Arith Design Document

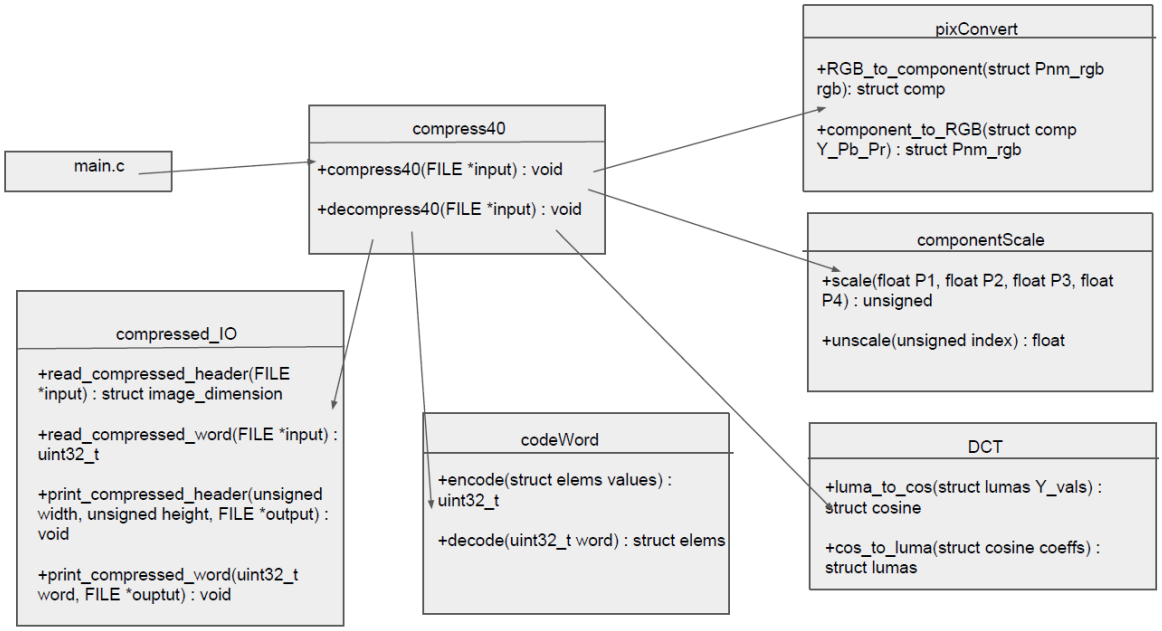
COMP 40 : Machine Structure and Assembly Programming Language

by George Pesmazoglou

and Alexandru Ungureanu

Prof. Noah Mendelsohn

1. Overall Architecture



As can be observed in our diagram, we will implement 5 extra interfaces for this project. The main function has access to the compress40 interface, which is the "controller" interface of the program. In order to perform the compression/decompression algorithms, the compress40 implementation has access to the pixConvert, componentScale, DCT, codeWord, and compressed_IO interfaces.

Note – other interfaces we will use but are not included in the diagram:

- Bitpack – our codeWord module will have access to the Bitpack module we will implement in part B; Bitpack does not know anything about codeWord, and codeWord does not know how word "parsing", or replacing part of the word work
- pnm – used for pnm images input and output, only compress40 has access to it

2. Architecture of the components

40image module

- Contains the main function of the program
- The main function is used to read and process command line arguments
- The given file is the image to be compressed/decompressed. If no file is given, the image is taken from standard input
- There are 2 possible command line options for the program: -c (given image should be compressed) or –d (given image should be decompressed)

