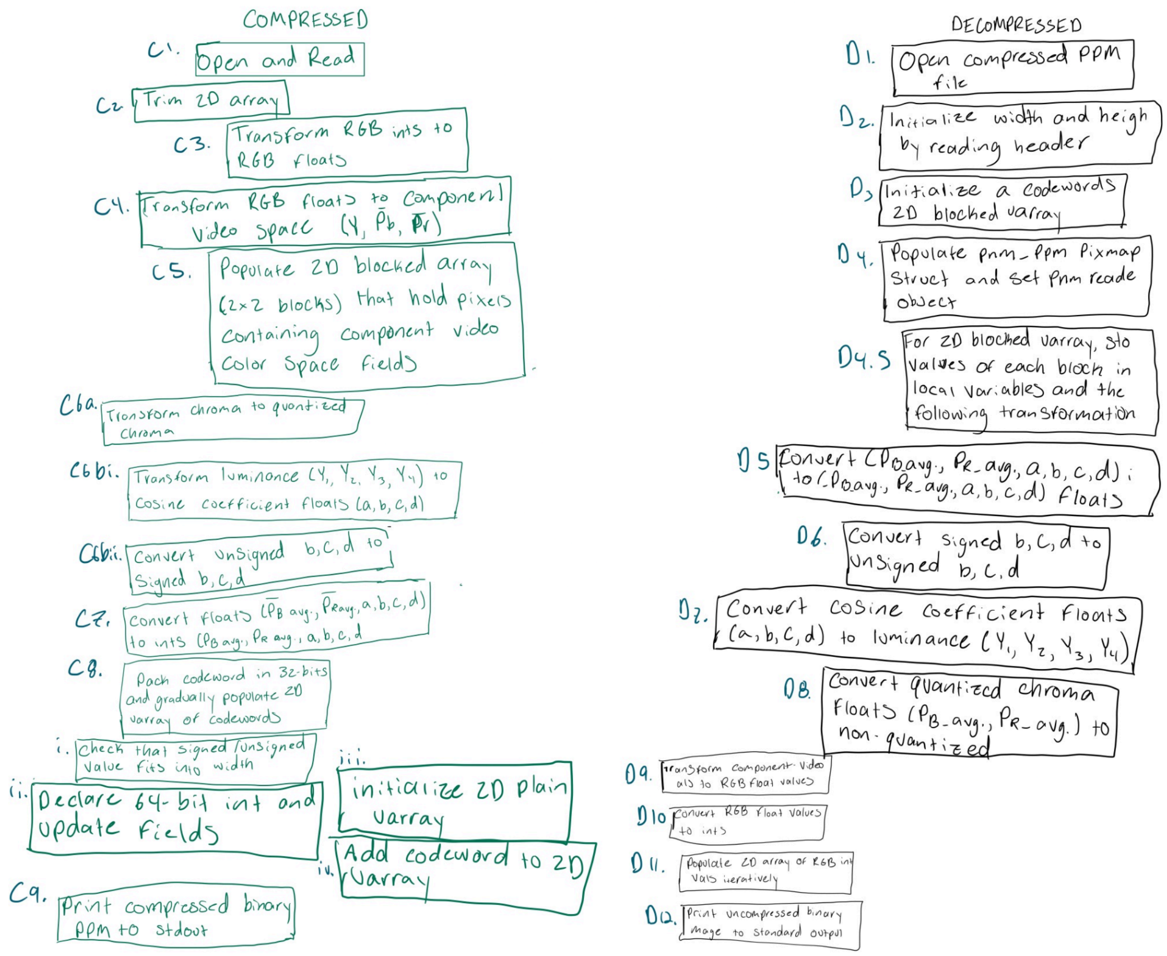
Arith Design Doc
Step Descriptions/Details
Contents: Box diagram image, box/step details, implementation and testing plan


