ENGR 456: Final Project

Administrative Information:

- **This assignment will be the last RISC-V Coding assignment. There will be no other RISC-V coding.**
- **Any student who can successfully complete this assignment such that I can run each part back-to-back and generate a valid (and correct) JPEG will be given a +20% boost to the final exam score. I.e. if the exam is worth 100 points, you will be given 20 points on top of your score.**
- **If any student earns the extra credit from doing exceptionally well on this assignment, and still ends with a grade that is not satisfactory, you may email me and I will ensure you finish will a reasonably good grade. Likewise, if you end with a grade that you are not happy with, and you haven't completed this assignment, your negotiating leverage will be weak.**
- **You have 12/6 11:59 pm to finish this assignment. This is the Saturday night before your final. We will also have a "dead week" the week before the final where the content in lecture will not be on the final but be useful for your future interviews. You may ask questions on the final project during this week also.**
- **You may AT ANY TIME post a question to discord about this assignment.**
- **This assignment will not be extended, there will not be redo's, there will not be late acceptances. It is due 2 days before the final, and therefore there is not extra time.**
- **A main file will be on Canvas:**
    - **It will run each part sequentially, but provide "sample arrays" of what the output should be from the previous function. You should start in order as that's generally how difficulty will scale.**

RISC-V (190 Points):

Overview:

You are tasked with building a 1-channel, baseline, 8-bit color, Huffman-encoded, JPEG encoder in RISC-V Assembly. This is a truly daunting task to start with, but luckily, I will be providing the necessary algorithms/building blocks to get everyone going. You may recognize the term JPEG as the file format used for almost all images on the modern-day web browser, but I doubt you've taken the time to appreciate the nuance of the compression algorithm, and why it has lasted the test of time. The JPEG algorithm breaks up an image into 8x8 pixel blocks (matrices) and performs the following steps:

- RGB -> YCbCr Conversion and Level-Shift (We will only be using the gamma (Y) channel)
- DCT-II (We will use the AA&N algorithm flavor)
    - DCT-I row-wise
    - DCT-I column-wise
- Quantization (We will use a provided Quantization table)
- Zigzagging
- Run Length Encoding

- Huffman Encoding (We will use a provided Huffman table)
- Bitstream concatenation

PLEASE NOTE: YOU MAY SEE WHILE(1) ON THE OUSTIDE OF SOME PROVIDED CODE, YOU CAN IGNORE THIS

RGB -> YCbCr + Level-Shifting (20 points):

- **Name:** rgb2y
- **Difficulty:** Easy
- **Required Equation:**
  - ```
    (SRA((R*38 + G*75 + B*15 - 16320), 7));
    ```
- **Description:**
  - JPEG's input is an 8x8x3 matrix, where the "3" denotes the color channel in RGB. You will need to use the RGB value of a given pixel (there are 8x8 = 64 pixels in the block) to produce the value in a different color space (Y or Luminance).
- **Input:**
  - X10 will point to a 8x8 block of unsigned 8-bit Red values
  - X11 will point to a 8x8 block of unsigned 8-bit Green values
  - X12 will point to a 8x8 block of unsigned 8-bit Blue values
  - X13 will point to the resulting 8x8 Y block (matrix)
- **Output:**
  - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.
- **Significance:**
  - By converting to YCbCr, we are able to take advantage of the human eyes sensitivity to color. We are much more sensitive to changes in Y (Luminance or brightness) than CbCr (Chroma). This means in more complete JPEG encoders, we can compress and trim the Chroma values more ambitiously than the Luminance values. This leads to more space savings while maintaining image fidelity! We level shift (subtract 128 to convert the unsigned 8-bit values to signed 16-bit values) to get an easier to interpret result from the future DCT-II transform (and one with more 0's which will be important later!)

DCT-I column-wise (15 points):

- **Name:** dct_1c
- **Difficulty:** Medium
- **Required Equation:**
    ```