- Calls the public compress40 or decompress40 public functions in the compress40 interface based on the given command line option

compress40 module

- "controller" interface of the program, used to perform the main logic of the image transformation
- Calls each of the 5 added interfaces to perform the logic, so it has access to the steps of the algorithm, but doesn't have access to their implementation specifics
- Function **void compress40(FILE *input)** – takes the input file as argument and compresses the image, printing the compressed image
- Function **void decompress40(FILE *input)** – takes the input file as argument and decompresses the image, printing the decompressed image

pixConvert module

- Interface responsible for pixel conversions between the RGB color space and the component video color space
- Function **struct comp RGB_to_component(struct Pnm_rgb rgb)** – takes a structure containing the RGB values of the pixels (0 to 255), changes to floating-point representation and converts the pixel into component video color space; returns a structure containing the $Y/P_B/P_R$ components of the pixel
- Function **struct Pnm_rgb component_to_RGB(struct comp Y_Pb_Pr)** – takes a structure containing the $Y/P_B/P_R$ components of the pixel and converts the pixel to RGB color space representation; returns a structure containing the red-green-blue values of the pixels (0 to 255)
- Type **struct Pnm_rgb** – used to represent a pixel in terms of red, green and blue intensity. Fields: red, green, blue (unsigned integers) – can be any number from 0 to 255
- Type **struct comp** – used to represent a pixel in terms of the component video color space elements. Fields: Y (float) – luminance value from 0 to 1.0; Pb, Pr (float) – chroma elements from -0.5 to 0.5

DCT module

- Interface responsible for transforming between luma values and *scaled* cosine coefficients
- Function **struct cosine luma_to_cos(struct lumas Y_vals)** – transforms a structure of 4 luma values of 4 different pixels into cosine coefficients *a*, *b*, *c*, *d*; the cosine coefficients are then scaled as following: *a* to a 9 bits unsigned integer, and *b*, *c*, *d* to 5 bits signed scaled integers; returns a structure containing the 4 scaled cosine coefficients
- Function **struct lumas cos_to_luma(struct cosine cos_coeffs)** – transforms a structure of 4 scaled cosine coefficients to four luma values; returns a structure containing the four luma values for the pixels

- Type **struct cosine** – used to represent the four scaled cosine coefficients of a pixel. Fields: a (unsigned 9 bits integer), b, c, d (signed 5 bits integers) that represent the scaled cosine coefficients
- Type **struct lumas** – used to represent four luma values of four different pixels. Fields: Y1, Y2, Y3, Y4 (floats) – the 4 luma values of the 4 pixels

componentScale module

- Interface responsible for converting between chroma element values and *quantized* chroma values
- Function **unsigned scale(float P1, float P2, float P3, float P4)** – computes the average value of 4 chroma values and scales it to a 4 bits unsigned integer; returns the scaled unsigned value
- Function **float unscale(unsigned index)** – computes the chroma value associated to the scaled chroma values index given as the parameter; returns the chroma value – floating-point number from -0.5 to 0.5

codeWord module

- Interface responsible for encoding from the scaled components of 4 pixels into a 32-bit word and decoding from a 32-bit word to the scaled components of the pixels
- Function **uint32_t encode(struct elems pixel_elems)** – takes a structure containing the a, b, c, d, P_B and P_R scaled elements of the four pixels and returns the word obtained by packing the elements in big-endian order
- Function **struct elems decode(uint32_t word)** – takes a word stored in big-endian order and returns a structure containing the unpacked a, b, c, d, P_B and P_R scaled elements of the four decompressed pixels
- Type **struct elems** – used to represent the scaled elements of four pixels; Fields – struct cosine (explained earlier), Pb_avg, Pr_avg (4 bits unsigned integers) – the quantized representations of the average chroma values of the four pixels
- a is between 0 and 1; because greater than 0.3 values of b, c, d are very rare, we will take any b, c, d values that are outside the [-0.3, 0.3] interval to be -0.3 or 0.3 (whichever is closer)

compressed_IO module

- Interface responsible for inputting and outputting compressed images
- Function **struct image_dimensions read_compressed_header(FILE *input)** – reads a compressed header that has the format specified in the spec. Returns a structure with the dimensions of the image (read from the header)
- Function **uint32_t read_compressed_word(FILE *input)** – reads the next unread word in the compress file and returns it. Checked runtime error if no unread word left
- Function **void print_compressed_header(unsigned width, unsigned height, FILE *output)** – prints a header in the format specified in the spec to the given file
- Function **void print_compressed_word(uint32_t word, FILE *output)** – writes the given word in the given file
- Type **struct image_dimension** – structure containing the dimensions of the image (for instance, width)

Bitpack module

- This file contains the implementation of functions provided in the bitpack.h file, given in this assignment.