COMPRESSED
C1. Open and Read
C2. Trim 2D array
C3. Transform RGB ints to RGB floats
C4. Transform RGB floats to Component video space (Y, P̄b, P̄r)
C5. Populate 2D blocked array (2×2 blocks) that hold pixels containing component video color space fields
C6a. Transform chroma to quantized chroma
C6bi. Transform luminance (Y1, Y2, Y3, Y4) to cosine coefficient floats (a, b, c, d)
C6bii. Convert unsigned b, c, d to signed b, c, d
C7. Convert floats (P̄B avg., P̄R avg., a, b, c, d) to ints (PB avg., PR avg., a, b, c, d
C8. Pack codeword in 32-bits and gradually populate 2D varray of codewords
i. Check that signed/unsigned value fits into width
ii. Declare 64-bit int and update fields
iii. initialize 2D plain Uarray
iv. Add codeword to 2D Uarray
C9. Print compressed binary ppm to stdout
DECOMPRESSED
D1. Open compressed PPM file
D2. Initialize width and heigh by reading header
D3. Initialize a codewords 2D blocked varray
D4. Populate pnm_ppm pixmap struct and set pnm reade object
D4.5 For 2D blocked varray, sto values of each block in local variables and the following transformation
D5. Convert (PB avg., PR_avg., a, b, c, d) i to (PB avg., PR_avg., a, b, c, d) floats
D6. Convert signed b, c, d to unsigned b, c, d
D7. Convert cosine coefficient floats (a, b, c, d) to luminance (Y1, Y2, Y3, Y4)
D8. Convert quantized chroma floats (PB_avg., PR_avg.) to non-quantized
D9. Transform component video als to RGB float values
D10 Convert RGB float values to ints
D11. Populate 2D array of RGB int vals iteratively
D12. Print uncompressed binary mage to standard output

I. Box/Step Details:

Compression

C1.) Open and Read input file with PPM reader: use pnm reader to open valid file and populate the pixels array. Each cell will correspond to a pixel with r,g,b fields.

a) Input: file (commandline arg) if given, or standard input
b) Output: Initialized pnm ppm struct
c) No info lost
d) Testing: Assert that pointers are not null, print the contents of pixels array and check dimensions to ensure that it has read in. Test files that trigger all different paths (ie: std input, valid file, invalid file, very small files, very large files, etc). Write binary to an output file and diff test with

C2.) Trim 2D array width and height until even dimensions: check the width and height fields initialized in step 1, and if not even, trim pixels array until dimensions are even

e) Input: pnm struct (containing pixels 2D array, width, height)
f) Output: Void, but updates pixels array
g) Info lost: if dimensions are not initially even, we will lose some of the info on the borders (at most need to trim two sides of the 2D array)
h) Testing:
    i) Assert that new dimensions have been updated
    ii) Run valgrind on odd-dimension image to ensure that no memory leaks occur after a trim.

For each pixel in pixels array, iterate through in row major order and perform the following transformations. Once in video color space, put into blocked array containing struct with (Y, P_B, P_R):

C3.) Transform RGB ints to float: Store floating point representations of RGB fields for a pixel

i) Input: A position in the pixels array
j) Output: a struct of RGB floats
k) Loss of info can occur: Both unsigned ints and floats use 32 bits to represent the data, but since floats are not discrete (ie: can have decimals) they represent a broader range of data and use some bits for sign and fraction. Thus, precision of info may be lost during conversion rounding
l) Testing: Compare int value to converted float value, test on very small and very large int values

C4) Transform RGB float pixel to component video color space: Use RGB values to get luminance and chromaticity representation Y, P_B, P_R using linear transformation equations

m) Input: RGB float struct
n) Output: component video color space struct (floats)
o) No info lost
p) Testing: see implementation plan for testing details

C5) Initialize and populate component video color space 2D blocked array: Initialize a blocked 2D array with the blocksize of 2, trimmed width and height. Populate each cell to hold a transformed pixel (ie: struct with Y, P_B, P_R fields).

