HW4 Design Document
Nadine Shen Molesky, Taylor Ampatiellos
Date: March 1st, 2016

Part A: 40image

Overall design/architecture:

Overview of program/Interactions between Components:

Due to our use of stepwise refinement in solving the given problem, our program has a structure in which we have one small main function which calls to sub-functions to carry out the programs, each of which call to smaller sub-functions, and so on until we have all of the necessary sub-sub-functions needed to complete the program. In other words, the program is composed of a multitude of very small functions with specific jobs, which are called by larger functions and complete the given task.

Our program will first read in the command line and call the corresponding compress40.h function. Here is an outline of the compression program once compress40() is called:

- Read in image with pnm.h functions
- Trim rows and columns if necessary
- call int_to_float() from code_value.h file, for each pixel
- call RGB_to_component() from RGB_component.h file, for each pixel
- call block_to_word() from block_word.h, for each 2x2 block
    - takes average Pb and Pr values
    - calls Arith40_index_of_chroma()
    - calls pixel_to_cosinespace() from cosinespace_pixel.h
    - calls float_to_5bit_sint() and float_to_9bit_uint() from code_value.h
    - calls word_pack() from word_pack_unpack.h
        - uses Bitpack functions
- writes compressed image to stdout using a simple print function

Here is an outline of the decompression program once decompress40() is called:

- Read in header of compressed file
- Allocates an empty pixmap of correct dimensions using pnm.h
- Calculates the denominator of the image
- For each codeword, call word_to_block() from block_word.h, which creates a 2x2 block
    - calls word_unpack() from word_pack_unpack.h
        - uses Bitpack functions
    - call Arith40_chroma_of_index()
    - calls cosinespace_to_pixel() from cosinespace_pixel.h
- call component_to_RGB() from RGB_component.h file, for each pixel in each block
- call float_to_8bit_uint() from code_value.h to quantize each pixel
- write image to stdout using pnm.h function

Overview of Components:

We have packaged together most of the functions into five main components (not including the provided ones), which each perform the corresponding compression and decompression functions for a single step in the process. Each component also has its own printing and testing functions to test it as one unit. Block_word.h contains functions to call the functions to perform step (4) in the specs for both compression and decompression, and converts a codeword to a block and vice versa. RGB_component.h contains functions to convert RGB values to component values, and vice versa. cosinespace_pixel.h functions convert the cosine space values to pixel values and vice versa. code_value.h packages all of the functions that have to code one value type to another value type. It contains functions to convert the cosine space values to either 5-bit or 9-bit integer values during compression, and convert between a float and a normal 8-bit unsigned integer during compression and decompression.

Components, their architecture, and descriptions ([ ] indicate pseudocode)

- compress40.h
    - Contains overall compress and decompress functions (high-level)
    - void compress40(FILE *input)
        - read in image with pnm.h function
        - trim rows and columns if necessary
        - call necessary compression functions described below
        - write compressed binary image to stdout, use function:
            - void print_compressed(uint64_t word, width, height)
    - void decompress40(FILE *input)
        - read header of compressed file with function:
            - void read_header()
        - allocates array of pixels with pnm.h
        - calculate denominator with function:
            - int den calc_denominator(pixmap)
        - call necessary decompression functions described below
        - print image with pnm.h write function
- arith40.h
    - (provided header file) - contains functions to convert between chroma and indices of quantization array
- Block_word.h
    - uint64_t word block_to_word(struct block) - performs all of step (4) in the spec:
        - struct pbpr average(block);
            - computes average of the four Pb and Pr values
        - calls Arith40_index_of_chroma() function
        - calls the other necessary functions for step (4) described below
    - struct block word_to_block(uint64_t word) - performs all of step (4) for decompression, in the same way
- RGB_component.h
    - struct comp RGB_to_component(R, G, B)

