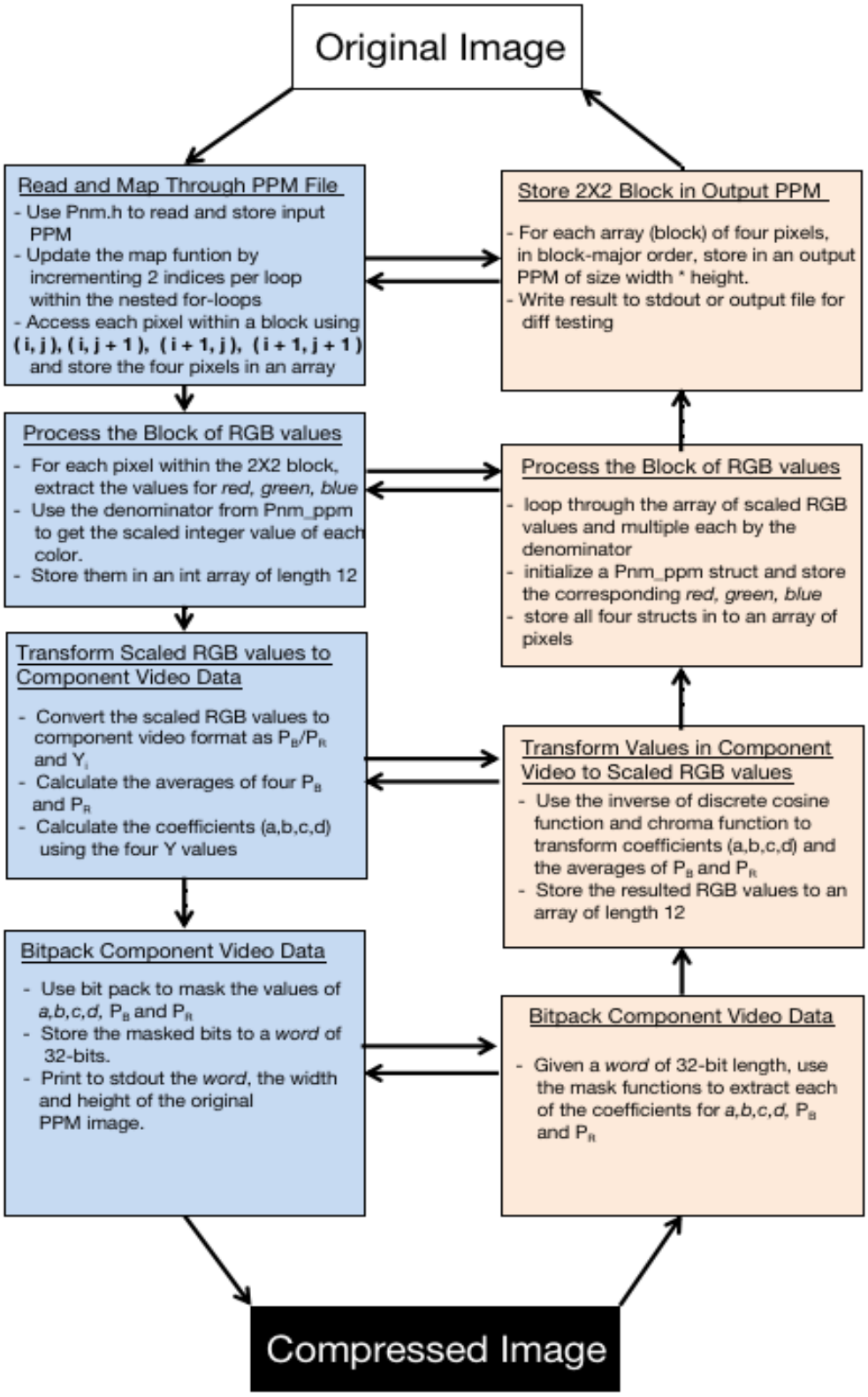
Kevin Yuan & Brian Zeng

Arith Implementation Plan



bitpack.c

Create a module called bitpack which will contain all functions used to compress/decompress a ppm file. Add a main for testing. In main, call a method that returns a file pointer from an input file specified in the command line; if not specified, it should return stdin. If a provided filename can't be opened for reading, output an informative message to stderr and terminate with the EXIT_FAILURE code. Then close the file.