q) Input: Current plain pixels 2D array index, struct with component video transformation of RGB pixel at that index
r) Output: 2D blocked array with component video pixels in each cell
s) No info lost in initialization/population of blocked 2d array
t) Testing:
    i) Run valgrind before and after implementing to ensure no memory leaks occur after creating blocked array. Repeat for different image sizes
    ii) Assert correct dimensions, and print the original pixels array and the pixels array after swapping to ensure that data transfer was successful.
    iii) Test on very small and very large ppm files (ex: if pixels smaller than blocksize, very large images, etc)

Iterate through blocked array in block order, and for each 2x2 block:

C6 a.) **Transform chroma to quantized chroma**: Call this for P_B and P_R. Calculates the average from four cell values in a 2x2 block, and call given function to convert to quantized 4-bit representations

u) Input: P_B and P_R vals in current blocked 2D array block
v) Output: transformed 4-bit values $P_{B\_avg,}$, P_{R_avg}
w) Info lost: Mapping to a 4 bit representation, so data representation may compromise certain precision for new representation (ie: mapping to a different range of values)
x) Testing: assert that output values are in the range of block input vals, respectively (since output should be an average of the 4 cell values) (see further diff test details in implementation plan)

C6 bi.) **Transform luminance to cosine coefficients:** Using discrete cosine transformation to transform the luminance values in the block to cosine coefficients a,b,c,d.

y) Input: $Y_{1,}$ $Y_{2,}$ $Y_{3,}$ Y_4 vals in current uarray2b block
z) Output: Update y values to hold cosine coefficients: a, b, c, d (ie: represent average brightness, degrees to which brightness increases in directionals)
aa) No info lost
bb) Testing: Print to check that transformed values are reasonable numbers (see details in implementation plan for more structured testing)

C6 bii.) **Transform brightness degree coefficients :** b, c, d

cc) Input: b, c, d (brightness degree coefficients from prev step)
dd) Output: b, c, d 5 bit signed representation
ee) Info lost: This conversion uses the assumption that converted 5-bit signed values lie between −0.3 and 0.3 (essentially mapping to a range that sometimes uses rounding), meaning we lose information in the case that values are outside the range ±0.3. However, more precise in non-rare cases.
ff) Testing: inputs and print outputs to compare values by eye (see implementation plan for further testing)... test on very large/very small values, different negative/non-negative input combinations

C7.) **Convert floats to ints:** Before packing into codeword, get integer representation of float for a block containing floats: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d)

gg) Input: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d) floats
hh) Output: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d) ints
ii) Info lost: Integers do not hold fraction, so can lose data precision in this step from rounding
jj) Testing: compare input and output on large and small floats to ensure that transformations are successful

C8.) **Pack codeword in 32 bits**

kk) Input: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d) ints
ll) Output: Compressed 32-bit codeword containing input info, in which each value can be accessed based on LSB and width. Value with lowest LSB goes in rightmost spot, next value has width index in 32-bit word, etc…
mm) Info lost
nn) Testing: test C8 components individually, then diff test on full output vals (see implementation plan)

C8 i) Check that signed/unsigned value fits into width: For each input in C8 (prev step), ensure that the value fits in the width. Call one of the following based on value type (signed or unsigned)

→ bool Bitpack_fitsu(uint64_t n, unsigned width)
→ bool Bitpack_fitss(int64_t n, unsigned width);

oo) Input: int value and width (ie: corresponding to chart on spec pg 5)
pp) Output: True if fits
qq) No info lost
rr) Testing: Test on values that will return true and on values that will return false. Test edge conditions (ie: largest int that will fit into a width, width >> int, signed/unsigned int inputs)

C8 ii) Declare 64-bit int and update fields: Initialize a 64-bit codeword with zero. For each input in C8 (prev step), update value from (LSB to LSB+width) index. Call one of the following based on value type (signed or unsigned)

