Jose Lemus (jlemus01), Rex Umezuruike (rumezu01)
Homework #4
Design Document

Problem Statement

The goal is to create a compressor and decompressor capable of converting a full color portable pixmap into a compressed binary image file that is about 3 times smaller, or converting a compressed binary image file into a portable pixmap.

Use Cases

Given a portable pixmap test1.ppm of size 300 bytes, feeding it into the program with the command 40image -d test1.ppm should output a compressed binary image file of about 100 bytes in size.

Assumptions and constraints

The program operates under the assumption that it will receive a command -d or -c on the command line, signifying either decompression or compression. The program must also receive a valid ppm image, or compressed binary image from either standard input, or as a command line argument.
The program also runs under the constraint that the image should not generate values for $|a|, |b|, |c|$ during the discrete cosine operations that fall far out of the -0.3 and 0.3 range.

General architecture

The program is split into two main compartments, compress and decompress. These two compartments are both dependent on the same set of components. These components carry out individual functions such as converting to and from RGB and component color space.

Individual components

40Image

Reading in the command (and potentially the file) or reading from stdin.
Opening the file, checking that it exists, and doing other basic error checking.

Compress40

void compress (FILE file); Takes a full-color ppm as an input file and prints the compressed version to standard output with a header. It is a CRE to be passed a NULL file pointer.
void decompress (FILE file); Takes a compressed ppm as an input file and prints the decompressed version to standard output with a header. It is a CRE to be passed a NULL file pointer.

ConvertColorSpace

rgb_comp_convert(void *rgbStruct, void *compPixelStruct); Takes a pointer to a Pnm_rgb struct with valid RGB values and a compPixel struct. Runs calculations to convert the RGB values into $Y/P_B/P_R$ values and stores them to compPixel struct. It is a CRE for either argument to be a null pointer.
comp_rgb_convert(void *rgbStruct, void *compPixelStruct); Takes a pointer to a compPixel struct with valid values, and a Pnm_rgb struct. Runs calculations to convert the $Y/P_B/P_R$ values into RGB values and stores them to rgbStruct.

It is a CRE for either argument to be a null pointer.

DiscreteCosine

DiscCos_luminToCoeff(void *r,)
Takes a struct that holds the four luminance values of the 4 in a 2 by 2 block. Calculates the corresponding a, b, c, and d values for those Y values.
It is a CRE for r to be a null pointer.

DiscCos_CoeffToLumin(void *r)
Takes a struct holding four float values representing the a, b, c, d values. Converts them into the four corresponding Y_1, Y_2, Y_3, Y_4 values.
It is a CRE for r to be a null pointer.

Bitpack

bool Bitpack_fitsu(uint64_t n, unsigned width);
Takes an unsigned integer and a width value. Returns a bool representing whether the integer could be represented in a number of bits equal to width.
It is a CRE for width to not be between or equal to 0 and 64.

bool Bitpack_fitss(int64_t n, unsigned width);
Takes a signed integer and a width. Returns a bool representing whether the integer could be represented in a number of bits equal to width.
It is a CRE for width to not be between or equal to 0 and 64.

uint64_t Bitpack_getu(uint64_t word, unsigned width, unsigned lsb);
Returns an unsigned value stored in a word, given the width of the word and the location of its least significant bit.
It is a CRE for width to not be between or equal to 0 and 64.

int64_t Bitpack_gets(uint64_t word, unsigned width, unsigned lsb);
Returns a signed value stored in a word, given the width of the word and the location of its least significant bit.
It is a CRE for width not be between or equal to 0 and 64.

uint64_t Bitpack_newu(uint64_t word, unsigned width, unsigned lsb, uint64_t value);
Returns a new word identical to the word passed in, but the field specified by width and least significant bit are replaced by the bits in value, an unsigned integer.
It is a CRE for value to not fit in the number of bits designated by width, or for width to not be between or equal to 0 and 64.

uint64_t Bitpack_news(uint64_t word, unsigned width, unsigned lsb, int64_t value);
Returns a new word identical to the word passed in, but the field specified by width and least significant bit are replaced by the bits in value, a signed integer.
It is a CRE for value to not fit in the number of bits designated by width, or for width to not be between or equal to 0 and 64.

Jose Lemus (jlemus01), Rex Umezuruike (rumezu01)
Homework #4
Design Document

floatToScaledInt

int5 floatToInt_new5B(float num) Takes a float value in the range -0.3 and 0.3 and returns a 5 bit, signed, scaled integer. Any number outside the range is rounded to -0.3 or 0.3, depending on its proximity to either.
uint9 floatToInt_newu9B(float num) Takes a positive float value between 0 and 1, and converts it to an unsigned, 9-bit, scaled integer. It is a CRE for the num to be outside the 0-1 range.

Data Types

- Pnm_ppm
- Pnm_rgb

Comp_img
A two-dimensional array that holds Comp-pixel structs. It is built upon an uarray2b implementation. It represents an image in component video color space.

Comp_pixel
A struct that holds three integers. It represents a pixel in component video color space, and stores the Y, P_B, and P_R components.

Lumin_block
A struct that holds the four luminance values of a block.

Compressed_block
A struct that holds a,b,c,d and values as well as the 4-bit quantized representations of the chroma values (average P_B and average P_R), for a two by two block.