```
for(uint8_t i = 0; i < 8; i++){ // for each column

    a00 = x[i+ 0] + x[i+56];
    a10 = x[i+ 8] + x[i+48];
    a20 = x[i+16] + x[i+40];
    a30 = x[i+24] + x[i+32];
    a40 = x[i+24] - x[i+32];
    a50 = x[i+16] - x[i+40];
    a60 = x[i+ 8] - x[i+48];
    a70 = x[i+ 0] - x[i+56];

    a01 = a00 + a30;
    a11 = a10 + a20;
    a21 = a10 - a20;
    a31 = a00 - a30;
    neg_a41 = a40 + a50;
    a51 = a50 + a60;
    a61 = a60 + a70;

    a22 = a21 + a31;

    a23 = a22 * 11585;
    mul5 = (a61 - neg_a41) * 6270;
    a43 = (neg_a41 * 8867) - mul5;
    a53 = a51 * 11585;
    a63 = (a61 * 21407) - mul5;

    temp1 = a70 << 14;
    a54 = temp1 + a53;
    a74 = temp1 - a53;

    // Keeping everything homogenous
    y[i+0] = a01 + a11 << 2;
    y[i+32] = a01 - a11 << 2;

    // Only 10 bits are required before the decimal
    temp1 = a31 << 14;
    temp = temp1 + a23;
    y[i+16] = (temp + 0x800) >> 12; // Lazy Rounding
    temp = temp1 - a23;
    y[i+48] = (temp + 0x800) >> 12; // Lazy Rounding
    temp = a74 + a43;
    y[i+40] = (temp + 0x800) >> 12; // Lazy Rounding
    temp = a54 + a63;
    y[i+8] = (temp + 0x800) >> 12; // Lazy Rounding
    temp = a54 - a63;
    y[i+56] = (temp + 0x800) >> 12; // Lazy Rounding
    temp = a74 - a43;
    y[i+24] = (temp + 0x800) >> 12; // Lazy Rounding
}
```

- **Description:**
    - You will perform the first of two steps in the DCT-II transform, a DCT-I column-wise transform. We will be implementing the AA&N algorithm which will give the proper DCT output (after we scale it in the Quantization step). For this reason, do not compare your output to a sample DCT-I output without expecting there to be a scaling difference. Instead use the provided input/output pair to test your work.
- **Input:**
    - X10 will be the address to a 8x8 block of signed 16-bit Luminance values.
    - X11 will be the address of the resulting 8x8 block (matrix). Rounded to 16 bits.
- **Output:**
    - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.
- **Significance:**
    - The DCT-II transform can be completed by doing 8 column-wise DCT-I transforms and 8 row-wise DCT-I transforms. The DCT-II transfers the image to the frequency domain to compress the high-frequency part of the image more-aggressively than the low-frequency part of the image.

DCT-I row-wise (15 points):

- **Name:** dct_1r
- **Difficulty:** Easy (if you've done dct_1c already)
- **Required Equation:**

```
for(uint8_t j = 0; j < 64; j = j + 8){ // for each row

    a00 = x[j+0] + x[j+7];
    a10 = x[j+1] + x[j+6];
    a20 = x[j+2] + x[j+5];
    a30 = x[j+3] + x[j+4];
    a40 = x[j+3] - x[j+4];
    a50 = x[j+2] - x[j+5];
    a60 = x[j+1] - x[j+6];
    a70 = x[j+0] - x[j+7];

    a01 = a00 + a30;
    a11 = a10 + a20;
    a21 = a10 - a20;
    a31 = a00 - a30;
    neg_a41 = a40 + a50;
    a51 = a50 + a60;
    a61 = a60 + a70;

    a22 = a21 + a31;

    a23 = a22 * 11585;
    mul5 = (a61 - neg_a41) * 6270;
    a43 = (neg_a41 * 8867) - mul5;
    a53 = a51 * 11585;
    a63 = (a61 * 21407) - mul5;

    temp1 = a70 << 14;
    a54 = temp1 + a53;
    a74 = temp1 - a53;

    // Keeping everything homogenous
    y[j+0] = a01 + a11;
    y[j+4] = a01 - a11;