→ uint64_t Bitpack_newu(uint64_t word, unsigned width, unsigned lsb, uint64_t value)
→ uint64_t Bitpack_news(uint64_t word, unsigned width, unsigned lsb, int64_t value);

ss) Input: int64_t codeword, width of value, lsb of value, and value
tt) Output: None, but update 64 bit codeword to contain value
uu) No info lost
vv) Testing: compare fields before and after initialization

C8 iii) Initialize 2D plain uarray with width = (width of blocked arr2D / blocksize) and height = (height of blocked arr2D / blocksize). This will be the dimensions of the number codewords. No info lost. Test by checking dimensions, examining output, asserting non-null. Test edge cases for very small/very large/ invalid dimensions

C8 iv) Add codeword to 2D uarray: We will iterate through 2x2 blocks, and the codewords will be appended to a list of 64 bit integers such that they will be added in "row major" order (ie: iterating through blocks horizontally)

ww) Input: uint_t codeword
xx) Output: Adds a packed codeword iteratively to plain uarray2 (from prev step) in row major order
yy) No info lost
zz) Testing: Check that not null/ codeword values initialized to correct positions, check on cases where minimum num codewords, as well as very large dimensions

C9.) Print compressed binary to standard output: Once all blocks have been packed into codewords, and those codewords have been added to a list, we can print the header and iterate through the list, reading each code word in big-endian order to standard output.

aaa) Input: List of packed codewords
bbb) Output: Writes compressed binary of file to standard output
ccc) Info lost: no
ddd) Testing: Take a compressed file and decompress it. Then use this as the input for a compression and direct standard output to a file. Diff this file with the original compressed image. Run valgrind to ensure all allocated memory has been freed.

Decompression

D1.) Open Compressed PPM file: use pnm reader to open valid file and populate the pixels array. Each cell will correspond to a pixel with r,g,b fields.

a) Input: PPM file (commandline arg) if given, or standard input
b) Output: Initialized pnm ppm struct
c) No info lost
d) Testing: Assert that pointers are not null, print the contents of pixels array and check dimensions to ensure that it has read in. Test files that trigger all different paths (ie: std input, valid file, invalid file, very small files, very large files, etc).

D2.) Initialize width and height by reading header: Set width and height read in from compressed PPM file header

e) Input: PPM file with given header format
f) Output: width and height variables
g) No info lost
h) Testing: assert that width and height are expected values based on input file, test on files with invalid width/height, minimum width/height, very large files

D3.) Initialize a codewords 2D blocked uarray by reading in file: Initialize a 2D array of D2 width and height containing codewords

i) Input: width, height, blocksize, PPM file
j) Output: blocked uarray with 2x2 blocks
k) No info lost
l) Testing: Test on different files as in prev step, assert that array is not null

D4.) Populate pnm_ppm pixmap struct and set to pnm reader object: populate struct that will hold decompressed pixels with attributes from prev step, with width and height multiplied by blocksize (2), since decompressed will have 4 pixels for each codeword (since codewords correspond to 2x2 blocks)

m) Input: adjusted_width, adjusted_height, denominator, methods, unpopulated array with adjusted width and height
n) Output: initialized pnm ppm struct (ppm rdr object)
o) No info lost
p) Testing: Test on different dimensions (valid, invalid, etc) and assert that fields were initialized to expected values. Assert that pixels array is not null

D 4.5) For 2D blocked uarray, store values of each block in local variables and do the following transformations:

D5 b.) Convert ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d) ints to ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d) floats: get float representation of integers for a block containing ints: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d)

q) Input: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d) ints
r) Output: ($P_{B\_avg,}$, P_{R_avg}, a , b, c, d) floats
s) Info lost: Integers do not hold fraction, so can lose data precision in this step from rounding (ie: different range of values from int domain to float range)
t) Testing: compare input and output on large and small floats to ensure that transformations are successful, note changes due to mapping different ranges/rounding

