Homework 4 Design Document Arith

| Jeremy Batchelder and Daniel Cetlin | 10/13/17 |

===

1) **What are the major components of your program, and what are their interfaces? Components include functions as well as abstract data types. An interface includes contracts as well as function prototypes.**

- **rgb_vcs_converter.h**
  - `void rgb_to_vcs(Pnm_ppm pmap);`
    - Converts the given Pnm_ppm's uarray2 of Pnm_rgb pixel structs to a blocked uarray2b of Vcs component video structs. It is a checked runtime error for pmap to be NULL.
  - `void vcs_to_rgb(Pnm_ppm pmap);`
    - Converts the given Pnm_ppm's blocked uarray2b of Vcs component video structs to a regular uarray2 of Pnm_rgb pixel structs. It is a checked runtime error for pmap to be NULL.

  -
    ```
    void apply_rgb_to_vcs(int i, int j, A2 array2,
                          A2Methods_Object *ptr, void *cl);
    ```
    - Used in rgb_to_vcs when iterating through the uarray2 of pixels.
    - The apply function converts the Pnm_ppm pixel value ptr into a Vcs struct and places the Vcs struct at position i and j in the uarray2b contained within in the closure.

  -
    ```
    void apply_vcs_to_rgb(int i, int j, A2 array2,
                          A2Methods_Object *ptr, void *cl);
    ```
    - Used in vcs_to_rgb when iterating through the uarray2b of Vcs structs.
    - The apply function converts the Vcs struct value in ptr into a Pnm_rgb pixel value and places the Pnm_rgb struct at position i and j in the uarray2 contained within in the closure.
    - The pixel will be set in the range 0-255.

- **compress40.h**
  - `void trim(Pnm_ppm pmap);`

        - Ensures that the given Pnm_ppm height and width are even. It will correct the height and/or width if either is odd. It is a checked runtime error for pmap to be NULL.

    -
      ```
      typedef struct Closure {
            int counter;
            Vcs* values;
            FILE* fp;
      }* Closure;
      ```
        - Closure struct used in apply_compress and apply_decompress.

    -
      ```
      void apply_compress(int i, int j, A2 array2,
                     A2Methods_Object *ptr, void *cl)
      ```
        - cl contains a Closure struct, which has a counter that will cycle between 0 and 3, a file pointer that is NULL, and an allocated array of 4 Vcs structs called values.
        - If the counter is less than four, the array of Vcs structs called values at the counter position is set to the Vcs value within ptr. The counter is updated.
        - If the counter equals 4, it is reset to 0, and apply_compress prints the 32 bit word representing the compressed group of 4 Vcs structs in values.

    -
      ```
      void apply_decompress(int i, int j, A2 array2,
                     A2Methods_Object *ptr, void *cl)
      ```
        - cl contains a Closure struct, which has a counter that will cycle between 0 and 3, a (non-null) file pointer, and an allocated array of 4 Vcs structs called values.
        - If the counter equals 0, apply_decompress fills the 4 Vcs structs in values from the decompressed 32 bit word scanned from the file pointer.
        - Sets the current Vcs struct within ptr equal to values at the position of counter. Counter is increased (0 -> 1 -> 2 -> 3 -> 0...).

    -
      ```
      extern void compress40  (FILE *input);
      ```
        - Prints the given pnm image in bits after compressing to stdout.

    -
      ```
      extern void decompress40(FILE *input);
      ```
        - Prints the given compressed file in proper pnm format after decompressing to stdout. It is a Checked Runtime Error for there to be fewer than 32 * width * height bits or for the header to be incorrect.

- **vcs.h**

    -
      ```
      typdef struct Vcs {
            float y, pb, pr
      } Vcs;
      ```

        - Similar to Pnm_rgb, this struct represents a pixel through Y, Pb, and Pr values.

- **bit_handler.h**
    - `void print_bits(uint32_T word);`
        - Prints the given word in binary in Big Endian format.
    - `uint32_T extract_bits(FILE* fp);`
        - Reads in 32 bits from the given Big Endian formatted binary file and returns a word filled with the 32 bits. If at any point, the end of file character is read in, it would imply that the given file is either too short, or the last word is incomplete. In either case, the program will fail with a Checked Runtime Error.
    - `uint32_T convert_to_bits(Vcs* values);`
        - This function converts, quantizes, and compresses the given 4 Vcs structs within values into a single 32 bit word and returns the word.
    - `void interpret_bits(Vcs* values, uint32_T word);`
        - This function decompresses, quantizes, and converts values packed within the 32 bit word into Vcs structs and fills the values with the 4 structs.

