COMP 40 Homework #3 Design Document

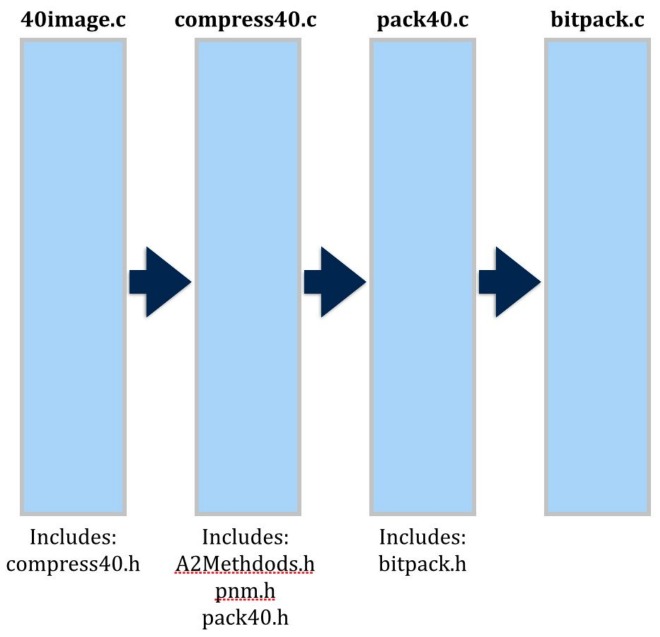
by Theodore Tan (ttan02) and Bill Yung (byung01)

14th October 2015

1. Overall Architecture

Main Problem: This program should take in a single image file (either on the command line or from stdin) and either a) compress the image to a smaller one about 3 times smaller, or b) decompress a small image to its original size.

Architecture of Program: The program is divided into 4 main modules as outlined in the diagram below.



40image(Client): Opens a file and passes it to the compress40 interface. Based on the user's specifications, compress40 will either run compress or decompress.

compress40: Reads and writes the image in either compressed or decompressed form based on which function is called.

pack40: Packs a 2 x 2 block into a 32-bit word, or unpacks a word into a 2 x 2 block. This interface will hide the details of bitpack from compress40.

Bitpack: Does the actual packing into the 32-bit word, based on a, b, c, d, average of Pb, and average of Pr. Also unpacks the bit word to obtain the variables a, b, c, d, average of Pb, and average of Pr.

2. **Detailed Architecture (40image)**

Interfaces Included	Description
compress40.h	Holds the compress and decompress functions.
Functions	**Description**
int main	Opens a ppm image file from the command line or stdin and passes it to compress_or_decompress.
void compress_or_decompress	Pointer to a function to either compress or decompress the image that has been read in. This pointer is set in main according to user specifications.

3. **Detailed Architecture (compress40)**

Interfaces Included	Description
A2methods.h	Interface for the ADT used to read in image.
pnm.h	Interface to read in images in ppm format.
pack40.h	Holds the pack or unpack functions to convert from 2 x 2 blocks to 32-words and vice versa.
Functions	**Description**
void compress40	Takes in a pointer to a ppm file, opens it, and compresses it.
void decompress40	Takes in a pointer to a ppm file, opens it, and decompresses it.
Helper Funcs (compress40)	**Description**
Pnm_ppm read_image	Reads in an images and returns a struct containing an image (2D UArray of pixels) and its dimensions.
void trim_image	Trim image to ensure width and height are both even numbers. If already even, do nothing.
void convert_CV	Converts from scaled unsigned integers to floating-point representation. Convert RGB values to component video color space (Y/Pb/Pr). Stores the color space values in new blocked array of structs, as defined:

	struct CV_colorspace { float Y; float Pb; float Pr; };
UArray_T compressor	Contains the loops to map through the blocked array and gets the 2 x 2 blocks to pass to pack. It then calls pack and stores the 64-bit code word in an UArray of 64-bit words. Returns the UArray_T of 64-bit words.
uint64_t pack	Takes in a 1D UArray of struct CV_colorspace (2 x 2 pixel block), packs it and returns a 64-bit word.
void write_compressed	Print the header of a compressed binary image followed by a sequence of 32-bit code words. Uses the UArray that compressor returns. Uses putchar() to print out the 32-bit words byte by byte.
Helper Funcs (decompress40)	Description
Pnm_ppm create_image	Reads the header of the compressed binary image. Allocates a new 2D blocked array of width and height as specified, size of the colored pixel, and a denominator of choice.
UArray2_T decompressor	Malloc's Array of struct CV_colorspace Contains the loops to read in 32-bit words byte by byte using getchar(). After reading a 32-bit word, it passes the word to unpack(). Stores the 1D UArrays that unpack returns into the array within the the UArray of struct CV_colorspace
UArray_T unpack	Takes in a 64-bit word, unpacks it and returns a 1D UArray of struct CV_colorspace (2 x 2 pixel block) as defined above.
void convert_RGB	Takes in a 2D array of struct CV_colorspace as defined above. Converts from local variables to RGB values. Stores the color space values in the Pnm_ppm created in create_image().
void write_decompressed	Takes in a Pnm_ppm and writes it out using Pnm_ppmwrite()

4. Detailed Architecture (pack40/unpack40)

Interfaces Included	Description
bitpack.h	Interface to use bitpacking functions to convert a, b, c, d, Pb, and Pr values into a 32-bit codeword.
Functions	Description
uint64_t pack	Takes in a 1D UArray of struct CV_colorspace (2 x 2 pixel block), packs it and returns a 64-bit word.
UArray_T unpack	Takes in a 64-bit word, unpacks it and returns a 1D UArray of struct CV_colorspace (2 x 2 pixel block).
Helper Funcs (pack40)	Description

struct bitWord initialize_bitWord	Calls set_chroma_index() and cosine_transform() to set a, b, c, d, Pba, and Pra.
	
	`struct bitWord {`
	`        unsigned    a;`
	`        signed      b, c, d;`
	`        unsigned    Pba(index), Pra(index);`
	`};`

void set_chroma_index	Converts Pba, Pra from floats to scaled integers. Takes in an UArray of four pixels. Calculates the average value of the chroma elements of the four pixels. It then converts and puts a 4-bit quantized representation of the chroma value into our struct. It gets called twice for Pb and Pr.
void cosine_transform	Converts a, b, c, d from floats to scaled integers. Takes in an UArray of four pixels and transforms the four Y values of the pixels into cosine coefficients a, b, c, and d. Converts b, c, d to five-bit signed values assuming they lie between -0.3 and +0.3.
uint64_t pack_word	Takes in a bitWord struct and packs the a, b, c, d, Pba(index), Pbr(index) into a 32-bit code word within a 64-bit word using the bitpack.h interface.
Helper Funcs (unpack40)	Description

struct bitWord unpack_word	Takes in 64-bit word and extracts a, b, c, d, Pba(index) and Pra(index). Initializes the bitWord struct with these values. Uses the bitpack.h interface to extract these values.
UArray_T initialize_colorspace	Takes in a struct bitWord and calls reverse_cosine_transform() and get_chroma() to initialize the CV_colorspace of each pixel in the 2 x 2 block. Returns a 1D UArray of struct CV_colorspace.
void reverse_cosine_transform	Converts a, b, c, d from scaled integers to floats. Transforms a, b, c, d into four Y values and stores them separately in UArray2_T of CV_colorspace structs.
void get_chroma	Converts Pba, Pra from scaled integers to floats. Takes in a 4 -bit quantized representation of a chroma value. It then converts and returns the average value of the chroma elements of the four pixels.

5. **Detailed Architecture (Bitpack)**

Functions	Description
bool Bitpack_fitsu **bool** Bitpack_fitss	**Width-test functions**: This will test if an integer can be represented in k bits. Returns true if it fits, false otherwise.
uint64_t Bitpack_getu **int64_t** Bitpack_gets	**Field-extraction functions**: This extracts values of a specified width from a word, starting from a specified lsb. Fields of width zero are defined to contain the value zero. **Checked Runtime Errors**: - The width **w** that is passed into the function must satisfy **0 <= w <= 64** - **w + lsb <= 64**
uint64_t Bitpack_newu **uint64_t** Bitpack_news	**Field-update functions**: Updates the a field within the word at the lsb with a given value of a specific width. Returns a new word identical to the original with updates if specified. All updates will be done on the copy of the original word, so that the original word is not mutated. **Checked Runtime Errors:** - The width **w** that is passed into the function must satisfy **0 <= w <= 64** - **w + lsb <= 64** - If the function is passed a value that does not fit in the specified width, it must raise the exception

