Design Document: Integer and Logical Operations

Overall Architecture:

Structure:

rpeg/ Cargo.toml docs/ design.md README.md src/ main.rs lib.rs codec.rs color_space.rs dct.rs tests/ test_codec.rs test_dct.rs bitpack/ Cargo.toml src/ bitpack.rs lib.rs tests/ test_bitpack.rs array2/ Cargo.toml src/ array2.rs lib.rs

Component Architectures

The `rpeg` crate is the main application that handles image compression and decompression. It relies on the `bitpack` crate for packing and unpacking data into 32-bit words and the `array2` crate for managing 2D arrays of pixels. The `csc411_image` and `csc411_rpegio` crates are used for reading and writing image files.

rpeg

The `rpeg` crate is responsible for compressing and decompressing images. It consists of the following modules: - `main.rs`: Handles command-line arguments and orchestrates the compression/decompression process. - `lib.rs`: Defines the public interface for the library, including `compress_image` and `decompress_image`. - `codec.rs`: Implements the core compression and decompression logic. - `color_space.rs`: Handles RGB-to-YCbCr and YCbCr-to-RGB conversions. - `dct.rs`: Implements the Discrete Cosine Transform (DCT) and its inverse.

bitpack

The `bitpack` crate provides functions for packing and unpacking signed and unsigned integers into bit fields within a 64-bit word. It consists of: - `bitpack.rs`: Implements the core bitpacking functions, including `fitss`, `fitsu`, `gets`, `getu`, `newu`, and `news`. - `lib.rs`: Defines the public interface for the crate.

array2

The `array2` crate provides a 2D array implementation for storing image data. It consists of: - `array2.rs`: Implements the 2D array data structure. - `lib.rs`: Defines the public interface for the crate.

Universal Testing

Universal Laws

The following universal laws will be tested to ensure the correctness of the implementation: - **Bitpacking Laws**: - `getu(newu(word, width, lsb, value),`

`width, lsb) == value - gets(news(word, width, lsb, value), width, lsb) == value` - **Color Space Laws**: - `ycbcr_to_rgb(rgb_to_ycbcr(image))` `image` (approximately equal due to floating-point precision errors). - **DCT Laws**: - `inverse_dct(dct(y_values))` `y_values` (approximately equal due to floating-point precision errors).

Testing Strategy

Each component will be tested in isolation using unit tests. The following test cases will be implemented: - **Bitpacking**: - Test `fitss` and `fitsu` with values that fit and do not fit in the specified bit widths. - Test `gets` and `getu` with various bit fields and edge cases (e.g., extracting all 64 bits). - Test `newu` and `news` by packing and unpacking values and verifying the results. - Verify that `newu` and `getu` are inverses. - Verify that `news` and `gets` are inverses. - **Color Space**: - Test RGB-to-YCbCr and YCbCr-to-RGB conversions with known values. - Test clamping of out-of-range values. - Verify that RGB-to-YCbCr and YCbCr-to-RGB conversions are approximate inverses. - **DCT**: - Test the DCT and inverse DCT with known inputs and outputs. - Test edge cases (e.g., all zeros, all ones).

rpeg

1. `main.rs`

This file will handle command-line arguments and orchestrate the compression or decompression process.

Functions:

- `main()`
  - Parse command-line arguments (`-c` for compress, `-d` for decompress, and the input file).
  - Call `compress_image()` or `decompress_image()` from `lib.rs` based on the command-line option.
  - Handle standard input if no file is provided.

2. `lib.rs`

This file defines the public interface of your library. It will coordinate the compression and decompression processes by calling functions from `codec.rs` and other modules.

Functions:

- `compress_image(input: &str) -> Result<(), Box<dyn std::error::Error>>`
  - Read the PPM image from the input file or standard input using `csc411_image`.

  - Call `codec::compress_image_data()` to perform the compression steps.
  - Write the compressed binary image to standard output using `csc411_rpegio`.
- `decompress_image(input: &str) -> Result<(), Box<dyn std::error::Error>>`
  - Read the compressed binary image from the input file or standard input using `csc411_rpegio`.
  - Call `codec::decompress_image_data()` to perform the decompression steps.
  - Write the decompressed PPM image to standard output using `csc411_image`.

3. `codec.rs`

This file will contain the core logic for compression and decompression. To avoid confusion with the functions in `lib.rs`, we will rename the functions to better reflect their purpose.

Functions:

- `compress_blocks(image: PpmImage) -> Result<Vec<u32>, Box<dyn std::error::Error>>`
  - Compress each 2x2 block of pixels into a 32-bit word using the `bitpack` crate.
  - Return a vector of compressed 32-bit words.
- `decompress_blocks(compressed_data: Vec<u32>, width: u32, height: u32) -> Result<PpmImage, Box<dyn std::error::Error>>`
  - Decompress each 32-bit word into a 2x2 block of YCbCr pixels using the `bitpack` crate.
  - Return the decompressed `PpmImage`.