D6.) D6: Convert signed b,c,d to unsigned b,c,d

u) Input: signed b, c, d vals
v) Output: unsigned b, c, d vals

w) Info may be lost here since mapping range for these values is split in half
x) Testing: Test on very large values and very small values, note differences in output/ when data is lost, test on invalid inputs (ie signed inputs), assert that output for valid inputs is value with other sign

D7.) Convert cosine coefficient floats (a,b,c,d) to luminance (Y1, Y2, Y3, Y4): Using inverse of discrete cosine transformation to transform the cosine coefficients to luminance values for each block

y) Input: Cosine coefficients in block: a, b, c, d (ie: represent average brightness, degrees to which brightness increases in directionals)
z) Output: $Y_{1,}$ $Y_{2,}$ $Y_{3,}$ Y_4 luminance values
aa) No info lost
bb) Testing: see implementation plan for details

D8.) Convert quantized chroma floats ($P_{B\_avg,}$, P_{R_avg}) to non-quantized: Takes quantized 4-bit representations and converts them into non-quantized chroma vals using provided function (float Arith40_chroma_of_index(unsigned curr_quantized_chroma))

cc) Input: Quantized unsigned $P_{B\_avg,}$, P_{R_avg}
dd) Output: non-quantized unsigned $P_{B\_avg,}$, P_{R_avg}
ee) No data loss
ff) Testing: See implementation plan for testing details

D9.) Transform component-video vals to RGB float values: Use luminance and chromaticity representations (Y, $P_{B\_avg,}$, P_{R_avg}) to get RGB values (linear transformation equations)

gg) Input: component video color space struct (Y, $P_{B\_avg,}$, P_{R_avg})
hh) Output: RGB float struct
ii) No info lost
jj) Testing: see implementation for testing details

D10.) Convert RGB float values to ints: Turn float RGB values into ints

kk) Input: a struct of RGB floats, pixels array (which will be iterated through in row major order)
ll) Output: rgb int struct
mm) Loss of info can occur: Both unsigned ints and floats use 32 bits to represent the data, but since floats are not discrete (ie: can have decimals) they represent a broader range of data and use some bits for sign and fraction. Thus, precision of info may be lost during conversion rounding.
nn) Testing: Compare float value to converted int value, test on very small and very large float values

D11.) Populate 2D array of RGB int vals iteratively updates 2D array pixels (for which struct was previously initialized) to contain corresponding RGB values

oo) Input: A 2D pixels array, rgb int struct from above step
pp) Output: Update rgb fields for pixel at corresponding location in pixels array
qq) No info lost
rr) Testing: check that all fields have been updated, test on large and small input files, edge dimensions, and run valgrind to check for unintended memory leaks

D12.) **Print uncompressed binary image to standard output:** Once pixels array has been updated with uncompressed rgb pixel data, we can use the ppm reader to print binary to output with correct header

ss) Input: ppm object containing pixels array and its attributes
tt) Output: Writes uncompressed binary of file to standard output
uu) No info lost
vv) Testing: Run valgrind to check that no memory leaks occur/ all allocated data has been freed. See implementation plan for further diff testing

Note for us: Computers use little endian but they ask us to use big-endian (watch out for this). First thing is to take Big endian and make it into little endian

II. Implementation Order and PPM Diff Testing Plan: (see diagram descriptions for step references, where C refers to compression steps and D refers to decompression steps). Blue indicates outputs of compression function, and orange denotes inputs for decompression steps. Diff testing is done for a pair of steps (compression and analogous decompression step). E output from diff test will be recorded after tests for comparison with future E values. While certain steps will have specific testing as outlined in step description, this outlines intermittent diff testing plan structure timed by steps implementation plan.