Tests:

1) A valid filename and an invalid filename.
2) An empty file.
3) No arguments/ excessive arguments

Compression step 1: Read and Map Through PPM File

1. Use the ppm_ppmread function to ingest PPM image data from the provided FILE pointer. If the width or height or both are odd numbers, trim it into the closest even number that is less than it.

   Tests:

   1) Use valgrind to make sure that there is no memory leak in each of the following test cases.
   2) A ppm file with a valid header and raster, ensuring the width, height, and every pixel's gray value are within the specified maximum. Then use the ppm_ppmwrite function to print the file to standard output, the output should match the input file.
   3) A non-ppm file.
   4) An empty file.
   5) A ppm with incorrect heading (including height, width, maximum gray value not matching information in the raster or equal to 0)
   6) Valid ppm file as described in test case 2, with odd width, or odd height, or both. The expected ppm file we read in should be trimmed into even height and width.

2. Utilize a 2x2 unit as the foundational element for our mapping function, diverging from the traditional singular slot application. In each iteration, rather than processing a single element, the mapping function will consider a block of 2x2 elements. The loop counters (both for rows and columns) will increment by 2 in each iteration, jumping to the next 2x2 block in the sequence.

   Tests: print out the rgb value of the upper left pixel of each 2x2 matrix, check if it is consistent with the one in the original ppm.

   1) A ppm with even width and height (e.g. 10 x 10)
   2) A narrow ppm (e.g. 2x10 and 10 x 2)
   3) A 2x2 ppm

3. Create an apply function for the mapping function described in the last step. The apply function should take in a 12-slot array and store all RGB values of every pixel in the 2x2 sub-matrix in sequence into the array. (slot 1-3 should represent r, g, b value of topper left pixel, respectively, 4-6 representing upper right pixel, e.t.c.)

   Tests: print out the populated array and check if it is consistent with the one in the original ppm
   1) A ppm with even width and height (e.g. 10 x 10)
   2) A narrow ppm (e.g. 2x10 and 10 x 2)
   3) A 2x2 ppm
   4) Use Decompression step 4 to test the output of this step.

Decompression step 4: Store 2X2 Block in Output PPM

1. Create a function that takes in a single 12-slot array as described in the previous step, and returns a 2x2 matrix that contains data in the array.

   Tests:

   Pass in a 12-slot array that contains 12 unsigned int, each represents a R/G/B value of a single pixel in the ppm. Print out the 2x2 matrix to std output to check whether the matrix is consistent with the value in the matrix. Slot 1-3 should correspond with upper left pixel's R,G,B value, respectively. 4-6 should be upper right, 7-9 should be lower left, and 10-12 should be lower right.

2. Create an empty ppm with width and height of the original image. Utilize the mapping function described in compression step 1.2 to put all 2x2 matrices into the pnm.

   Tests:
   1. Put a single 2x2 matrix into a 2x2 ppm
   2. Put 9 2x2 matrices into a 6x6 ppm
   3. Put 5 2x2 matrices into a 2x10 (test 10x2 as well) ppm.
   4. Use diff to test the output of this step against the input PPM for compression step 1

Compression step 2: Process the Block of RGB values

1. Extract the unsigned values for *red*, *green and blue* from each pixel in the given 2X2 array of pixels using the Struct Pnm_rgb from Pnm.h

   Tests:
   1. Empty pixel with all values as 0
   2. Random pixel with random RGB values.

2. Create a function that takes in an unsigned integer and divides it by another integer, and returns the result.
    a. In this case, the first parameter would be an arbitrary unsigned integer extracted from the pixel, and the second parameter would be the *denominator* from the Pnm_ppm struct.

    Tests:
    1. Manually change denominator to 0
    2. Regular PPM with a valid denominator

3. Store the scaled integer values of each color in each pixel in an int array of length 12

    Tests:
    1. Print out the values of the array to see if the values are consistent
    2. Use Decompression step 3 to test the output of this step.

Decompression step 3: Process the Block of RGB values

1. Iterate over the array of scaled RGB values, multiplying each by the denominator.

    Tests:
    1. Provide a test array and multiply with known denominator, ensuring the outcome matches expectations.
    2. Ensure that resulting RGB values remain within acceptable bounds (0 to max RGB value).