    // Only 10 bits are required before the decimal
    temp1 = a31 << 14;
    temp = temp1 + a23;
    y[j+2] = (temp + 0x2000) >> 14; // Lazy Rounding
    temp = temp1 - a23;
    y[j+6] = (temp + 0x2000) >> 14; // Lazy Rounding
    temp = a74 + a43;
    y[j+5] = (temp + 0x2000) >> 14; // Lazy Rounding
    temp = a54 + a63;
    y[j+1] = (temp + 0x2000) >> 14; // Lazy Rounding
    temp = a54 - a63;
    y[j+7] = (temp + 0x2000) >> 14; // Lazy Rounding
    temp = a74 - a43;
    y[j+3] = (temp + 0x2000) >> 14; // Lazy Rounding
}
```

- **Description:**
  - This is the same thing as the dct_1c, but done across rows instead of columns.

- **Input:**
  - X10 will be the address to a 8x8 block of signed 16-bit DCT-I column-wise values.
  - X11 will be the address of the resulting 8x8 block (matrix). Rounded to 16 bits.
- **Output:**
  - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.

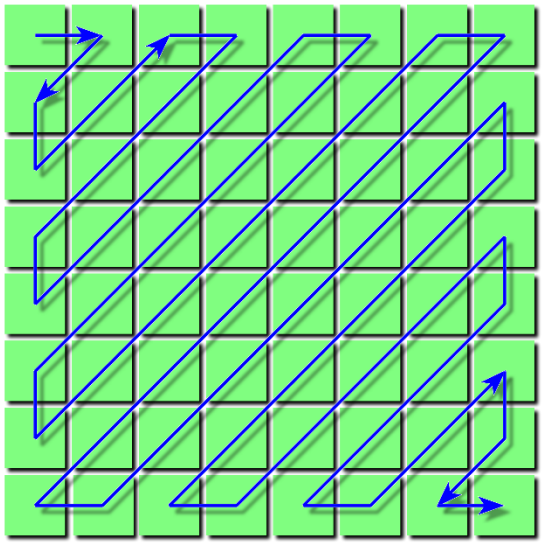
Quantization (20 points):

- **Name:** q_y
- **Difficulty:** Easy
- **Required Equation:**
  -
```
for (uint8_t i=0; i<64; i++){
    temp2 = x[i] * Q_Y[i];
    y[i] = (temp2 + 0x8000) >> 16;
}
```
- **Description:**
  - We will be element-wise multiplying (NOT MATRIX MULTIPLICAITON) two matricies
- **Input:**
  - X10 will be the address of an 8x8 input block
  - X11 will be the address of an 8x8 quantization table
  - X12 will be the address of the resulting 8x8 quantized block
- **Output:**
  - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.
- **Significance:**
  - We will be trimming off very high frequency values so they have less weight in our final image, this gives us more room to compress the image without loosing quality.

Zigzagging (20 points):

- **Name:** zigzag
- **Difficulty:** Easy
- **Required Equation**

```
y[ 0] = x[ 0];
y[ 1] = x[ 1];
y[ 5] = x[ 2];
y[ 6] = x[ 3];
y[14] = x[ 4];
y[15] = x[ 5];
y[27] = x[ 6];
y[28] = x[ 7];
y[ 2] = x[ 8];
y[ 4] = x[ 9];
y[ 7] = x[10];
y[13] = x[11];
y[16] = x[12];
y[26] = x[13];
y[29] = x[14];
y[42] = x[15];
y[ 3] = x[16];
y[ 8] = x[17];
y[12] = x[18];
y[17] = x[19];
y[25] = x[20];
y[30] = x[21];
y[41] = x[22];
y[43] = x[23];
y[ 9] = x[24];
y[11] = x[25];
y[18] = x[26];
y[24] = x[27];
y[31] = x[28];
y[40] = x[29];
y[44] = x[30];
y[53] = x[31];
y[10] = x[32];
y[19] = x[33];
y[23] = x[34];
y[32] = x[35];
y[39] = x[36];
y[45] = x[37];
y[52] = x[38];
y[54] = x[39];
y[20] = x[40];
y[22] = x[41];
y[33] = x[42];
y[38] = x[43];
y[46] = x[44];
y[51] = x[45];
y[55] = x[46];
y[60] = x[47];
y[21] = x[48];
y[34] = x[49];
y[37] = x[50];
y[47] = x[51];
y[50] = x[52];
y[56] = x[53];
y[59] = x[54];
y[61] = x[55];
y[35] = x[56];
y[36] = x[57];
y[48] = x[58];
y[49] = x[59];
y[57] = x[60];
y[58] = x[61];
y[62] = x[62];
y[63] = x[63];
```

- **Description:**
  - We will be reorganizing each 8x8 block into “zigzag” order, meaning you will read them according to this photo:
  - 
- **Input:**
  - X10 will be the address of an 8x8 input block
  - X11 will be the address of the resulting 8x8 zigzagged block
- **Output:**
  - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.
- **Significance:**
  - When we move to the next step (run-length encoding) we will be more likely to get a longer “run” of zeros when reading it in this fashion.

Run-Length Encoding (40 points):

- **Name:** rle
- **Difficulty:** Medium
- **Required Equation:**