- **algorithms.h**

    -
      ```
      void discrete_cos_transform(Vcs* values, unsigned* a,
                                  int* b, int* c, int* d);
      ```
        - Performs the discrete cosine transform on the Y values within the 4 Vcs structs in values and sets the values of *a, *b, *c, and *d accordingly.

    -
      ```
      void inv_discrete_cos_transform(Vcs* values, uinta,
                                  int b c d);
      ```
        - Performs the inverse of the discrete cosine transform on the given a, b, c, and d values and sets the Y values within the 4 Vcs structs in values.
    - `int quantize_coef(float* xp);`
        - Restricts the range of *xp to ±.3 and quantizes *xp to the range of ±15 and returns the new value.

    -
      ```
      void quantize_chroma(Vcs* values, unsigned* Pb,
                      unsigned* Pr);
      ```
        - Sets the values of *Pb and *Pr from the pb and pr values within the 4 Vsc structs in values.

===

2) How do the components in your program interact? That is, what is the architecture of your program?

The compress40 interface will call the corresponding functions to complete the steps that are necessary to compress the image. The trim function will be called to properly trim the image. The respective rgb_vcs_converter interface functions will be called to convert the pixels into VCS or into RGB. Then the default mapping function will be called with either of the apply functions to properly compress or decompress the image. Within the apply functions, either the print_bits or interpret_bits will be called.

The rgb_vcs_converter interface will convert each pixel in the pmap from RGB color space into component video space and vice versa. No other part of our program knows what the RGB color space consists of.

The print_bit or interpret_bits function will take in a uint32_T that will be returned from the bitpack_newu function. Before the bitpack_newu function is called, the bit_handler interface will call the given functions in the algorithms interface to convert the 2 by 2 blocks of VCS structs to bits.

The algorithms interface will accept corresponding values from the VCS struct and quantize and convert them into the proper bit representations. This interface is the only interface that knows about quantizing values in the VCS struct.

The reason that we developed a separate .h file for our VCS struct is because multiple interfaces need to know about the VCS struct in order to access the values that are stored in the 2 by 2 block. These values will then be converted into bits after being quantized.

==

3) What invariant properties should hold during the solution of the problem?

- The only interfaces that will ever know about the RGB space (represented by Pnm_rgb) are *compress40.h* and *rgb_vcs_converter.h*.
- The interface *bit_handler.h* will be the only part of our program that knows that we are encoding and decoding bits.
- Any Pnm_ppm passed through rgb_to_vcs and then again passed through vcs_to_rgb will be identical to its initial state.
- `print_bits(extract_bits(fp))` should be equivalent to the first 32 bits to which file pointer fp is currently pointing.

==

4) What test cases will you use to convince yourself that your program works?

- **Rgb_vcs_converter.h**
    - We will test this by making two identical Ppm_pnm structs representing the same image. We will apply rgv_to_vcs on one of the Ppm_pnm structs and then apply vcs_to_rgb onto the same struct. We will use the ppmdiff function (implemented during lab inside ppmdiff.c) and ensure that the output is 0 when inputting both Ppm_pnm structs.
    - We will also perform this test on a very small file with a countable amount of pixels. We will print out the Pnm_rgb values then transform the Pnm_ppm using rgb_to_vcs and print out the Vcs values. We will then perform our own manual calculations using the algorithm. and ensure that the output is correct.
- **Compress40.h**
    - We will test our trim function by running our function with images that have an odd number of rows and columns
- **Algorithms.h**
    - We can test each of our algorithm interface functions by passing in a filled closure struct from a 4 by 4 array we have manually created. We can test the output of these functions from the calculations we manually receive
- **Bit_handler.h**
    - `Interpret_bits` and `convert_to_bits` can both be checked by passing in a closure struct with VCS structs from four different pixels. We can check the values that are returned from either function with the manually calculated values that we get. We will check that our final 64 bit word that is returned from `Bitpack_newu` has the first 32 bits set to 0 and the following 32 bits set to what we'd expect based on our own calculations.
    - Extract_bits will be checked by calling `print_bits(extract_bits(fp))` on a known FILE* fp. What is printed should be equivalent to the first 32 bits to which file pointer fp is currently pointing.
    - Extract_bits will also be tested for the cases where an improperly formatted file is passed to the decompressor. After the header, a file is improperly formatted if there are an insufficient amount of words and/or the last word is not 32 bits. In either of these cases, Extract_bits will attempt to parse an EOF character before all 32 bits of a given word are read. We will test this for a file with no bits, and for a file with less than 32 bits. In either case, Extract_bits should have a CRE.

==

5) How will your design enable you to do well on the challenge problem in Section 2.3 on page 12?