2. Initialize four new Pnm_rgb structs and store the respective *red*, *green*, and *blue* values for each pixel (Pnm_rgb), and then store the four pixels into a 2X2 array

    Tests:
    1. Given a test array of component video values, ensure the reverse-transformed RGB values match the originals (or are acceptably close, given potential rounding).
    2. Print out the values to see if the order of the input values is retained.
    3. Use Compression step 2 to test the output of this step.

Compression step 3: Transform Scaled RGB values to Component Video Data

1. Employ standard conversion formulas to convert the scaled RGB values into the component video format.
    a. For each pixel, the three scaled RGB values will be transformed into three component video values: P_r, P_g, and Y, as defined by the following equations:

$$y = 0.299 * r + 0.587 * g + 0.114 * b;$$
$$pb = -0.168736 * r - 0.331264 * g + 0.5 * b;$$
$$pr = 0.5 * r - 0.418688 * g - 0.081312 * b;$$

Tests:

1. Provide arrays with pre-defined RGB values and compare the resulting component values to expected outcomes.
2. Test with extreme RGB values (0 and maximum) to ensure accurate handling of convergent boundaries.

2. Compute the four coefficients a, b, c, and d using a discrete cosine transformation function. quantized as follows:

$$a = (Y4 + Y3 + Y2 + Y1)/4.0$$
$$b = (Y4 + Y3 - Y2 - Y1)/4.0$$
$$c = (Y4 - Y3 + Y2 - Y1)/4.0$$
$$d = (Y4 - Y3 - Y2 + Y1)/4.0$$

THEN:

i. a is quantized by multiplying it by 511 and rounding, resulting in 9 unsigned bits;
ii. b, c, and d are quantized to scaled integers within the range of -15 to +15.
iii. Any values exceeding this range will be quantized to +/- 15, resulting in 5 unsigned bits

3. Calculate the average of P_b and P_r in all four pixels to obtain the values P_B and P_R using the non-linear quantization to unsigned integers, ranging from 0 to 15. These integers will correspond to indices in a Chroma value set {±0.35, ±0.20, ±0.15, ±0.10, ±0.077, ±0.055, ±0.033, ±0.011}
    a. Determine the index of the value in the set that best approximates P_B and P_R by comparing the actual P_B and P_R chroma values to the chroma value set

4. Use **unsigned Arith40_index_of_chroma(float x)** to convert the chroma value between −0.5 and +0.5 and returns a 4-bit quantized representation of the chroma value

Tests:

1. Provide arrays with pre-defined RGB values and compare the resulting component values to expected outcomes
2. Use edge RGB values (0 and maximum) to ensure conversion boundaries are correctly managed.
3. Calculate the coefficients for arrays with known Y values and validate against expected outcomes.
4. Use Decompression step 2 to test the output of this step.

Decompression step 2: Transform Values in Component Video to Scaled RGB values

1. Employ the inverse of the discrete cosine transform and chroma function to reconvert the coefficients a, b, c, d and the averages of P_g and P_r back to their RGB format.

$$r = 1.0 * y + 0.0 * pb + 1.402 * pr;$$
$$g = 1.0 * y - 0.344136 * pb - 0.714136 * pr;$$
$$b = 1.0 * y + 1.772 * pb + 0.0 * pr;$$

$$Y1 = a - b - c + d$$
$$Y2 = a - b + c - d$$
$$Y3 = a + b - c - d$$
$$Y4 = a + b + c + d$$

Tests:

1. Utilize coefficient and average values known to correspond to certain RGB values and validate the transformation process.

2. Store the resultant RGB values into a 12-slot array, ensuring each pixel's RGB values are stored sequentially.

Tests:

1. Compare the stored values against the expected RGB values for accuracy.
2. Use Compression step 3 to test the output of this step.

Compression step 4: Bitpack Component Video Data

1. Implement the **Width-test()** functions
    a. Implement the **Bitpack_fitsu()** function:
        i. This function will determine if a given integer can be represented using the specified number of bits in unsigned format.
        ii. If the value is between 0 and (2^width) - 1, return true. Otherwise, return false.
    b. Implement the **Bitpack_fitss()** function:
        i. This function will determine if a given integer can be represented using the specified number of bits in two's-complement representation.
        ii. If the value is between -(2^(width-1)) and (2^(width-1)) - 1, return true. Otherwise, return false.

