## 40image

1. **Overall architecture of solution**
   Our program will have 2 main functions: image compression and decompression.

   Figure 1 below shows the overall architecture of our program, and the interactions between different components. The key point to note is that compression and decompression are inverse functions of each other; thus, they will share several components that we create.

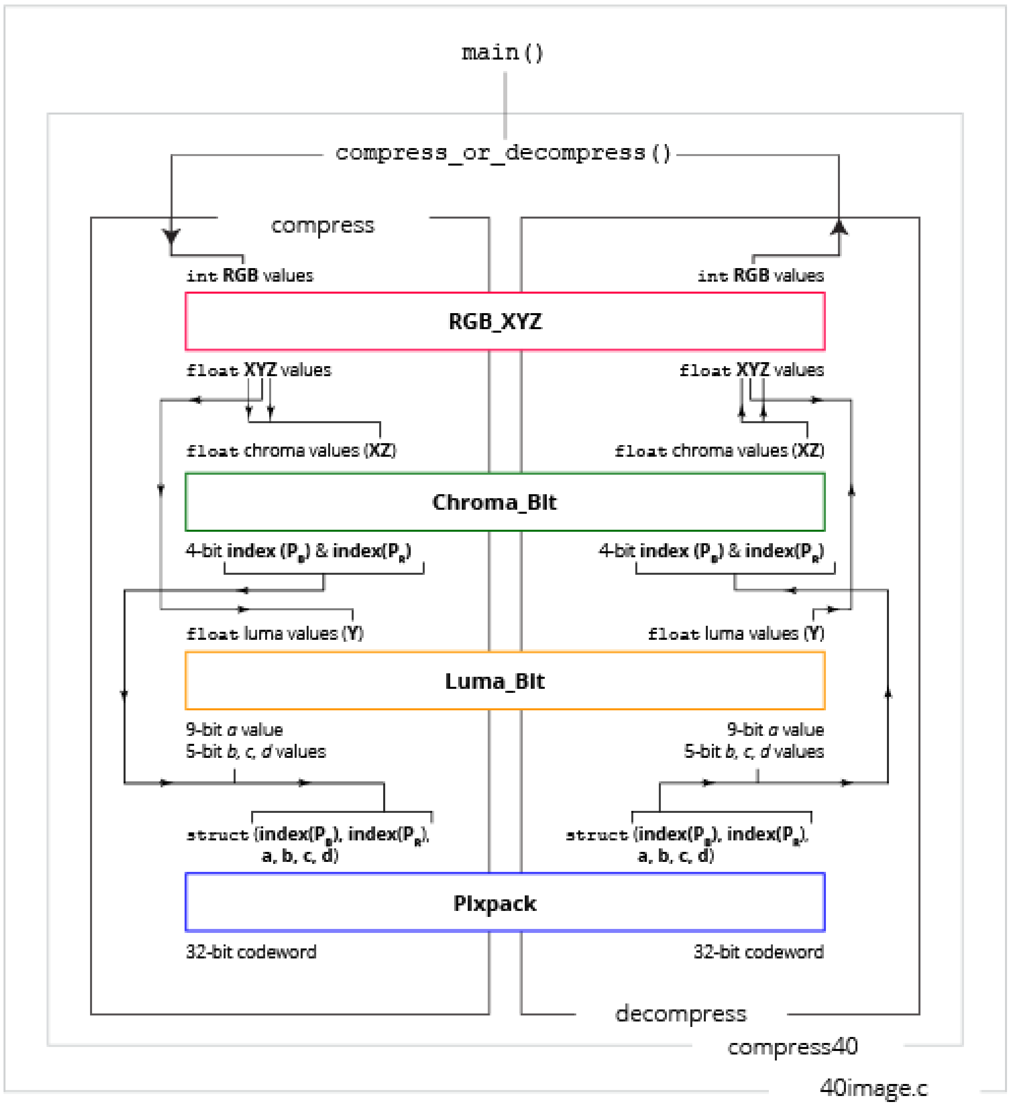

*Figure 1: Overall architecture diagram*

1. `RGB_XYZ`
   - Converts between **integer** RGB values, and **floating point** component-video XYZ values
2. `Chroma_Bit`
   - Converts between **floating point** component-video $P_B/P_R$ values, and **4-bit quantized** average $\overline{P_B}/\overline{P_R}$ values
3. `Luma_Bit`
   - Converts between **floating point** component-video Y values, and a **9-bit unsigned** a value & **5-bit signed** b/c/d values
4. `Pixpack`
   - Converts between a **struct** of individual bit-represented Chroma & Luma values, and a **32-bit codeword** for 4 pixels in the image
   - Note: Uses the `Bitpack` component

