Group 14C: J. Lagasse, A. Sjogren, O. Onyenokwe, L. Wu
ECE 354: Lab 4 Report
4/19/2017

1. Introduction:

The purpose of this lab is to combine what we learned in all three previous labs to compress an image using primarily our own verilog and C code. By doing this, one is supposed to get a better understanding of how the ARM core processor and the FPGA work together in one system. One also learns how to change and update QSYS hardware designs and the verilog code that they interface with. This system works by first taking a photo with the camera hardware. This photo then is traversed and converted into a one bit per pixel array, such that each pixel on the screen becomes converted into either a 1 or 0 depending if it is black or white. This data then gets passed into one FIFO buffer, which handles feeding data into the RLE, which then passes its output to another FIFO buffer. This last FIFO buffer then sends its output to our C code on the ARM processor, which we then save onto the SDRAM, decompress, and print to the screen. We then find the compression ratio, which is the total bytes of the pixels on the screen divided by the bytes retrieved from the output, and print this out to the console. Once all this is completed, one can clearly see how the FPGA communicates with an ARM processor and how hardware simulated in Verilog can be utilized with software to complete a task such as file compression.

2. Detailed Procedure:

Hardware

1. Edit the hardware files from Lab 2 to incorporate 8 new input/output PIO pins in QSYS and the verilog files. Wire all components in QSYS. Obtain the RLE verilog code from Lab 3 and the required files listed on the course page. Place all these files in the same folder.
2. Edit the verilog code so that it is able to handle all the new input/ output PIO pins and that the RLE and FIFO buffers are instantiated and wired together properly.
3. Compile onto DE1-SoC board. Note that the camera hardware is used for this lab.

Software

1. Use the Altera Monitor program to compile C code onto the board. We will be using capture_image.c
2. Edit capture_image.c so that it is able to do the following:
    a. Convert the captured image into a one bit per pixel representation to be used to shorten compression. Organize this data to work with the RLE code correctly.
    b. Send the one bit per pixel data to the RLE hardware so that it can perform compression.
    c. Store the resultant compressed image onto the SDRAM.
    d. Write a function to decompress the RLE compressed image properly and display the decompressed image.
    e. Compute and display the the compression ratio for each picture.

