implemented standalone.



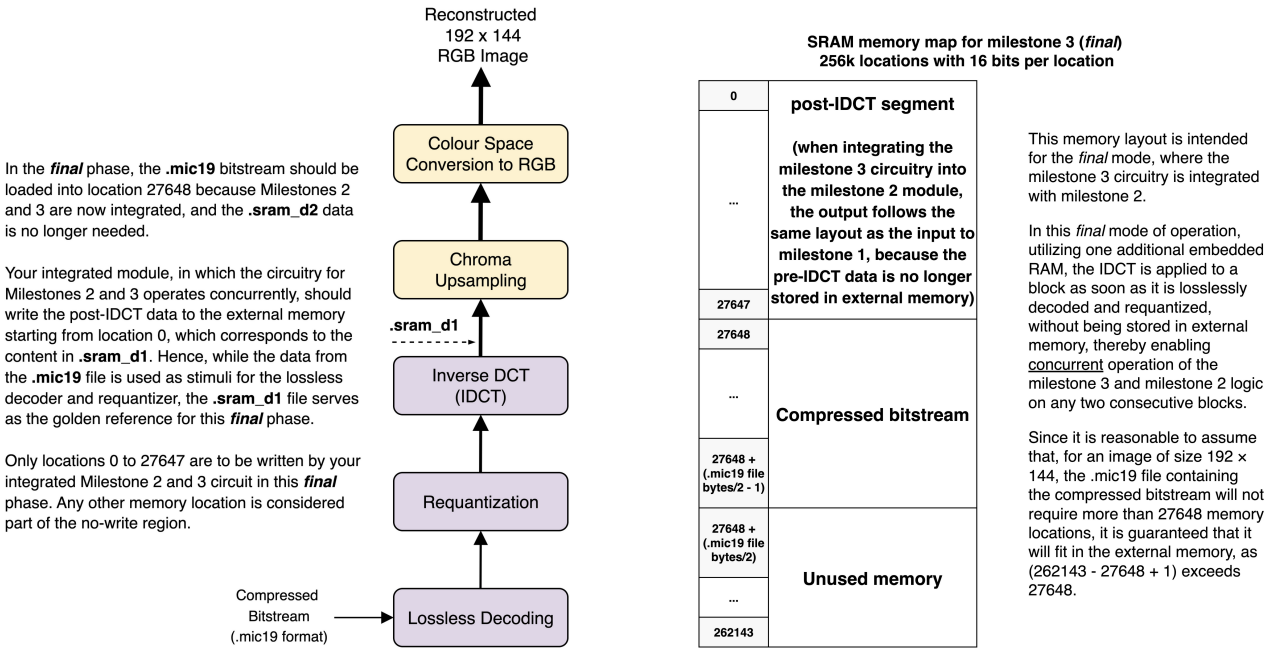
(a) Integration flow.



(b) Verification approach.

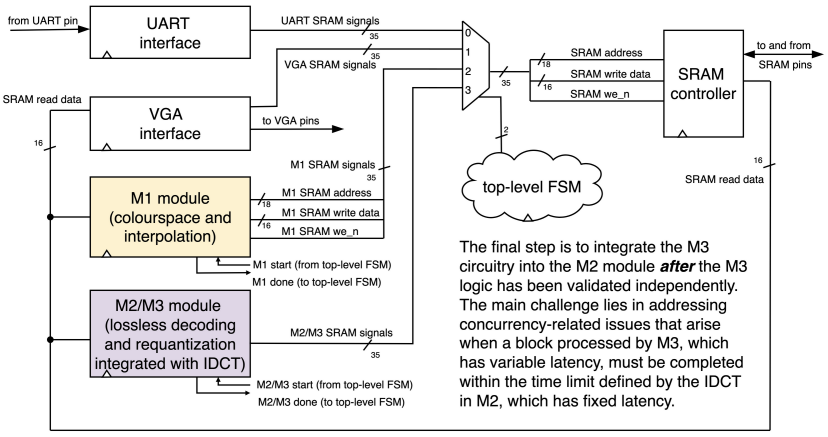
Figure 41: Milestone 3 in the intermediate mode.

During the design and verification of the Milestone 3 logic, it is inevitable that errors requiring correction will arise. Without the intermediate mode, troubleshooting using the post-IDCT samples would be extremely difficult, since the IDCT module obscures the sources of such errors before they appear in the post-IDCT results. To address this, a standalone implementation of Milestone 3 can be verified against the pre-IDCT samples, facilitating root cause analysis of implementation errors. Although this standalone mode does not need to be integrated within the IDCT module, it is still necessary to ensure that, in the worst-case scenario, that is, the maximum number of samples required to losslessly decode and requantize the densest blocks, the design satisfies the upper bounds on latency needed to maintain multiplier utilization from Milestone 2 after integration.

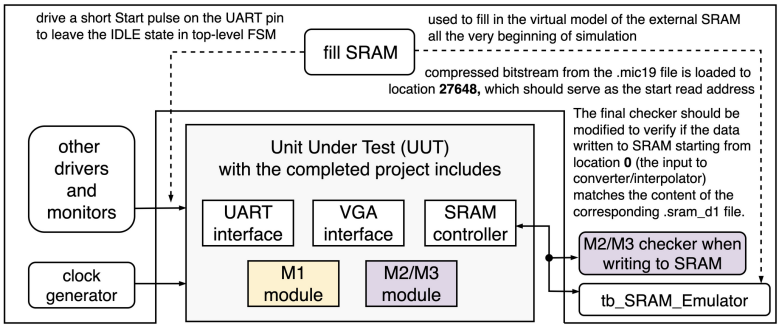


(a) Image decoding flow. (b) Memory map.