```
y_code[0] = 0x0000; // not used, but put here for easier coding on later steps
y_val[0] = x[0]; // EOI/DC Pass along
zero_count = 0;
out_index = 0;
for (i = 1; i < 64; i++){
    val = x[i];
    if (val == 0) {zero_count++;}
    else {
        while (zero_count > 15){
            y_code[out_index] = 0x00f0;
            y_val[out_index] = 0x0000; // not used, but put here for easier coding on later steps
            zero_count -= 16;
            out_index++;
        }
        y_code[out_index] = zero_count << 4 | get_size(val);
        y_val[out_index] = val;
        zero_count = 0;
        out_index++;
    }
}
if x[63] != 0 {
    y_code[out_index] = 0x0000;
    y_val[out_index] = 0x0001;
}
```

```
uint16_t get_size(short val){
if (val == 0) {return 0;}
if (val < 0) {val = ~val + 1;} // use 2's comp to find size
// algo: http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogObvious
uint16_t size = 0;
if (val & 0xff00) {
    val = val >> 8; // UNSIGNED >>
    size |= 8;
}
if (val & 0xf0) {
    val = val >> 4; // UNSIGNED >>
    size |= 4;
}
if (val & 0xc) {
    val = val >> 2; // UNSIGNED >>
    size |= 2;
}
if (val & 0x2) {
    size |= 1;
}
return (size+1);
```

- **Description:**
  - You will be reading each value in the matrix one by one (reading an complete first row before moving onto the next row) and finding how many zeros came before the value, and the size of the value.
    - **Ex.**

-26	**-3**	**0**	**-3**	**-2**	**-6**	**2**	**-4**

1	**-3**	**1**	**1**	**5**	**1**	**2**	**-1**
1	**-1**	**2**	**0**	**0**	**0**	**0**	**0**
-1	**-1**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Take the highlighted value as an example. It has 5 zeros before it, and it has length 1 (1's complement not counting the sign bit)
Zeros: 5 -> 0101
Size: 1 -> 0001
Zeros+Size: 01010001
Val: -1 -> 0

- **Cases:**
  - If there are no more non-zero values in the matrix, the value 00000000 is put in the code matrix. This is the EOB code. Note: a EOB code is not used when the 64th value of the matrix is non-zero.
  - If there are more non-zero values in the matrix, but you've already hit 16 zeros, you must encode 11110000, the ZRL code.
  - On the most common case, you will encode zzzz_ssss where zzzz denotes the number of zeros (represented as 4-bit unsigned) before the encountered non-zero value. Ssss denotes the size of the non-zero value (in 1's complement without its sign bit). You will also write the non-zero value to the value array.

- **Input:**
  - X10 will be the input value matrix
  - X11 will be the resulting code matrix
  - X12 will be the resulting value matrix (1's complement, leave off the sign-bit)
- **Output**
  - Do not change the value of x10 or x11. They are memory locations. You may sw/lw, but do not overwrite the registers.
- **Significance:**
  - We have now collapsed all the runs of zeros into 8-bit codes!

Huffman-Encoding (30 points):

- **Name:** huffman_y
- **Difficulty:** Medium
- **Required Equations:**

```
uint8_t get_size_AC(uint16_t val){
    // algo: http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogObvious
    uint8_t size = 0;
    if (val & 0xff00) {
        val = SHR(val, 8);
        size |= 8;
    }
    if (val & 0xf0) {
        val = SHR(val, 4);
        size |= 4;
    }
    if (val & 0xc) {
        val = SHR(val, 2);
        size |= 2;
    }
    if (val & 0x2) {
        size |= 1;
    }
    size++;
    if (size < 2){return 2;}
    return size;
}
```