3. Once all these operations are coded, test and edit code until it works properly and consistently.

3. Hardware Changes:

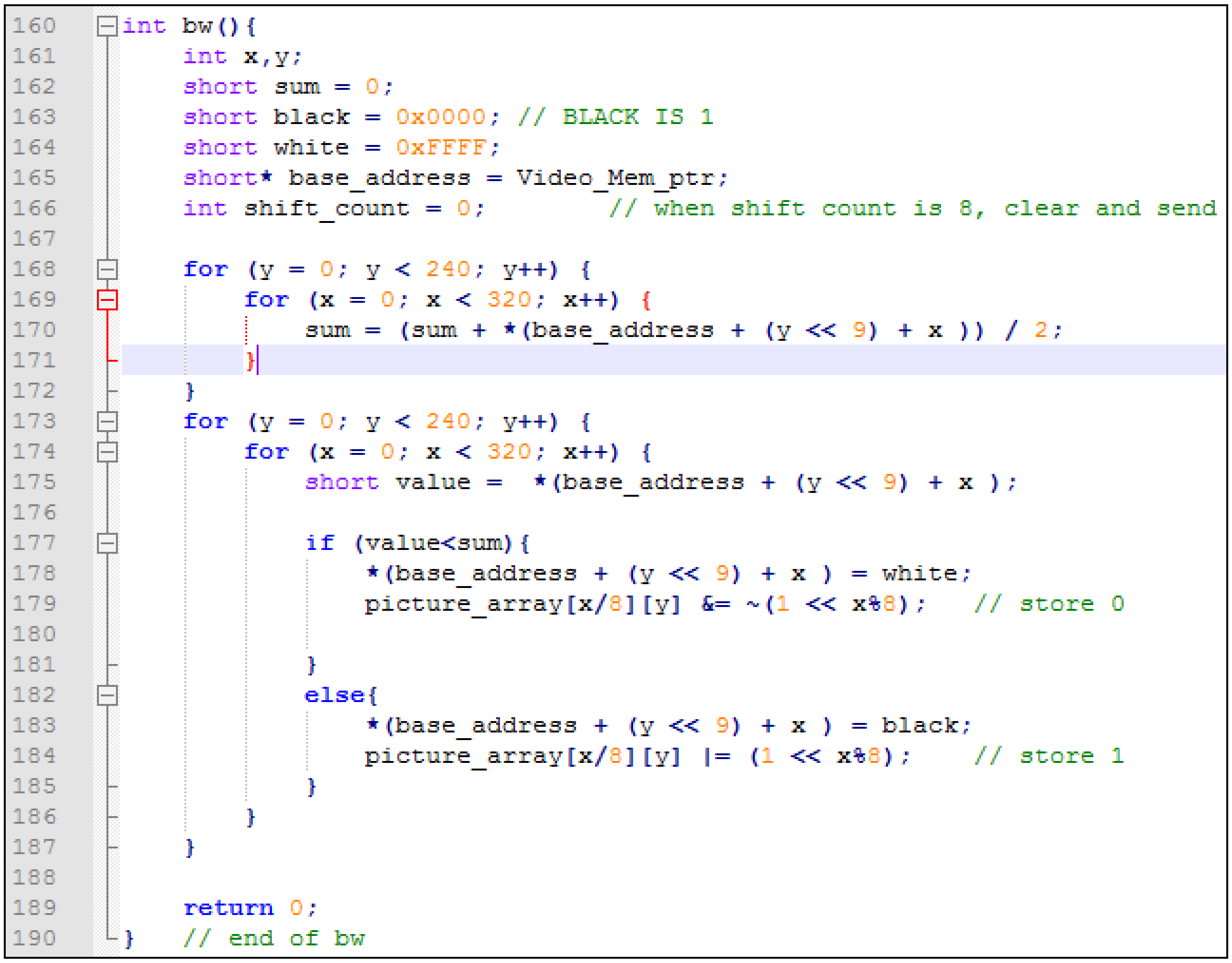
The hardware changes for this lab were fairly straightforward. First, we took the QSYS files from Lab 2 and added eight input/output PIO ports to support the RLE encoder program. These 8 ports had to be wired together, their data needed to be exported, and their base addresses needed to be checked to make sure they were compatible with the verilog files. These eight ports were also used by name in the verilog code, which we also had to update to make two FIFO buffer modules (one for input, one for output) and a RLE module. We did not specifically need to change the RLE verilog code from Lab 3, however we did for our project in an attempt to try to solve hardware errors we encountered. Once all these changes were made and all necessary files added to the hardware, it was compiled, which took a half hour or more each time. Once it compiled properly, no other changes to the hardware where needed unless hardware bugs were found, which would involve making necessary changes and recompiling.

4. Software Changes:

The software changes for this lab were less straightforward. We began by taking our old capture_image.c code and removing the functions that we did not need, such as mirroring an image, converting to grayscale, etc. The only methods needed from the existing code was the code to take pictures, resume the video, and convert the image to black and white (bw Figure 1). We added code the the black and white converter to save the black and white pixels into one bit representations on a double char array. Because each char is 8 bits and the input to the RLE is 8 bits, we were able to set up the array such that each char was exactly one unit of data to send to the RLE. We also reorganized the data in each byte so that the RLE would read it correctly, with the least significant byte being the first pixel since the RLE reads the data starting with the least significant bit.