Figure 42: Milestone 3 in hardware - final mode where Milestone 3 is integrated into Milestone 2.



(a) Integration flow.



(b) Verification approach.

Figure 43: Milestone 3 in the final mode.

After the correct functionality of the lossless decoder and the requantizer has been confirmed using the intermediate mode of Milestone 3, integration can focus solely on enabling the concurrent operation of the Milestone 3 logic on the current block and the Milestone 2 logic on the previous one. This final mode effectively hides the latency associated with Milestone 3 processing.

One notable change is that the **Fetch** S' task from Milestone 2 can now be removed and replaced by the Milestone 3 logic. To facilitate this integration, in the final mode of the project, one additional embedded RAM is permitted in addition to the 4 embedded RAMs already used in Milestone 2. This extra embedded RAM, together with the one used for storing the S' samples, can operate in a ping pong manner, that is, while one is being written to, the other is being read from, and then their roles are switched.

Below is a list of key challenges to be addressed when designing the hardware for implementing the functionality of the lossless decoder and the requantizer required in Milestone 3.

- The basic lossless decoding process is illustrated in Figure 18 and described in Section 2.2.1. This flow chart can be used to construct the FSM responsible for extracting samples from codewords. One or more pre-IDCT quantized samples can be derived from codewords that contain headers (prefixes) and payloads (small or large values, or codes representing runs of zeros). Because data is processed directly from the bitstream, codewords may span two consecutive memory locations in the external SRAM.

- A key challenge, therefore, is to design and verify a custom FSM that drives the SRAM addresses and increments them only after all bits have been extracted from the current memory location. To avoid long stalls when fetching new data from memory, a multiword shift register can buffer data on-chip (for example, up to four words of 16 bits each). In each clock cycle, the FSM must determine which bits will be processed and shifted out of the multiword shift register and when new data should be loaded from the SRAM, while it continues processing another codeword already preloaded in another 16-bit section from the same multiword shift register.

- As part of the bookkeeping, due to the presence of the end-of-block codeword, this custom FSM must keep track of the position of each sample within a block. Recall that luma and chroma have different block shapes, but all luma blocks are processed first, followed by all chroma blocks. Therefore, the FSM must also track the block count within each image plane before deciding whether to switch block shapes or confirm that decoding is complete. What is common to both luma and chroma is the mapping scheme defined in Table 1. Hence, once the implementation is verified for one component, the other can be completed more rapidly.

- After lossless decoding, the recovered coefficients are passed to the requantizer before being written into the dual-port RAM to await the block IDCT. Because the two possible quantization matrices for each plane (luma in Figure 11 and chroma in Figure 12) contain only powers of two, requantization can be implemented using a variable shifter. The matrix selection is defined by byte offset 3, bit 0, in the 20-byte `.mic19` header (see Table 3). Note that if luma uses quantization matrix 1, chroma also uses its corresponding matrix 1.

- The requantized samples must be placed into the 2D block used by the IDCT in either zig-zag or zag-zig order. These traversal orders are defined for both luma and chroma, for their respective block shapes, in Figure 16. The reference Python source may also be consulted for the exact index mappings.

- The main challenge lies in system integration. From a design standpoint, the lossless decoder, the requantizer, and the block IDCT must interface precisely so that the design can be integrated without violating the multiplier utilization constraint for Milestone 2. This requires careful scheduling of external memory reads and on-chip processing in the multiword shift register to avoid stalls and conflicts on the external memory bus when the block IDCT writes its output to the external SRAM. The main difficulty in Milestone 3 is verification, since the processing time varies across blocks; run time is data dependent, unlike in Milestones 1 and 2, where processing time depends only on image size and not on content.

This project brings together a wide range of design concepts used in practical digital hardware development, from digital filtering and block-based processing to variable-latency, data-dependent operations. It is a demanding but rewarding opportunity to deepen your understanding of digital systems and to strengthen your problem-solving skills. **Enjoy!**

References

[1] ITU-T. Digital Compression and Coding of Continuous-tone Still Images – Requirements and Guidelines. Recommendation T.81, International Telecommunication Union, September 1992. Also published as ISO/IEC IS 10918-1:1994.

[2] Terasic Technologies Inc. *DE2-115 User Manual*. Terasic Technologies Inc., Hsinchu, Taiwan, 2010.

[3] IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, 2005. DOI: 10.1109/IEEESTD.2005.96279.

[4] Nasir Ahmed, T. Natarajan, and K. R. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[5] ITU-R. Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios (Limited Range). Recommendation BT.601, International Telecommunication Union, October 2011. Limited range version.

[6] Fredric J. Harris. On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[7] James F. Kaiser. Nonrecursive Digital Filter Design Using the I_0-sinh Window Function. *Proceedings of the 1974 IEEE International Symposium on Circuits and Systems*, pages 20–23, 1974.