```
size_of_bits = get_size(x_val[0]);
out_index = 0
y_val[out_index] = huffman_table_DC_Y[size_of_bits];
y_size[out_index] = huffman_table_DC_Y_sizes[size_of_bits];
out_index++;
y_val[out_index] = x_val[0];
y_size[out_index] = size_of_bits;
out_index++;
in_index = 1;
code = 1; // to start loop
while (code > 0) {
    code = x_code[in_index];
    val = x_val[in_index];
    in_index++;
    while (code == 0xf0) {
        y_val[out_index] = get_AC_Y_val(code);
        y_size[out_index] = get_size_AC(get_AC_Y_val(code));
        out_index++;
        code = x_code[in_index];
        val = x_val[in_index];
        in_index++;
    }

    if ((val == 1) && (code == 0)) {break;} // code 0 and val 1 is the signal for EOB skip

    y_val[out_index] = get_AC_Y_val(code);
    y_size[out_index] = get_size_AC(get_AC_Y_val(code));
    out_index++;

    if (code == 0) {break;} // on EOB you don't write back val
    y_val[out_index] = val;
    y_size[out_index] = get_size(val);
    out_index++;
}
```

- **Description:**
  - You will read a code from the code array, and then use it to address the Huffman table to pull the resulting Huffman code. You will then concatenate the 16-bit Huffman value with the matched index of the value from the value array, and write it out into the val and size array respectively. If EOB or ZRL are read from the code array, the code (after Huffman encoding) is written without any value attached. You may return when encountering a EOB, or after processing your 64th code from the code array (as there can never be more than 64 codes.
  - Typically the first value is difference-encoded against the value of the previous block, but we are only doing one block so we are just going to pass it along.
- **Input:**
  - X10 will be the address to the value array
  - X11 will be the address to the code array
  - X12 will be the address to the DC Huffman Table
  - X13 will be the address to the DC Huffman Sizes table
  - X14 will be the address to the Huffman Table

    - X15 will be the address to the resulting value array
    - X16 will be the address of the resulting size array
- **Significance:**
    - The Huffman codes are smaller than 8-bit for the more frequent codes, and larger than 8-bit for the less frequent codes. This allows overall space to be saved, as the more frequent codes take less space. Huffman encoding and Run-Length encoding do not affect image quality.

Bit Concatenation (30 points):

- **Name: organize**
- **Difficulty: Hard**
- **Required Equations:**
    - N/A – I want you to come up with the algorithm for this as it is mainly bit manipulation and will be good practice for future interviews.
- **Description:**
    - You need to take variable length bit streams and combine them into an array of 8 bit values. So if you have 0110 and 1010010 come in, you need to combine them to 01101010010, and then you need to write out the first 8 bits (byte) to your output array leaving 010, which will be combined with the next incoming bit streams. At the end of the input array, if you still have bits left over, pad them with 1's at the end until it makes a byte then write it.
    - If the value you are to write is negative (for example your first negative should be -43), you will need to subtract 1 from the value, and then discard all the bits outside of the specified size. For example, for -43 (1111_1111_1101_0101) your size array says this is size 6. You need to subtract 1 (1111_1111_1101_0100), then get rid of all bits to the left of the 6^{th} bit. (0000_0000_0001_0101). Now you may continue with the normal algorithm.
    - In the event you write the byte 0xff or 255 to your byte array, you must "byte stuff" by writing 0x00 or 0 as your next byte. JPEG relies on 0xff as a special byte for settings on top of the file, so if you naturally produce one, you need to put 0x00 right afterwards to tell the decoder it's not a special control byte.
- **Inputs:**
    - X10 will be the address to the val array
    - X11 will be the address to the size array (that contains how many valid bits are in each val in the val array)
    - X12 will be the address to your output array.

Report (30 Points):

Describe 3 changes/optimizations that can be made to a RISC-V processor to improve its JPEG compression performance. (5 sentences per idea minimum). You will be graded on how realistic it is to implement the change, if it would actually work, and if the change is correctly detailed. Do not simply say something like "add a cache," instead explain what type of cache, and why, ect.