1) **C1: Open and Read *(uncompressed input PPM):*** *Populates* 2D pixels array
2) *D12: Print uncompressed PPM:* can direct output to a **decompressed PPM file**
   a) **Testing:** After C1 and D12 implemented, call D12 after C1. Then run quantitative ppmdiff(from lab) on **decompressed PPM output file** with input ppm. At this point, files should be identical since no data loss.
3) **C2: Trim 2D array dimensions until even:** *array contains* RGB int *values for each pixel*
4) *D11: Populate 2D array of RGB int vals*
   a) **Testing:** After C2 and D11 finished, call decompress on C2 output. D11 uses C2 output as input and runs D12, which again outputs a decompressed PPM. Run ppm diff with C1 input PPM; now, expect a **small difference**, as some data was lost in the trim.
5) **C3: Transform RGB ints to RGB floats**
6) *D10: Convert RGB float values to RGB ints*
   a) **Testing:** After C3 and D10, call decompress on C3 output, which will be used by D10 and run other functions through D12 (outputs an uncompressed PPM). Run ppm diff on decompression output and original uncompressed input PPM. Expect a very slight increase in ppmdiff *E* output, since some rounding data may be lost.
7) **C4: Transform RGB floats to Component Video Space (Y, Pb, Pr)**
8) *D9: Transform Component video vals to RGB float values*
   a) **Testing:** After C4 and D9, call decompress on C4 output. Run ppm diff on decompression output and compression input. Expect no change in *E* from prev testing.

9) **C5: Initialize/ populate 2D blocked array (2x2 blocks) that holds pixels containing component video color space fields**
    a) *See step details for testing*

10) **C6 a : Transform chroma vals (Pb, Pr) to quantized chroma floats** ($P_{B\_avg,}$, P_{R_avg})
11) **C6 bi : Transform luminance (Y1, Y2, Y3, Y4) to cosine coefficient floats** (a, b , c, d)
12) *D8: Convert quantized chroma floats* ($P_{B\_avg,}$, P_{R_avg}) *to chroma (Pb, Pr)*
13) *D7: Convert cosine coefficient floats (a,b,c,d) to luminance (Y1, Y2, Y3, Y4)*
14) *D 4.5) Store values of each block in local variables (ie: "combined C6 input")* → this step will be modified throughout step 6 as combined input values are transformed in step 6 components
    a) **Testing:** After C6 a/bi and D8/D7, call decompress on combined C6 output. Run ppm diff on decompression output and compression input. Expect no change in *E* from prev testing, since no data should be lost in transformation.
15) **C6 bii : Convert unsigned b, c, d to signed b,c,d**
16) *D6: Convert signed b,c,d to unsigned b,c,d*
    a) **Testing:** Same as above testing, but now uses converted b,c,d values with a val and chroma vals from prev testing step. Since this conversion may result in some data loss (see in step details), now expect a slight increase in E from ppmdiff.
17) **C7: Covert floats ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d) to ints ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d)**
18) *D5: Convert ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d) ints to ($P_{B\_avg,}$, $P_{R\_avg,}$ a, b, c, d) floats*
    a) **Testing:** After C7 and D5, call decompress on C7 output. Run ppm diff on decompression output and compression input. Expect slight increase in E from prev testing, since rounding in conversion may cause some information loss.

19) **C8 i-iv: Pack codeword in 32-bits and gradually populate 2D uarray of codewords**
20) *D4: Populate pnm_ppm pixmap struct and set to pnm reader object: (see step details for testing info and details) → this will take adjusted* ***dimensions of codewords 2D uarray***
21) ***D3: Initialize a codewords 2D blocked uarray*** *(indiv step testing in step details)*
    a) **Testing:** After C8 and D3, call decompress on C8 output. Run ppm diff on decompression output and compression input. Expect no change in E from prev testing, since we are packing data into a codeword without losing info.
22) **C9: Print compressed binary PPM to stdout**
23) *D2: Initialize width and height from reading header of PPM file*
24) *D1: Open compressed PPM file*
    a) **Testing:** After completing C9 and D2, call decompress on the PPM output file. Then ppmdiff test compression input with decompression output. Expect no additional increases in E value.
    b) **Testing:** After completing D1, perform same test. Again, expect no change in E.