4. color_space.rs

This file will handle color space conversions and related operations.

Structs:

- `pub struct YCbCrPixel`
  - represents a single pixel in the YCbCr color space.
  - `y`: Luma
  - `pb`: Blue-difference chroma component
  - `pr`: Red-difference chroma component
- `pub struct YCbCrImage`
  - represents the entire image in the YCbCr color space.
  - `width`, `height`, `pixels`

Functions:

- `pub fn rgb_to_ycbcr(rgb_image: &RgbImage) -> YCbCrImage`
  - Convert each image from RGB to YCbCr using the provided formulas.
    - y = 0.299 * r + 0.587 * g + 0.114 * b;
    - pb =-0.168736 * r- 0.331264 * g + 0.5 * b;
    - pr = 0.5 * r- 0.418688 * g- 0.081312 * b;
  - Store converted pixels in a YCbCrImage
- `pub fn ycbcr_to_rgb(ycbcr_image: &YCbCrImage) -> RgbImage`
  - Convert each pixel from YCbCr to RGB using the following formulas:
    - r = 1.0 * y + 0.0 * pb + 1.402 * pr;
    - g = 1.0 * y- 0.344136 * pb- 0.714136 * pr;
    - b = 1.0 * y + 1.772 * pb + 0.0 * pr;
  - Store the converted pixels in an RgbImage
- `pub fn approx_equal(x: f32, y: f32) -> bool`
  - Verify inverse properties using:
  - `[(x-y)^2]/[x^2 + y^2]`
  - True if `x` & `y` are zero.

Helper Functions:

- `trim_image(image: YCbCrImage) -> YCbCrImage`
  - Trim the last row or column if necessary to make the width and height even.
- `average_chroma(block: &[YCbCrPixel; 4]) -> (f32, f32)`
  - Calculate the average Pb and Pr values for a 2x2 block.

Unit Tests:

- Test RGB to YCbCr and YCbCr to RGB conversions with known values.
- Test clamping of out-of-range values.
- Test that RGB-to-YCbCr and YCbCr-to-RGB conversions are approximate inverses.

5. dct.rs

This file will implement the mathematical functions for the Discrete Cosine Transform (DCT) and its inverse.

Functions:

- dct(y_values: [f32; 4]) -> (f32, f32, f32, f32)
  - Perform the DCT on four Y values and return the coefficients (a, b, c, d).
- inverse_dct(a: f32, b: f32, c: f32, d: f32) -> [f32; 4]
  - Perform the inverse DCT to reconstruct the Y values.

6. tests/

This directory will contain unit tests for each module.

Files

- `test_codec.rs`
  - Test the `compress_block()` and `decompress_block()` functions.
- `test_dct.rs`
  - Test the `dct()` and `inverse_dct()`
  - Test edge cases (all zeros, all ones)

bitpack

1. bitpack.rs

This file contains the implementation of all the bitpacking functions. It is responsible for packing and unpacking signed and unsigned integers into specific bit fields within a 64-bit word. This functionality is crucial for the compression algorithm, where DCT coefficients and chroma indices are packed into 32-bit words.

Functions:

- `fitss(n: i64, width: u64) -> bool`
  - Determines if a signed integer `n` can fit into a bit field of `width` bits.
- `fitsu(n: u64, width: u64) -> bool`
  - Determines if an unsigned integer `n` can fit into a bit field of `width` bits
- `gets(word: u64, width: u64, lsb: u64) -> Option<i64>`
  - Extracts a signed integer ffrom a bit field in `word`
  - Check if `width + lsb` exceeds 64
    - Return `None` if true
  - Sign-extend the result if needed
- `getu(word: u64, width: u64, lsb: u64) -> Option<u64>`
  - Extracts an unsigned integer from a bit field in `word`
  - Check if `width + lsb` exceeds 64 (return `None` if true).
  - Extract the bit field using bitwise operations
- `newu(word: u64, width: u64, lsb: u64, value: u64) -> Option<u64>`
  - Updates a bit field in `word` with an unsigned `value`
  - Uses `fitsu` to check if `value` fits in `width` bits
  - Shift `value` to the correct position and `OR` it into `word`
- `news(word: u64, width: u64, lsb: u64, value: i64) -> Option<u64>`
  - Updates a bit field in `word` with a signed `value`
  - Uses `fitss` to check if `value` fits in `width`
  - Clear the target area in `word`
  - Shift `value` to the correct position and `OR` it into `word`

Helper Functions:

```
#[inline]
pub fn safe_shl(value: u64, shift: u64) -> u64 {
    if shift >= 64 { 0 } else { value << shift }
}

#[inline]
pub fn safe_shr(value: u64, shift: u64) -> u64 {
    if shift >= 64 { 0 } else { value >> shift }
}

#[inline]
pub fn safe_sar(value: i64, shift: u64) -> i64 {
    if shift >= 64 {
        if value >= 0 { 0 } else { -1 } // Preserve the sign bit
    } else {
        value >> shift
    }
}
```

- Helper functions to ensure consistent results when shifting and improves portability.