- Function **bool Bitpack_fitsu(uint64_t n, unsigned width)** - checks if an unsigned integer can be represented in the bit field. Parameters are the integer to be represented and an integer that corresponds to the width of the bit field. It returns a boolean value that corresponds to true or false
- Function **bool Bitpack_fitss(int64_t n, unsigned width)** - same as above, but for signed integers
- Function **uint64_t Bitpack_getu(uint64_t word, unsigned width, unsigned lsb)** - extracts an unsigned integer from a bit field. Parameters are the 64-bit bit field (unsigned integer), the width of the integer to be extracted (unsigned integer) and the location of the least significant bit. Returns an unsigned integer – the extracted integer.
- Function **int64_t Bitpack_gets(uint64_t word, unsigned width, unsigned lsb)** - similar function to the one above, but extracts and returns a signed integer.
- Function **uint64_t Bitpack_newu(uint64_t word, unsigned width, unsigned lsb, uint 64_t value) -** updates a word with a new word where only a few bits have been changed to represent a specific integer. Parameters are: the word, width of the new value, location of least significant bit and the actual value (unsigned integers). Returns a new 64 bits word (unsigned integer)
- Function **uint64_t Bitpack_news((uint64_t word, unsigned width, unsigned lsb, int64_t value)** – similar to the one above, but the value to replace with is a signed 64 bits integer

3. Testing

To test if the compression/decompression program works correctly, we will do unit testing for each module we include in our project. After each module is tested separately and works correctly, we will test the functionality of the program itself. In order to test the **pixConvert**, **componentScale**, **DCT** and **codeWord** modules, we will write unit tests that take in some inputs, perform one function on the inputs, and then perform the inverse function on the outputs of the first function. Since the functions are the inverse of one another, the final outputs should match the initial inputs. We understand that precision will be lost, for instance when working with floating-point values, so we will test if the final outputs' values are *approximately* equal to the original inputs, that is if they fall in a small range of values associated with each input.

Bitpack tests

- Each function should be tested separately.
- There will be a main function that calls each function with given parameters, and the output will be tested.
- For example we will test Bitpack_fitsu for different sizes of bit fields and widths, widths larger than the size of the bitfield. We will also check how the program handles wrong input, like passing a signed integer instead of a signed one.

pixConvert tests

- Take 3 RGB values as inputs, call RGB_to_component to get the floating-point $Y/P_B/P_R$ values, and then call component_to_RGB for the component video color space values obtained earlier to get the RGB unsigned integer values. Those values should match the initial RGB input **exactly**. While some precision is lost when scaling a floating-point value, the closest integer value to the scaled result should still be the initial RGB value. The margin of error when scaling exists, but is not large enough to affect the result by more than 0.5.

- Take 3 $Y/P_B/P_R$ values, and call component_to_RGB and then RGB_to_component on the outputs. The final outputs will not match $Y/P_B/P_R$ **exactly**, so we will test if they fall in a small range of values close to the original values (margins of error).
- Test for the "extreme" RGB values: 0 and 255 and "extreme" $Y/P_B/P_R$ values: 0 to 1 for Y, -0.5 and 0.5 for the chroma values
- Test to see if either of the function's output goes out of the bounds of the RGB or $Y/P_B/P_R$ values (they should not)
- Check for bigger input values, where more precision should be lost
- Check for wrong input (should throw an error)