Testing

Quantization of Chroma:
Functions: unsigned Arith40_index_of_chroma(float x) &
float Arith40_chroma_of_index(unsigned n)

- Write a test function to make sure that ‘unsigned Arith40_index_of_chroma(float x)’ will in fact return the closest chroma in the quantization table
- Important to note that the index_of_chroma function and the chroma_of_index function are practically inverses of each other (keep in mind that some data may be lost when going from chroma to index or vice-versa due to floating-point calculations). However, the relationship between these two functions

might prove to be useful when testing if we make sure to account for the potential small difference in output.

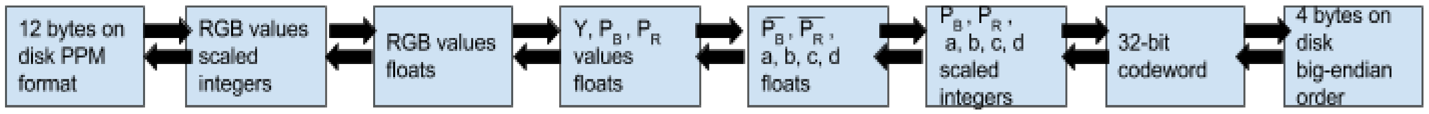
Bit packing:

- Bit-packing functions obey a ton of algebraic laws that should be discovered, coded, and checked against
- Test corner cases that are relevant to bit pack implementation such as testing of a width of 0 and a width of 64 as well as the testing of packing and unpacking bits (at the least significant and most significant bits)
- Test that a signed or unsigned integer can fit into a specified width before placing it into a word

Discrete Cosine Transformation:

- There are clearly constraints for the values of ***a***, ***b***, ***c***, ***d, Y,*** P_B, and P_R for the DCT. For a given image, we will have at least two different tests.
- (1) Check to make sure that the ***a***, ***b***, ***c***, and ***d*** values fall within their expected ranges. When |b|, |c|, and |d| are < 0.3 they should be coded as signed 5-bit scaled integers, but when those values are > 0.3 they should evaluate to either +0.3 or -0.3 (whichever is closer).
- (2) Also check to make sure that the **Y,** P_B, ***and*** P_R values fall within their expected ranges. The constant of proportionality restricts P_B and P_R values to the range of -0.5 to +0.5. Following the same logic, the **luminance Y** is a real number that should fall between the range 0 and 1.
- (3) Next, implement a more thorough check that can force the values of ***b***, ***c***, and ***d*** assuming that they have incorrect values (such as overly large magnitudes or floating-point inconsistencies that would lead to major coding errors).

Order of Writing our Code:


12 bytes on disk PPM format
RGB values scaled integers
RGB values floats
Y, P_B, P_R values floats
$\overline{P_B}$, $\overline{P_R}$, a, b, c, d floats
P_B, P_R, a, b, c, d scaled integers
32-bit codeword
4 bytes on disk big-endian order

Although not absolutely necessary, a critical part of our overall testing plan is the order in which we aim to write and test our functions. Compression goes from the beginning of this representation to the end, while decompression goes from the end of this representation back to the beginning. Rather than writing our functions in a linear order (from left to right) we plan to take advantage of the inverse properties of this assignment and test our code for one part with the part that immediately follows it. This will not only implement additional testing but it will add a layer of security that the program is being correctly implemented; a function and its inverse will always have a defined relation (universal law). As for actually implementing this logic, we plan to package some of these related parts into their own respective files (for modularity purposes and information hiding). These functions in separate files would be declared as extern in order to allow our overhead function/file to call them and test their output.

A specific example of this method would be the following steps:
(1) Creating a file that implemented a function to read in an image in PPM format which implies a two dimensional array of Pnm_rgb structs (struct size of 12 bytes).
(2) In the aforementioned file there will also be the implementation of another function that can scale the RGB integer values (via the denominator) to float values.

(3) Lastly, our overhead file (which links everything together) will include a function with the purpose of calling both of the external functions (which will read in the image and scale the RGB integers to RGB floats). This overhead file could then take those RGB float values and convert them back to RGB scaled integers (making function calls, of course) and then print the resulting image to stdout in order to make sure that the initial image read in just about matches the outputted image.

Overall Testing Plan:
Use a “test-as-you-build” sort of approach and create and run unit tests to make sure that we can identify any bugs in our arith program as they appear. Test often and incrementally so that we are confident we are not working with broken code. Minor bugs in some source files may not manifest until much later in the scope of our program, and since we will be using stepwise refinement as a problem-solving technique, this initial problem could prove to be much more troublesome as we progress. This potential issue would prove be *especially troublesome* in arith because in using stepwise refinement the problem is broken down into parts, subparts, and so on, until the individual sub-sub-parts solve the original problem. The fact of the matter is that almost every function is co-dependent on another function so we will have to test not only methodically but carefully.

Questions

How will your design enable you to do well on the challenge problem?
Each step of the program is compartmentalized and relatively simple to check for errors. It is possible to change some parts of the program without affecting areas related to calculations.

An image is compressed and then decompressed. Identify all the places where information could be lost. Then its compressed and decompressed again. Could more information be lost? How?
During the initial compression, some data can be lost when the image is trimmed so its height and width are even numbers. Additional data can be lost when the RGB values are scaled down with the denominator and turned into floats. The largest amount of data is lost when the numbers are turned into scaled integers. No additional data should be lost during the second compression and decompression, because the values should have already lost their initial precision during the first set of operations.