Examples of ways that the codeword could be changed:
- -Fields are in different places within the codeword.
- -The size of each field has been changed.
- -Codeword is now printed in hex

Our design enables us to easily change how the codeword is formatted because we calculate each field value independently. If the fields needed to be a greater amount of bits, we could change our functions in our algorithms interface to support this. If the format of the codeword is changed, we can easily change how the field values are placed into the codeword. If the codeword needs to be printed in hex, we could convert the returned values from our algorithms.h interface directly into hex.

==

6) 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?

Information is first lost when we calculate the average Pb and Pr values of a block of 4 pixels. The individuality of each Pb and Pr value is forever lost. Then, more information is lost during the quantization of the averaged Pb and Pr chroma values where the given index of chroma is returned. The float values between ±0.5 are restricted to the range of 0 to 15 (just four bits), which involves rounding.

Information can also be lost during the quantization of b, c and d (which are derived from the Y values of the 2x2 block of Vcs pixels. If *b, c,* and *d* are greater than |0.3|, their values will be changed to the upper bound of 0.3 or the lower bound of -0.3. Information will also be lost when we round a after multiplying it by 511.

In the decompression stage, when quantizing each pixel into the range of 0 to 255, some more information will be lost due to rounding.

Information will be lost if the image is recompressed and re-decompressed. If the pixel values weren't quantized and rounded to the range of 0-255 in the first decompression stage, no information would be lost in the second compression/decompression. This is because each Pr and Pb value in a block of four would be identical, and no rounding would be necessary for any quantization of b, c, d, Pb, and Pr. This will hold true for every time the file is recompressed and re-decompressed.

==

7) Test cases for our bitpacker.h interface

Our testing plan for our bitpacker interface.

- Bitpack_fitsu()
  - Test with case that will fail our CRE (width <= 64)
  - Test case where n won't fit
  - Test case where n will fit
  - Test case where n is small negative (Should return false)
  - Test case where n is positive
  - Test case where (0,0) (Need to ask a course staff about this one)
- Bitpack_fitss()
  - Test with case that will fail our CRE (width <= 64).
  - Test case where width = 1, value = 1 (Need to ask course staff about this one).
  - The amount must be greater than or equal to (-1) * pow(2, width) and less than or equal to pow(2, width) - 1. Check where value equals edge case. E.g (width = 4, value = -16 or width = 4, value = 15).
  - Check where value exceeds edge case by 1. E.g (width = 4, value = -17 or width = 4, value = 16).
- Bitpack_getu()
  - Test with case that will fail our CRE (width <= 64) and (width + lsb <= 64).
  - Test with case where word is zero and width is 0 (We need to ask a course staff about this outcome).
  - Test getting all 64 bits of a word.
  - Test getting 1 digit from each edge, and somewhere in the middle.
  - Test where lsb + width = 64 and lsb > 0.
- Bitpack_gets()
  - Test with case that will fail our CRE (width <= 64) and (width + lsb <= 64).
  - Test with case where value that we are extracting has a leading 1.
  - Test with case where value that we are extracting has a leading 0.
  - Test getting 1 digit from each edge, and somewhere in the middle.
  - Test getting all 64 bits of a word.
  - Test where lsb + width = 64 and lsb > 0.
- Bitpack_newu()
  - Test case that fails CRE (width <= 64 and width + lsb <= 64)
  - Test case that fails CRE (Bitpack_fitsu(value, width))
  - Test for width 64 and width 1.
  - Test where lsb + width = 64 and lsb > 0.

    - Test that small positive value (e.g. 0x0000 0000 0000 44A3) will copy properly for minimum width (4).
    - Test that small positive value (e.g. 0x0000 0000 0000 44A3) will copy properly for excessive width (e.g. 20). This would mean that the leading zeros are still maintained up to bit 20.
- Bitpack_news()
    - Test case that fails CRE (width <= 64 and width + lsb <= 64)
    - Test case that fails CRE (Bitpack_fitss(value, width))
    - Test where lsb + width = 64 and lsb > 0.
    - Test for width 64 and width 1.
    - Test that large negative value (e.g. 0xFFFF FFFF FFFF 0000) will copy properly for minimum width (5).
    - Test that large negative value (e.g. 0xFFFF FFFF FFFF 0000) will copy properly for excessive width (e.g. 20). This would mean that the leading ones are still maintained up to bit 20.
    - Test that small positive value (e.g. 0x0000 0000 0000 44A3) will copy properly for minimum width (5).
    - Test that small positive value (e.g. 0x0000 0000 0000 44A3) will copy properly for excessive width (e.g. 20). This would mean that the leading zeros are still maintained up to bit 20.

===