Test:

1. Input: Value = 5, Width = 3;
   Expected Output: true for Bitpack_fitsu (Since 5 in binary is 101 which fits in 3 bits) and false for Bitpack_fitsu (Since 3 bits can represent signed integers only in the range −4 to 3)

2. Input: Value = 10, Width = 3;
   Expected Output: false for both (Since 10 in binary is 1010 which doesn't fit in 3 bits)

3. Input: Value = -5, Width = 3 (test for Bitpack_fitss);
   Expected Output: false (Since -5 in two's complement binary can't be represented in just 3 bits)

2. Implement **Exception** Handling

a. Implement the exception handling mechanism in the functions to raise the **Bitpack_Overflow** exception whenever there is an overflow condition.

Tests:

1. Input: Word = 01101101, Width = 2, lsb = 1, Value = 100
   Expected Output: Raise Bitpack_Overflow exception (Since the value 100 can't be packed in width of 2 bits)

3. Implement the Field-update functions
    a. Implement the **Bitpack_newu()** function:
        i. Replace the specified bits in the word with a new value in unsigned format and return the updated word.
    b. Implement the **Bitpack_news()** function:
        i. Replace the specified bits in the word with a new value in signed format and return the updated word.

Both function should:

1. Throw runtime error if w is not within the range of 0 to 64 inclusive. In the same manner, a checked run-time error should be initiated when using these functions with a width w and lsb combination where their sum exceeds 64.
2. If Bitpack_news receives a value that cannot be accommodated in the specified width of signed bits, it should trigger the Bitpack Overflow exception (utilizing Hanson's RAISE macro from the Except interface). In the same way, when Bitpack_newu is provided a value that exceeds the defined width for unsigned bits, it should also prompt the Bitpack Overflow exception.

Test for Bitpack_newu:

1. Input: Word = 01101101, Width = 4, lsb = 2, Value = 1001
   Expected Output: 01100101 (Replace bits from position 2 to 5 with 1001)

Test for Bitpack_news function:

1. Input: Word = 11001101, Width = 4, lsb = 2, Value = 2
   Expected Output: 11001001 (Replace bits from position 2 to 5 with binary representation of 2)

Test for both:

1. Input: Word = 01101101, Width = 2, lsb = 1, Value = 100
   Expected Output: Raise Bitpack_Overflow exception (Since the value 100 can't be packed in width of 2 bits)

4. **Assembling** the functions:
    a. From compression part 3 we got the binary representation of a, b, c, d, P_B, P_R. For bit-packing, we simply call Field-update function and pass in the corresponding LSB, width, and binary information to pack them into a 32-bit word. Print the word to standard output

Test:

1. Use test cases from compression step 3 as binary representation of a, b, c, d, P_B, P_R and store them into a word with the method described above. Check if the printed word is consistent with expected output.

Decompression step 1:

1. Implement the **Field-extraction** functions

- Implement the Bitpack_getu() function:
  - Extract and return the specified number of bits from the given word in unsigned format. Should be c.r.e. if w is not in the range of $0 \leq w \leq 64$ or $w + lsb > 64$

- Implement the Bitpack_gets function:
  - Extract and return the specified number of bits from the given word in signed format. Should be c.r.e. if w is not in the range of $0 \leq w \leq 64$ or $w + lsb > 64$

Test for Bitpack_getu function:
Input: Word = 01101101, Width = 4, lsb = 2
Expected Output: 1101 (Extracted bits from position 2 to position 5)
Test for Bitpack_gets function:
Input: Word = 11001101, Width = 4, lsb = 2
Expected Output: -3 (Extracted bits from position 2 to position 5 in two's complement)

Test for both:
1. Input: Word = 11001101, Width = 0, lsb = 2
Expected Output: 0

2. Input: Word = 11001101, Width = 65, lsb = 2
Expected Output: Runtime error

3. Input: Word = 11001101, Width = 60, lsb = 20
Expected Output: Runtime error

2.

Use field extraction function to get a, b, c, d, P_B, P_R by passing in the word and the lsb - width pair according to the table below:

Value	Type	Width	LSB
a	Unsigned scaled integer	9 bits	23
b	Signed scaled integer	5 bits	18
c	Signed scaled integer	5 bits	13
d	Signed scaled integer	5 bits	8
$index(\overline{P_B})$	Unsigned index	4 bits	4
$index(\overline{P_R})$	Unsigned index	4 bits	0

Test:
Use compression step 4 to test the output of this step. The decompressed binary representation of the 6 value should be consistent with themselves before compression.