2. lib.rs

This file defines the public interface of the `bitpack` crate. It does not contain any function implementations but re-exports the functions from `bitpack.rs` so that they can be used by other crates.

Functions:

- Re-exports from `bitpack.rs`:
  - `pub use bitpack::{fitss, fitsu, gets, getu, newu, news};`
  - These functions are made available to other crates that use bitpack

3. Testing

The `bitpack` module should be thoroughly tested to ensure correctness. Suggested test cases:

- `test_bitpack.rs`
  - Test the fitss and fitsu functions with values that fit and do not fit in the specified bit widths.
  - Test the gets and getu functions with various bit fields and edge cases (e.g., extracting all 64 bits).
  - Test the newu and news functions by packing and unpacking values and verifying the results.

- Test shifting by 64 bits
- Test packing and unpacking values at the boundaries of their bit fields
- Test invalid inputs (`width + lsb > 64`)
- Verify that `newu` and `getu` are inverses.
- Verify that `news` and `gets` are inverses.

Places Where Information Could Be Lost

1. Trimming the Image
   - Loss: If the image width or height is odd, the last row or column is trimmed to make them even.
   - Impact: Pixels in the trimmed row or column are permanently lost.
2. Color Space Conversion (RGB to YCbCr)
   - Loss: Floating-point precision errors during the conversion from RGB to YCbCr.
   - Impact: Small inaccuracies in the Y, Pb, and Pr values.
     - These losses should be negliable, but could potentially compound.
3. Chroma Subsampling (Averaging Pb and Pr)
   - Loss: The Pb and Pr values for each 2x2 block are averaged, reducing the resolution of the chroma components.
   - Impact: Fine color details are lost, especially in areas with high color variation.
4. Quantization of Chroma Values
   - Loss: The csc411_arith::index_of_chroma function quantizes Pb and Pr values into 4-bit indices.
   - Impact: Precision is lost because the original floating-point values are mapped to a limited set of discrete values.
5. Discrete Cosine Transform (DCT)
   - Loss: The DCT coefficients (b, c, d) are quantized to 5-bit signed values, assuming they lie between -0.3 and 0.3.
   - Impact: High-frequency details (represented by b, c, d) are lost if their values fall outside the ± 0.3 range or are rounded during quantization.
6. Bit Packing
   - Loss: The DCT coefficients and chroma indices are packed into a 32-bit word, which involves truncation or rounding.
   - Impact: Small precision errors may occur due to truncation of floating-point values or rounding of integers.
7. Decompression (Inverse Operations)
   - Loss: During decompression, the inverse operations (e.g., inverse DCT, YCbCr to RGB conversion) introduce additional floating-point precision errors.
   - Impact: These errors compound the losses from the compression step.

Summary:

Places Where Information Could Be Lost

Step	Type of Loss	Impact
Trimming	Pixel loss	Permanent loss of pixels in the last row or column.
RGB to YCbCr conversion	Floating-point precision errors	Small inaccuracies in color representation.
Chroma subsampling	Resolution reduction	Loss of fine color details.
Chroma quantization	Quantization errors	Limited precision for Pb and Pr values.
DCT coefficient quantization	Quantization errors	Loss of high-frequency details.
Bit packing	Truncation or rounding errors	Small precision errors in DCT coefficients and chroma indices.
Decompression	Floating-point precision errors	Compounded inaccuracies during inverse operations.

Subsequent Compressions

Further loss occurs during subsequent compressions due to: - **Accumulation of quantization errors**: Each compression step introduces rounding errors, which compound over time. - **Loss of high-frequency details**: High-frequency details are lost during the first compression and further degraded in subsequent compressions. - **Amplification of floating-point errors**: Floating-point errors from color space conversions and inverse DCT operations compound with each cycle. - **Chroma subsampling artifacts**: Repeated subsampling further blurs color details. ### Real-World Scenario Most image formats used online, such as JPEG, rely on lossy compression. Each time an image is downloaded, edited, or re-uploaded, it undergoes recompression, leading to data loss and a gradual degradation of quality. This phenomenon, known as generation loss, is why images shared repeatedly on social media often appear significantly worse than the original. Common signs of this degradation include: - Blurring: Loss of fine details and sharpness. - Color Banding: Visible bands of color in gradients. - Blockiness: Noticeable pixel blocks, especially in areas with high detail. - Noise: Unwanted artifacts, such as ringing or moiré patterns. - Loss of Contrast and Saturation: Colors may appear washed out or less vibrant.

This real-world example mirrors the information loss observed in the compression and decompression process of the rpeg program, where repeated cycles of compression and decompression lead to cumulative degradation.