We will also reuse the following existing components:
1. `pnm`, for manipulating PPM image files
2. `bitpack`, for packing bit fields into a 64-bit word
3. `arith40`, for converting between index and chroma values
4. `compress40`, for switching between compress and decompress functions

In addition, there will be 1 component that is unique to the compression function.
1. `write_compressed`
   - Writes compressed binary image to standard output in big-endian, row-major order
2. `read_compressed`
   - Reads compressed binary-image output into 32-bit codewords that our component Pixpack can read

The following `structs` will be defined:
1. `RGB_pixel`
   - Represents 1 pixel in an uncompressed PPM file
   - 3 integers: red, green, blue values
2. `RGB_block`
   - Represents 4 pixels (2x2 block) in an uncompressed PPM file
   - 4 `RGB_pixels`
3. `XYZ_pixel`
   - Represents 1 pixel in component video XYZ color space
   - 3 floats: luma, Pb and Pr values
4. `XYZ_block`
   - Represents 4 pixels (2x2 block) in component video XYZ color space
   - 4 `XYZ_pixels`
5. `bit_block`
   - Represents 4 pixels (2x2 block) in their individual bit fields
   - 1 unsigned integer: *a* value
   - 3 signed integers: *b*, *c*, *d* values
   - 2 unsigned indices: index($\overline{P_B}$), index($\overline{P_R}$)

## 2. Individual component architecture

*Note: all functions listed below are* `extern`.

### RGB_XYZ

Invariant: pixels

- `XYZ_block RGB_to_XYZ (RGB_block)`
  Transforms each pixel in block from RGB color space into component video color space.

- `RGB_block XYZ_to_RGB (XYZ_block)`
  Transforms each pixel in block from component video color into RGB color space, quantizing the RGB values to integers in the range [0, 255] with the `quantize_XYZ` function.

### Chroma_Bit

- `XYZ_block average_chroma (XYZ_block)`
  Finds the average chroma values of the pixels in the block, and returns it in a new block.

- `bit_block chroma_to_bit (XYZ_block, bit_block)`
  Converts chroma values of the pixels in the `XYZ_block` into its 4-bit quantized representations, and adds it to the given `bit_block`. Returns the modified `bit_block`. (Calls `Arith40_index_of_chroma` from `arith40.h`.)

- `XYZ_block bit_to_chroma (bit_block, XYZ_block)`
  Converts 4-bit representations of chroma values of the pixels in `bit_block` into floating-point numbers, and adds it to the given `XYZ_block`. Returns the modified `XYZ_block`. (Calls `Arith40_chroma_of_index` from `arith40.h.)`

### Luma_Bit

- `XYZ_block dct (XYZ_block)`
  Converts individual luma values of the pixels in the block into cosine coefficients a/b/c/d, and returns it in a new block.

- `bit_block luma_to_bit(XYZ_block, bit_block, int p, int q)`
  Converts cosine a/b/c/d values of the pixels in the `XYZ_block` into its `p`-bit (for a) and `q`-bit (for b/c/d) quantized representations, and adds it to the given `bit_block`. Returns the modified `bit_block`.

- `XYZ_block bit_to_luma(bit_block, XYZ_block, int p, int q)`
  Converts `p`-bit/`q`-bit quantized representations of cosine a/b/c/d values of the pixels in `bit_block` into floating-point numbers, and adds it to the given `XYZ_block`. Returns the modified `XYZ_block`.

- `XYZ_block inverse_dct (XYZ_block)`
  Converts cosine coefficients a/b/c/d of the pixels in the block into the individual luma values, and returns it in a new block.

**Pixpack**

- `uint64_t pack (bit_block, bit_block)`
  Packs the 2 given `bit_block` into a 64-bit codeword (i.e. a `uint64_t`).

- `bit_block unpack_32bit (uint64_t)`
  Unpacks 32-bits from a 64-bit codeword (i.e. a `uint64_t`), and puts the relevant values into a `bit_block`. This function has to be called twice to fully unpack a 64-bit codeword.

## 3. Each component's testing plan

We will be testing the program in a stepwise function, i.e. we will code one component, test it fully (with the plan listed below), before moving onto the next component. This plan ensures that each component is functional before we move on to the next (leaving the debugging to the end will only leave us distraught).