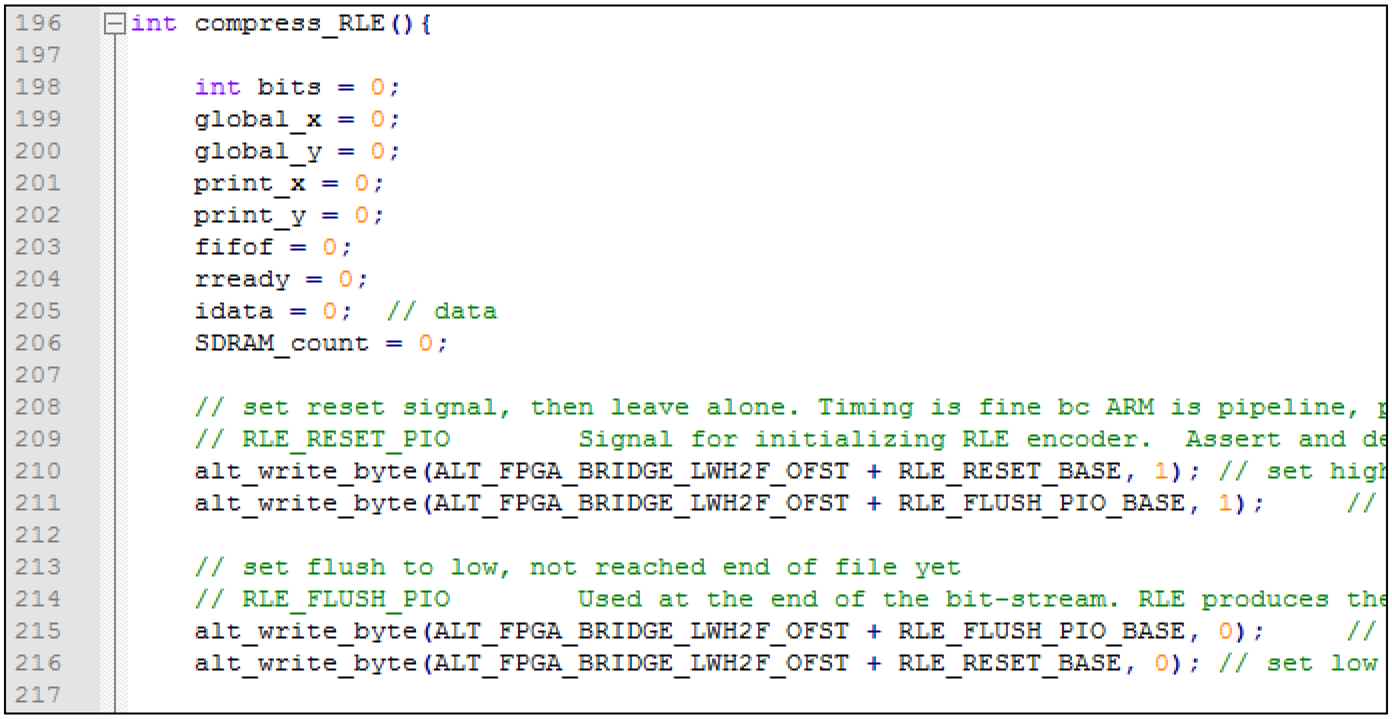
From there, we tried to understand how to go about communicating between the ARM and FPGA. We saw the methods that were given from the header files but we were not sure how to implement them. We asked TA Niket questions about implementation until we understood the diagram on the 5th slide of the powerpoint for this lab, which made it much easier to conceptualize the process. Once we understood that we were only writing the input and output code directly to/from the ARM code (and not the inputs between the FIFOs and the RLE), we were able to code the compression system (compression_RLE Figure 2) so that we could send data to the FIFO/RLE until we ran out of data (or the buffer was full, which never happened because our image is so small). We also concurrently checked the output from the FIFO/RLE until we knew we received all the data, which we computed by counting up the number of bits in the output files until it equaled the total amount of pixels on the screen. We saved each output to the SDRAM as we went along and counted how many outputs we stored to use for computing the compression ratio. This was the basis of our compression algorithm.

We used two incrementing methods to keep track of where we have read from / printed to the screen (increment_global_xy for reading the image and increment_print_xy for printing the decompressed image Figure 3). To show the difference between the captured black and white image and our output, we turned the screen black in between compression and decompression (black_screen Figure 4) To decompress (decompress_RLE Figure 5), we read each output from the SDRAM until we reached the total amount of SDRAM stored. For each output, we check if it is black or white and print to the screen the specified number of black or white pixels. When this is complete, we print the compression ratio to the console, which is the total bytes that make up the pixels on the screen divided by the bytes retrieved from the output. See screenshots below for our specific code.