componentScale tests

- Take 4 chroma values as inputs, call scale for those values. Then take the output of the function and call unscale for it. Check if the final output is **approximately** equal to the average of the original values, by testing if it falls into a "margin of error" range of values.
- Take a 4 bits unsigned integer, call unscale and get a chroma value. Call scale for 4 variables with the same chroma value (the one outputted by unscale), and see if the 4 bits unsigned integer final output is **exactly** equal to the original one. Again, some information is lost because floating-point computations are not exact, but the closest integer to the scaled float value should still be the original integer (the margin of error cannot be bigger than +- 0.5)
- Test for "extreme" values: -0.5 and 0.5 for the chromas, 0 and 15 for the scaled unsigned integer
- Test to see if either of the function's output goes out of the bounds of the 4 bits integers or $Y/P_B/P_R$ values (they should not)
- Check for bigger input values, where more precision should be lost
- Check for wrong input (should throw an error)

DCT tests

- Take four luma values as inputs, call luma_to_cos, and then cos_to_luma for the outputs of the previous function. The final results will not match the original input values **exactly**, but should fall within a small "margin of error" interval.
- Take cosine coefficients as inputs, call cos_to_luma and then luma_to_cos for the outputs of the previous function. The final outputs will not match the original floating-poin coefficients **exactly**, but should again be within a small margin of error
- Test for "extreme" values: 0 to 1 for the lumas, -0.3 to 0.3 for b, c, d and 0 to 1 for a
- Test to see if either of the function's output goes out of the bounds mentioned previously
- Check for bigger input values, where more precision should be lost
- Check for wrong input (should throw an error)

codeWord tests

- Take six scaled component video elements, pack the word and then use the inverse function to unpack. Here we are not modifying any of the values (just composing or "parsing" a word), so the initial inputs and final outputs should match **exactly**
- Take a word, use decode to get the elements and then encode on these elements to pack a word. The initial and final word should match **exactly** again, since no information is lost
- Test for wrong inputs (more bits then specified). Result should be a checked runtime error

Compressed_IO tests

- Read a header and print it immediately after. They should match perfectly.
- Read a word and print it immediately after. They should match perfectly.

General tests for the whole program (after unit testing is complete)

- Test for images of different sizes
- For small images, see if the output matches our expected output (few computations, can do on paper)
- For all images, try compressing and decompressing back. Results should be similar, but in most of the times will not match **exactly**. We will explain why in the 5th headline.
- For all compressed images, try decompressing and compressing back. Result should be **exactly** the same. We will explain why in the 5th headline.
- Test if the given command line option is performed correspondingly
- Valgrind tests for memory errors/leaks

4. Challenge problem

Since our design is modular and has a different module for each compression/decompression step, we believe we will do well on the challenge problem. The 5 extra interfaces are independent for each other (they perform different tasks), so changing one of them will not require us to change the others too. Moreover, the **compress40** module doesn't know anything about the implementation of the 5 extra modules. The 5 extra modules do not know **compress40** exists. Thus, a change in a module will not require changes in the modules that are higher or lower in the modules hierarchy.
To add to this, each of our modules will have several independent private methods that perform the steps in the module's work. Thus, changing one step will require changing only one of the private methods, and not the others or the public methods.

5. Lost information

When an image is compressed and decompressed, information is lost because:

- Compressing is done by taking the average chroma values for each 2x2 square. When decompressing, the chroma values of the 2x2 square they originally came from will now have equal chroma values (equal to the average of the initial values). Thus, the resulting decompressed image will imitate the original one, but some information is lost.
  However, if we compress and decompress again, since each 2x2 square will have pixels with the same chroma values, this information will not be lost anymore.
- When performing floating-point computation, some information is lost in precision (in DCT for instance, where the function-inverse function combination never outputs the **exact** original input). If we compress and decompress again, some information will again be lost in floating-point computations.
- A very small quantity of information is lost if the original image has odd width and/or height, since the image is trimmed. When compressing and decompressing again, the image will now have the trimmed sizes, so this information is not lost anymore.