### File & Image Handling (40image and compress40)

| Function | Test | Result |
|---|---|---|
| Command line handling | Options other than -c and -d. | Use an assertion. |
| Reading a file | Non-ppm files, or non pixmap files. | Use an assertion. |
| Trim | Image that has a width or height of <2px. | We will implement an exception that will be raised in this instance, to tell the client that the program only accepts images that have widths and heights of ≥2px. |
| | Image with odd height and/or width. | Trim should reduce odd dimension by 1 to make it even. Even dimensions should remain unchanged. |

### RGB_XYZ

| Function | Test | Result |
|---|---|---|
| `RGB_to_XYZ` | A 2x2 pixel block w/<br>- Extreme RGB values (e.g. FF0000, 00FF00, 0000FF, FFFFFF)<br>- Same RGB values<br>- Varied/realistic RGB values | XYZ values should adhere to given equations, and be represented as floating point numbers. |
| | - RGB values that are out of range (e.g. <000000 & >FFFFFF) | Use an assertion. |
| `XYZ_to_RGB` | A 2x2 pixel block w/<br>- Extreme XYZ values (Y = 0 or 1, X and Z = ±0.5)<br>- Same XYZ values<br>- Varied/realistic XYZ values | RGB values should adhere to given equations, and be represented as unsigned integers. |
| | - XYZ values that are out of range (e.g. Y < 0 & > 1, X and Z < -0.5 & > 0.5) | Use an assertion. |

**Chroma_Bit**

| **Function** | **Test** | **Result** |
|---|---|---|
| `average_chroma` | A 2x2 pixel block w/<br>- Extreme $P_B$ and $P_R$ values (±0.5)<br>- Same $P_B$ and $P_R$ values<br>- Varied/realistic $P_B$ and $P_R$ values | $P_B$ and $P_R$ values should average out correctly |
| | - $P_B$ and $P_R$ values that are out of range (< -0.5 & > 0.5) | Use an assertion. |
| `chroma_to_bit` | A 2x2 pixel block w/<br>- Extreme $\overline{P_B}$ and $\overline{P_R}$ values (±0.5)<br>- Realistic $\overline{P_B}$ and $\overline{P_R}$ values | Extreme $\overline{P_B}$ and $\overline{P_R}$ values should quantize to ±0.35.<br>All other values will quantize according to the specified values on p8 of the HW specs. |
| | - $P_B$ and $P_R$ values that are out of range (< -0.5 & > 0.5) | Use an assertion. |
| | **Invariants:**<br>- non-chroma values in `bit_block` remain unchanged<br>- sending a float to `Arith40_index_of_chroma` will always return its 4-bit quantized representation in an `unsigned` | |
| `bit_to_chroma` | A block of bit fields w/<br>- Extreme index($\overline{P_B}$) and index($\overline{P_R}$) values (±0.35)<br>- Realistic index($\overline{P_B}$) and index($\overline{P_R}$) values | $\overline{P_B}$ and $\overline{P_R}$ values obtained are equal to the floats which index($\overline{P_B}$) and index($\overline{P_R}$) represent. |
| | - index($\overline{P_B}$) and index($\overline{P_R}$) values that are outside of the valid $P_B$ and $P_R$ range (< -0.5 & > 0.5) | Use an assertion. |
| | **Invariants:**<br>- non-chroma values in `XYZ_block` remain unchanged<br>- sending a 4-bit quantized representation in an `unsigned` to `Arith40_chroma_of_index` will always return its float | |

**Luma_Bit**

| Function | Test | Result |
|---|---|---|
| `dct` | A 2x2 pixel block w/<br>- Extreme Y values (0 or 1)<br>- Same Y values<br>- Varied/realistic Y values | Resulting a, b, c, d values should adhere to given equations. |
| | - Y values that are out of range (< 0 & > 1) | Use an assertion. |
| | **Invariant:** non-luma values in `XYZ_block` remain unchanged | |
| `luma_to_bit` | A 2x2 pixel block w/<br>- Extreme a, b, c, d values (a: 0 or 1, b/c/d: ±½ )<br>- Same b, c, d values<br>- Varied/realistic a, b, c, d values | Extreme b, c, d values (>0.3 or <-0.3) should quantize to ±($2^{width}$ - 1), while extreme a values will quantize to 0 or 511.<br>All other b, c, d values will quantize to integers within a range of ±($2^{width}$ - 1), while a values will quantize to integers within a range of [0, 511]. |
| | - a, b, c, d values values that are out of range (a: < 0 & > 1, b/c/d: <-½ & >½) | Use an assertion. |
| | **Invariant:** non-luma values in `bit_block` remain unchanged | |
| `bit_to_luma` | A block of bit fields w/<br>- Extreme a, b, c, d values<br>- Same b, c, d values<br>- Varied/realistic a, b, c, d values | Floating-point a, b, c, d values obtained are equal to what the individual a, b, c, d bit-fields represent. |
| | - a, b, c, d values that are outside of the valid ranges. | Use an assertion. |
| | **Invariant:** non-luma values in `XYZ_block` remain unchanged | |
| `inverse_dct` | A 2x2 pixel block w/<br>- Extreme a, b, c, d values (a: 0 or 1, b/c/d: ±½ )<br>- Same b, c, d values<br>- Varied/realistic a, b, c, d values | Resulting Y values should adhere to given equations. |
| | - a, b, c, d values values that are out of range (a: < 0 & > 1, b/c/d: <-½ & >½) | Use an assertion. |
| | **Invariant:** non-luma values in `XYZ_block` remain unchanged | |