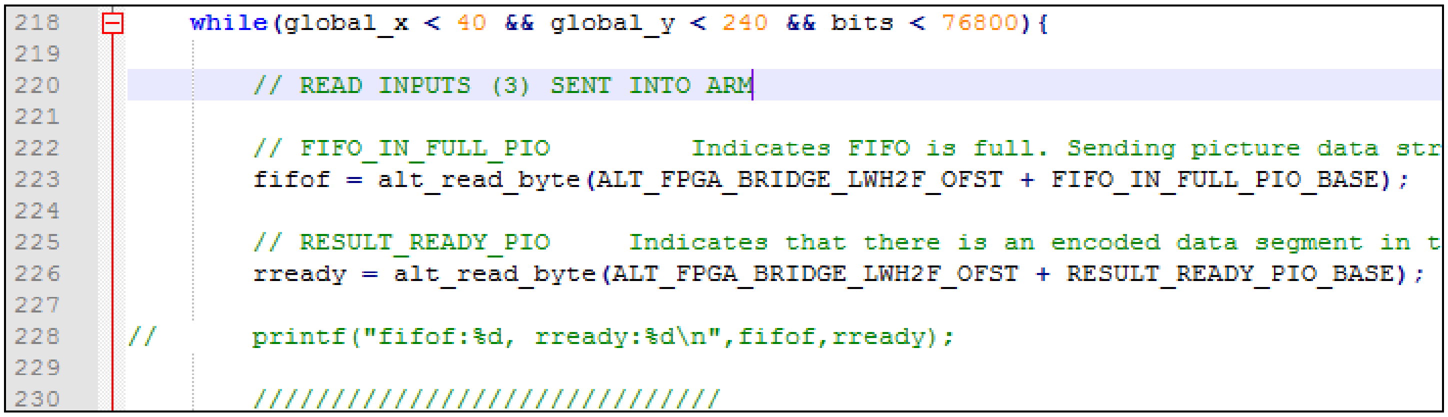


```c
int bw(){
    int x,y;
    short sum = 0;
    short black = 0x0000; // BLACK IS 1
    short white = 0xFFFF;
    short* base_address = Video_Mem_ptr;
    int shift_count = 0;        // when shift count is 8, clear and send

    for (y = 0; y < 240; y++) {
        for (x = 0; x < 320; x++) {
            sum = (sum + *(base_address + (y << 9) + x )) / 2;
        }
    }
    for (y = 0; y < 240; y++) {
        for (x = 0; x < 320; x++) {
            short value =  *(base_address + (y << 9) + x );

            if (value<sum){
                *(base_address + (y << 9) + x ) = white;
                picture_array[x/8][y] &= ~(1 << x%8);   // store 0

            }
            else{
                *(base_address + (y << 9) + x ) = black;
                picture_array[x/8][y] |= (1 << x%8);    // store 1
            }
        }
    }

    return 0;
}   // end of bw
```

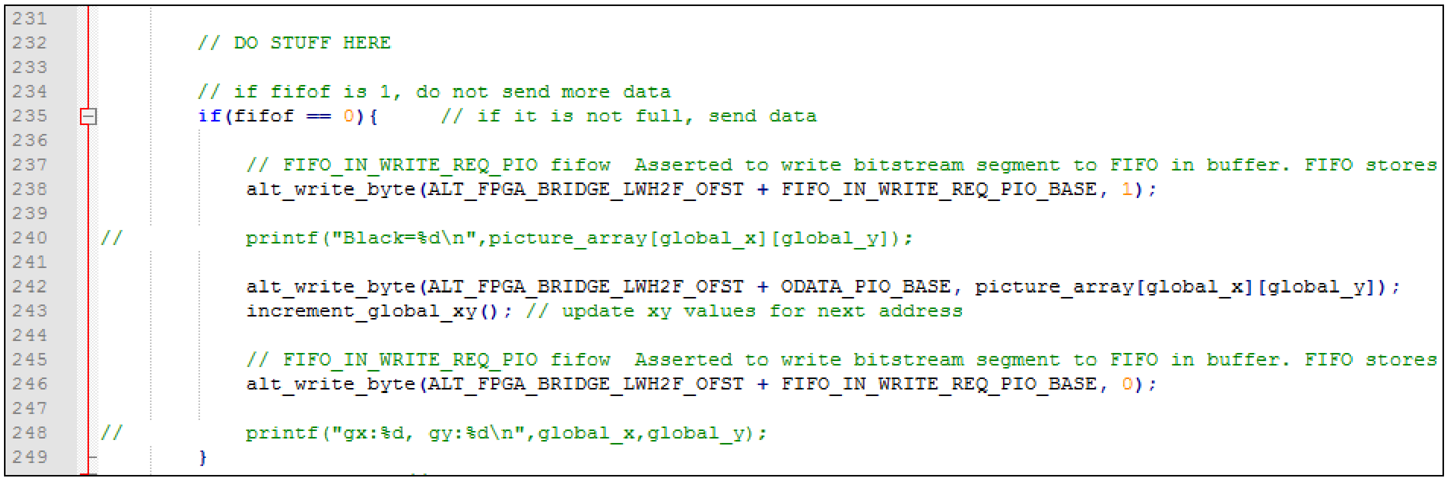
Figure 1: Method to convert captured image to black and white image. It also saves the pixel data into a double char one bit per pixel array called picture_array sorted for use with the RLE.