        - performs the matrix multiplication for one block
        - calls multiply_matrix()
            - returns component values
            - takes in RGB values
    - struct RGB component_to_RGB(Y, Pb, Pr)
        - performs the matrix multiplication for one block
        - calls multiply_matrix()
            - returns RGB values
            - takes in component values
- cosinespace_pixel.h
    - struct pixels cosinespace_to_pixel(struct cosines)
        - transforms cosine coefficients a, b, c, d to four Y values
    - struct cosines pixel_to_cosinespace(struct pixels)
        - transforms four Y values to cosine coefficients a, b, c, d
- code_value.h
    - int float_to_5bit_sint(float number)
        - if value is more than 0.3, converts to +-0.3
        - converts to 5-bit signed int
    - int float_to_9bit_uint(float number)
        - quantizes float to unsigned int between 0 and 511
    - static inline int float_to_8bit_uint(float number)
        - quantizes float to unsigned int between 0 and 255
    - float int_to_float(int number)
        - changes int to float
- word_pack_unpack.h
    - struct values word_unpack(uint64t word)
        - unpacks one code word and stores a, b, c, d, Pb, and Pr into a struct, using the bitpack field extraction functions.
    - uint64t word word_pack (struct values)
        - packs a, b, c, d, Pb, and Pr into the code word
- Unpackaged functions
    - map_blocks(array, apply, *cl)
        - maps through each 2-by-2 block so that each function can operate on just one block

Testing

Overview

The fact that the structure of our program runs in two directions along the same path (compressed to decompressed and vice versa) allows us to utilize a unique and straightforward testing method. We will begin by testing the first first function of the compression portion of the program, which is to read in an image in PPM format and trim the edges. We will test this by

simply having the program read in the image, then output what has been read in. Once we certify this works, we will move on to the next step, which is to change the RGB values of the image into component video space. Once this step is also confirmed to work as it should, rather than move directly on to step three, we will instead try to work backwards and convert this format back to RGM and output the result (which is the last step of decompression).

This pattern will continue as we extend our program, always testing the reverse compatibility of our functions by compressing an image further and then decompressing it. By the time we do this for the last step of compression, we will also have a fully working decompression function!

All of the steps in the compression either work on single pixels or blocks at a time. Each time we implement one function, we will first test it on the single pixel or block (by inputting values), and then test it with a whole image by making it the apply function of a mapping function.

Tests for each component

For every function/component, after testing that it individually works, we will test it in the context of the entire compression (up to that step) and then a decompression, and view the resulting image.

- Block_word.h
    - Since this component contains multiple sub-steps in it, we will test each step separately.
    - We will test the block_to_word function by inputting (Y, pb, pr) values and printing the codeword to the screen, and vice versa for the word_to_block function.
- RGB_component.h
    - First we will ensure that our multiply_matrix() function works by entering many different matrices and comparing the results with an online matrix calculator. We will then make sure the RGB_to_component() and component_to_RGB() functions work by inputting RGB or component values and printing the result, and comparing it with the results from a matrix calculation done with a calculator. We will make sure to test corner or error cases, such as inputting all 0's or negative numbers.
- cosinespace_pixel.h
    - We will test that the functions to convert a Y value to 4 cosinespace values, and vice versa, work by manually entering the Y value and printing the cosinespace values, and vice versa. After making sure it works on a pixel, we will call it on a whole block.
- code_value.h
    - We will test each of these functions by entering a variety of floats/ints, printing the results, and making sure that the they are coding the numbers correctly by comparing it to a result calculated by hand.
- word_pack_unpack.h

- We will test this function by packing a, b, c, d, Pb, and Pr values into the code word, then unpacking said word and see if we get the result for each variable that we should. We will also test for error handling by passing invalid values (i.e. a negative value for a).

Other Tests

We will test that the compressed image is about 3 times smaller than the decompressed image by outputting them into files and looking at the size of the files.

We will use our ppmdiff program created in lab to compare the original image and the image generated after compressing and decompressing. The errors between the images should be in the 0.1% to 2.5% range, if not slightly higher for a lossy image.

Conversion from RGB to component video should be inverse functions; check both directions by nesting them and checking if the output is the same as the input. This same idea also goes for the cosine transform and most of the other steps.

Check for approximate equality of inverse properties using the function ((x-y)^2)/((x^2) + (y^2)). We will do this by using the RGB, component video, and cosine transform functions to shift values to and from their original forms, then comparing them to the originals.

