<u>Arith Implementation Plan:</u>

Our design implementation for Arith follows the following steps starting with compression of images and then their decompression to original images.

1. Read input from a file or stdin. We expect that the input is a valid PPM image that should be compressed or decompressed. We expect the pnm reader to raise a checked runtime error if it is an invalid input.
2. After input reading, we will have an initialized Pnm_ppm orignal_image 2D array of pixels (red, green, blue) that should be compressed or decompressed.
    a. Test to make sure the width and height of the image are even numbers.
        i. Ensure that trimming works if needed.
3. The order in which we will implement each step in the following diagram is as follows:
    a. RGB pixels -> video color space (compress)
        i. Test conversions of RGB to Y, Pb, and Pr making sure there is no significant loss
            1. Test pure red (255, 0, 0), pure blue (0, 255, 0), pure green (0, 0, 255), black (0, 0, 0), and white (255, 255, 255). Make sure that the correct Y, Pb, and Pr values are returned
    b. Video color space -> RGB pixels (decompress)
        i. Test conversions of Y, Pb, and Pr to RGB value
        ii. Use Y, Pb, and Pr values obtained in the previous step and ensure that they convert to the correct RGB values (within the acceptable rounding limits)
    c. RGB pixels to stdout (decompress)
        i. We will store the compressed image to a file and pass it through our decompression code to get a new ppm image that we can use ppmdiff to compare them
    d. Video color space -> word (compress)
        i. Use bitpack functions to ensure that values are being packed into correct-sized
        ii. Test the function that converts the Y value for each pixel into a, b, c, and d by printing the resulting values and
        iii. Test that the values of a, b, c, and d are between -0.3 and 0.3
        iv. Bitpack functions
            1. Test the width-test functions by inputting different signed and unsigned integers as well as different widths and comparing the result with our own separate calculations.
    e. Word -> video color space (decompress)
        i. Test the function that converts the a, b, c, and d successfully converts into the correct Y values for each pixel.
        ii. Convert the resulting video color space to RGB pixels, then convert the RGB pixels to stdout and use ppmdiff to compare the resulting image to the original image
    f. Print word (compressed binary image)

        i. Ensure that each word is written in row-major order when printing to stdout
        ii. Test that each word is written to the disk in big endian order (using the reverse of the printbytes code from Machine Arithmetic lecture)
    g. Binary image -> video color space (decompress)
        i. Test different denominators and make sure they fit the specified restraints.
        ii. Ensure that returned a, b, c, and d values are within -0.3 and 0.3
4. Final tests
    a. Test that a compressed image is about 3 times smaller than its decompressed counterpart by outputting each into separate files and comparing the size of the files
    b. Test the final compressed and decompressed image by comparing it to the original image using ppmdiff

<u>Bitpack:</u>

We will implement functions for bitpack.h as indicated in the specification. We will test the correctness of the implementation by unit_testing whereby we will pre-calculate the expected values of the signed or unsigned integers, width and lsb value. After pre-calculating we will pass in the required values to the functions and then compare by assertions the expected value to the one obtained from the bitpack functions.

Original Image

RGB pixels -> video color space: (Arguments: A2Methods_mapfun map, Pnm_ppm image, A2Methods_T methods)

1. Convert each pixel from an unsigned int to a float
1. Declare a UArray_2 video_cs of same width and height as the original image but with a size of a struct that contains floats for y, pb and pr
2. Apply the provided linear transformation equations for each float pixel that yields values of y(luminance), pb and pr (side channels)
3. Return: UArray_2 video_cs
4. Information lost: trimming will occur if necessary to ensure that the width and height of the image are even

Video color space -> word: (Arguments: UArray_2 video_cs, A2Methods_mapfun map, A2Methods_T methods)

1. Declare a UArray_2 words of width and height half of UArray_2 video_cs but with a size of a struct holding the values of a,b,c,d,avpb and avpr
2. From the 2-by-2 block in UArray_2 video_cs take the average of value of the the pb and pr for each of the four pixels in the block and store in floats avpb and avpr
3. Use the provided function to convert avpb and avpr into 4-bit values
4. Retrieve the y values from the block and use DCT to get the a, b,c and d values
5. Convert b,c and d to 5 bit signed values by following rules in the specification
6. Packing a,b,c,d,coded avpb and coded avpr into a 32-bit word in the specified format indicated in the specification. Packing is done using the Bitpack implementation
7. The packed 32bit-word components are stored in the struct for each 2-by-2 block and the struct is stored in UArray_2 words
8. Information loss is expected when quantizing the values of b,c, and d
9. Return: UArray_2 words

Print word (compressed binary image) : (Arguments: UArray_2 words, A2Methods_T methods)

1. The compressed binary image is written to stdout by following the format indicated in the specification
    a. Since the machines store the bits in little-endian order, we can use a map function to write the bit values in big-endian
    b. The order writing is in a row-major style hence a methods row-major function can be used to do this
2. Printed binary image
3. No loss of information expected because the function deals with printing the bits
4. Return: None

Compressed Image

Binary image ->video space: (Arguments: A2Methods_T methods, A2Method_mapfun map, Binary image file pointer)

1. Open and read the compressed file containing a binary image as specified in the specification and retrieving the width and height of the image
2. We will try using different denominator values for the PPM image but we will start with 255
3. Assert that the number of bit word is as expected where the number should be equal to width * height. If there are less words, fail with a Checked Runtime Error
4. For each bit word we will retrieve its a,b,c,d, coded avpb and coded avpr values. We will use the provided function to get the uncoded float values of avpb and avpr
5. For a,b,c and d, we will use the provided functions from the specification to get four Y values for each video color space pixel of the 2-by-2 block.
6. Declare a UArray_2 video_cs that will hold structs containing the Y value, avpb and avpr of each pixel
7. Information loss is expected because the values of b,c and d were quantized, hence the obtained y values are not exact as obtained during compression
8. Return: UArray_2 video_cs

Video color space -> RGB pixels(Arguments: A2Methods_T methods, UArray_2 video_cs, A2Methods_mapfun map)

1. From the UArray_2 video_cs from step 2 below, we will apply the inverse transformation equations provided in the spec that will yield RGB pixels from the video color space pixel
2. The obtained RGB pixels will be quantized in a range 0 - 255 (we chose 255 as a default denominator value, we will try different denominator values in the compression section and use the same denominator in this part)
3. Add the quantized RGB pixels to a UArray_2 pixels that is found in the Pnm_ppm struct
4. Information loss is expected depending on the chosen denominator
5. Return UArray_2 pixels containing RGB pixel values

RGB pixels-> stdout: (Arguments: A2Methods_T methods, A2Methods_mapfun map, UArray_2 pixels)

1. Declare a Pnm_ppm image to hold the RGB pixels and initialize all required variables or members for the Pnm_ppm
2. Use Pnm_ppmwite function to print the ppm image to stdout
3. No information loss expected as the function writes RGB pixels to stdout
4. Return: None