	Bitpack_Overflow

Other considerations:
- **Shift** values are for <= 64 bits
- **Width** values are for <= 64 bits
- Bit fields to be accessed or updated fit entirely within the 64-bit word

6. **Testing Plan (compress40/decompress40)**
    - We designed compress40 and decompress40 so that the both have the same number of steps and each step of compress40 is the reverse of the corresponding step of decompress40, which is why we can short circuit the testing process. So we can build and test one step at a time for each function before building the next step.
    - We'll create a testing program that calls compress40 and the output will be passed to decompress40 to see if the end output is the same as the original input
    - **Specific Test Cases:**
        - Reading an image in compress40 and writing it out in decompress40
        - Compress40 will convert RGB values from scaled integers to floating point numbers and decompress40 will convert them from floating point to scaled integers.
        - Compress40 will convert RGB float values of an image to Component Video colorspace and decompress40 will reverse the conversion.
        - Compress40 will call pack40 and decompress40 will call unpack40 assuming pack40/unpack40 have been fully tested and functional.

7. **Testing Plan (pack40/unpack40)**
    - Because we have a separate interface from compress40.c to hide the details of the bitpack.h interface, we are able to test pack40 and unpack40 completely separately from the compress40 and decompress40 functions.
    - We'll create a testing program to compare the input given to pack40 and output from unpack40
    - **Specific Test Cases**:
        - Create an UArray of struct CV_colorspace, initialize values the elements within struct bitWord, print them out to check the output.
        - Create an UArray of struct CV_colorspace, pass it through pack, then pass the compressed binary image to unpack and compare the output of unpack with the original values of the struct
        - Do the discrete cosine transformation and other calculations on paper to make sure the values are correct.

8. **Testing Plan (Bitpack)**

- We'll create a testing program that does two key things: populates 64-bit words and prints them out.
- **Specific Test Cases**:
    - Bitpack_fitsu(), Bitpack_fitss()
        - Insert valid widths and make sure it returns true
        - Insert invalid widths and make sure it returns false
    - Bitpack_getu(), Bitpackgets()
        - Populate an entire 64-bit word with known values and call get to check if correct values are being stored.
    - Bitpack_newu(), Bitpack_gets()
        - Insert values into 64-bit words and print them out to check if the information is stored correctly e.g. stored at correct lsb
    - Checked Runtime Errors
        - Insert values that are bigger than 64-bits
        - Shift values must be <= 64 bits
        - Use a width **w** that does not satisfy **0 <= w <= 64**
        - Use a **w** and **lsb** that does not satisfy **w + lsb <= 64**
        - If the function is passed a value that does not fit in the specified width, it must raise the exception **Bitpack_Overflow**

9. **How will your design enable you to do well on the challenge problem in Section 2.3 on page 13?**

For compression, all that needs to be changed is the **pack_word** function. The first step in our compression process is deciding the order by which to put the values into the codeword. Once the order is decided, we will be able to loop through the variables and put them in the codeword in predetermined order. Therefore, the only thing that needs to be changed is the way that the variables are queued up to be put into the codeword..

For decompression, all that needs to be changed is the **unpack_word** function. The first step in our decompression process is reading a, b, c, d, Pba, and Pra into the bitWord struct. Once read, the rest of the program works in exactly the same way. Therefore, the only thing that needs to be changed is the way that we locate each local variable from the codeword.

10. **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?**

When an image is compressed, the average values of Pb and Pa of a 2 x 2 block of pixels is calculated. The average is then stored to represent the Pb and Pa for all four pixels combined in the compressed version. After the compressed binary image is decompressed, information is lost because Pb and Pa are averages and not the actually Pb and Pa of the original pixels. Then each pixel of the 2 x 2 block will store the averages as their own Pb and Pa. Because the previous averages are stored, when the image is compressed again, the average of Pb and Pa will be the same as before. So when decompressed, no extra information is lost since the value of Pb and Pa never changed.