Part B: bitpack

Additional Components

shift.h -

This header file will contain functions to make it easier to implement hardware shift instructions. These functions will make sure that something sensible is done if the user asks to shift something 64 bits, take care of the 2 different right shift types, and make sure the hardware doesn't operate on 32-bit values when we want to operate on 64.

Functions in shift.h:

- static inline shift_right(value, shift_amount)
  - If shift_amount = 64, call shift_64(value)
  - Else, shift right the specified amount
- static inline shift_left(value, shift_amount)
  - If shift_amount = 64, call shift_64(value)
  - Else, shift left the specified amount
- shift_64(value)
  - Changes the entire field to 0's

Testing the Functions

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

- We will test both of these functions by inputting a variety of n's (the integer in question) and widths, and printing the result, and comparing that result with the result that we calculate on our own.

uint64_t Bitpack_getu(uint64_t word, unsigned width, unsigned lsb);
int64_t Bitpack_gets(uint64_t word, unsigned width, unsigned lsb);

- We will test both of these functions by inputting various arguments and making sure they come out with the correct integer from the binary representation within the word. We will also test for the error cases, by calling a width less than 0 or over 64, and a width or lsb that does not satisfy w + lsb <= 64.
- For the new and get functions, we will test the following invariants from the specs:
    - Bitpack_getu(Bitpack_newu(word, w, lsb, val), w, lsb) == val;
    - if lsb2 >= w + lsb,
      getu(newu(word, w, lsb, val), w2, lsb2) == getu(word, w2, lsb2);
      (and similar cases for the signed get functions).

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);

- We will test these functions by inputting various arguments and making sure they come out the correct new word with the correct value inserted into the correct position. We will test the error cases by calling a width less than 0 or over 64, and a width or lsb that does not satisfy w + lsb <= 64. We will also test for error messages if the value does not fit width bits.

We will also test the functions using all 64 bits, and using no bits.

shift.h

- To test the functions in shift.h, we will input a variety of values and shift_amounts and verify that the output is correct. We will make sure to test shifts of length 64.

Questions

- How will your design enable you to do well on the challenge problem in Section 2.3 on page 13?
    - In our word_pack() and word_unpack() functions, we will have global variables for the format of the word - such as the length, the locations where new values are stored along the length of the word, and the width of each value. Therefore, if we need to make changes to the word format, we only will need to change those global variables. Since these are stored in their own file called word_pack_unpack, we would only have to make changes in one file.
    - In our code_value.h component, we have functions to change floats to 5-bit, 9-bit, and 8-bit ints, but the new codeword could require ints of a different number of bits. If we have time, we will write an additional, more modular function in code_value.h that allows the user to input the number of bits that they want to convert their float into.

- An image is compressed and then decompressed. Identify all the places where information could be lost. Then it's compressed and decompressed again. Could more information be lost? How?
    1. During compression, we trim the rows and columns to be even numbers. Therefore, edge information would be lost in an image without an even number of rows and columns if an image was compressed and then decompressed again. Information is only lost once in this step.
    2. In step 3, the RGB values are changed from an unsigned int representation into a floating point representation. This does not cause loss of information during compression, but during decompression, the quantization of floats into ints will cause a significant loss of information.
    3. In step 4a, when the Pb and Pr block averages are calculated, some information is lost, because when you decompress, there is no way to get the original values from the average. Information is only lost once in this step.
    4. In step 4b, the Pb and Pr block averages are quantized into a number from a specific set of numbers. This will cause a significant loss of information, especially in images that use saturated colors (when Pb and Pr are high).
    5. In step 4d, we convert cosine coefficients b, c, and d into five-bit signed values. This will cause some loss of information, especially if the cosine coefficients are outside of the +- 0.3 range (in which case we code them as + or - 0.3). Converting the coefficient a into 9-bit unsigned value will cause less loss but still may cause loss of information.
    6. More information could be lost with multiple compressions and decompressions because of the inexactitude of floating point arithmetic, from the multiple conversions between float to int, and the quantizations. These effects all together would cause repeated losses with repeated compressions and decompressions.