**Pixpack**

| Function | Test | Result |
|---|---|---|
| `pack` | A `bit_block` w/<br>- Uninitialized values | Use an assertion. |
| | **Invariant:** `bit_block` values remain unchanged | |
| `unpack_32bit` | A 64-bit word w/<br>- All zeroes<br>- All ones<br>- Varied/realistic values | Function is value-agnostic, and should always return the least significant 32 bits in individual bit fields in a bit block. |
| | **Invariant:** the 64-bit word will have its least significant 32 bits replaced by the next 32 bits after this function is called | |

**Bitpack**

The functions in Bitpack are value-agnostic - they only care about returning/modifying the values at the given **index**. Thus, we will give an example 64-bit value, and show the relevant results.

| | |
|---|---|
| **Hex:** | 0xFEEDF00D2CA7BABE |
| **Bin:** | 1111 1110 1110 1101 1111 0000 0000 1101 0010 1100 1010 0111 1011 1010 1011 1110 |

| Func | Test | Result |
|---|---|---|
| Width-test functions<br>**Invariant:** `n` bits can represent unsigned integers in the range [0, $2^n$ - 1] and signed integers in the range [$-2^{n-1}$, $2^{n-1}$ - 1]. | | |
| `fitsu` | (0xFEEDF00D2CA7BABE, 63)<br>(0xFEEDF00D2CA7BABE, 64)<br>(0xFEEDF00D2CA7BABE, 65) | False<br>True<br>False (width must be ≤ word length, CRE) |
| `fitss` | (0xFEEDF00D2CA7BABE, 63)<br>(0xFEEDF00D2CA7BABE, 64)<br>(0xFEEDF00D2CA7BABE, 65) | False<br>False<br>False (CRE) |
| Field-extraction functions<br>**Invariant:** bits extracted are not removed from the word. | | |
| `getu` | (0xFEEDF00D2CA7BABE, 3, 52)<br>(0xFEEDF00D2CA7BABE, 5, 61) | 6<br>CRE (w + lsb > 64) |
| `gets` | (0xFEEDF00D2CA7BABE, 3, 52)<br>(0xFEEDF00D2CA7BABE, 3, -1) | -2<br>CRE (w < 0) |

| Field-update functions<br>**Invariant:** only field of size `width` with least significant bit at `lsb` will be modified, all other bits will remain unchanged | | |
|---|---|---|
| `newu` | (0xFEEDF00D2CA7BABE, 28, 0, 262654581) | 0xFEEDF00D2FA7CA75 |
| `news` | (0xFEEDF00D2CA7BABE, 28, 0, 262654581) | CRE (value does not fit in 28 signed bits) |

4. **Challenge problem**

***Order***

If the order of the bit fields in the 64-bit codeword change, we will have to change the implementation of Pixpack's functions.

Given that we are storing the individual bit fields in a `bit_block`, we expect that we only need to modify the **order** (not the actual functionality) of our code to pack the bit fields into the codeword (or unpack them) using Bitpack.

***Width***

If the width of the bit fields in the 64-bit codeword change, we will have to change the implementation of compress and decompress.

Given that we are passing in the width of the bit fields to the functions that convert between the numeric and bit representations of our pixel data, we expect that we only need to modify the **function calls** (not the actual module's functionality) of our code to correctly quantize our floats.

5. **Lossy compression and decompression**

Compression converts pixels from RGB to XYZ color space, and from luma values to cosine coefficients. Decompression does the inverse of those conversions. However, decompression is not a full reversal of compression.

On compression, information is lost from:

- Averaging chroma values
- Quantization of averaged chroma values
- Quantization of cosine coefficients of luma values

Quantization forces values in a continuous range into discrete values. Thus, the actual values cannot be retained and are lost into the ether of nothingness.

As such, on decompression, approximate, rather than actual, values are obtained from the 'inverse' equations. Information is also lost from:

- Quantization of integer RGB values

On repeated compressions and decompressions on the same image, variability between pixels will be reduced and information can still be lost. However, as the pixels approach the compressed 'average', less and less information will be lost with each cycle of compression and decompression.