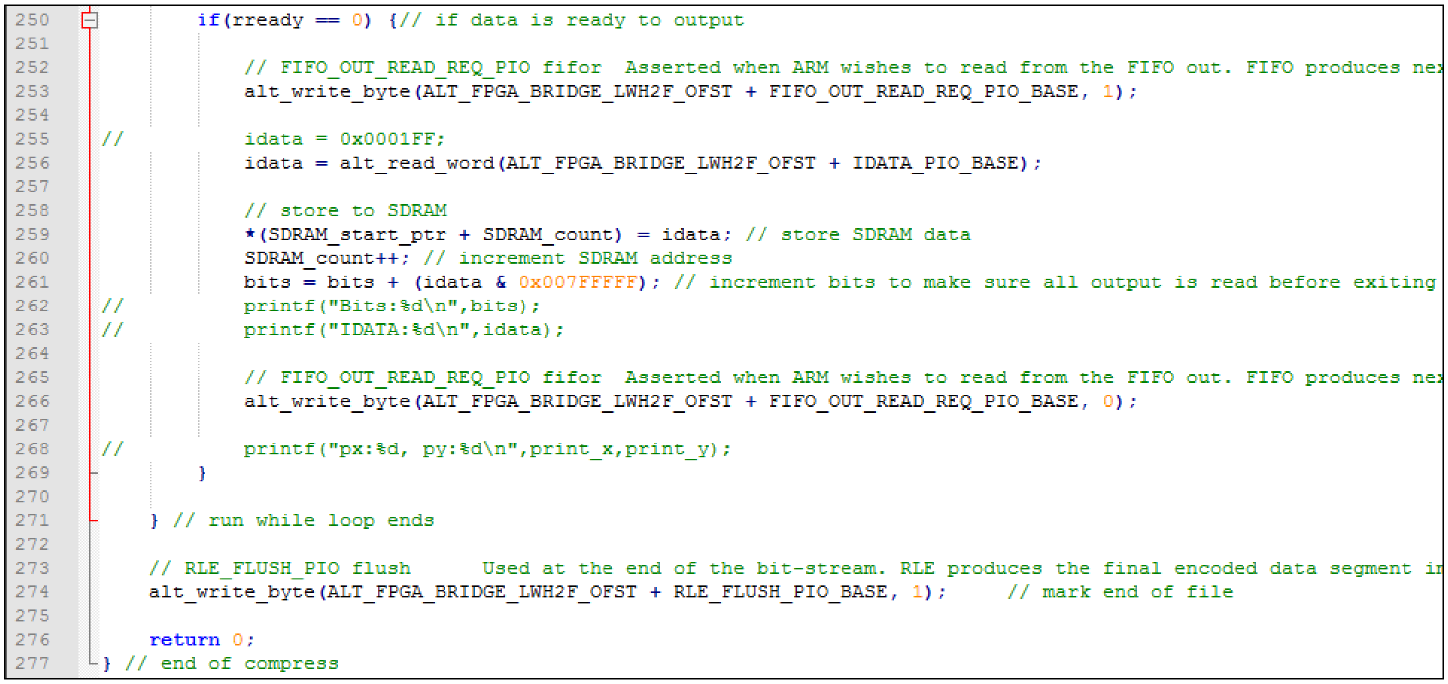
```c
int compress_RLE(){

    int bits = 0;
    global_x = 0;
    global_y = 0;
    print_x = 0;
    print_y = 0;
    fifof = 0;
    rready = 0;
    idata = 0;  // data
    SDRAM_count = 0;

    // set reset signal, then leave alone. Timing is fine bc ARM is pipeline, p
    // RLE_RESET_PIO        Signal for initializing RLE encoder.  Assert and de
    alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RLE_RESET_BASE, 1); // set hig
    alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RLE_FLUSH_PIO_BASE, 1);     //

    // set flush to low, not reached end of file yet
    // RLE_FLUSH_PIO        Used at the end of the bit-stream. RLE produces the
    alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RLE_FLUSH_PIO_BASE, 0);     //
    alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RLE_RESET_BASE, 0); // set low

```



```c
    while(global_x < 40 && global_y < 240 && bits < 76800){

        // READ INPUTS (3) SENT INTO ARM

        // FIFO_IN_FULL_PIO         Indicates FIFO is full. Sending picture data str
        fifof = alt_read_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + FIFO_IN_FULL_PIO_BASE);

        // RESULT_READY_PIO     Indicates that there is an encoded data segment in t
        rready = alt_read_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RESULT_READY_PIO_BASE);

//      printf("fifof:%d, rready:%d\n",fifof,rready);

        //////////////////////////////
```

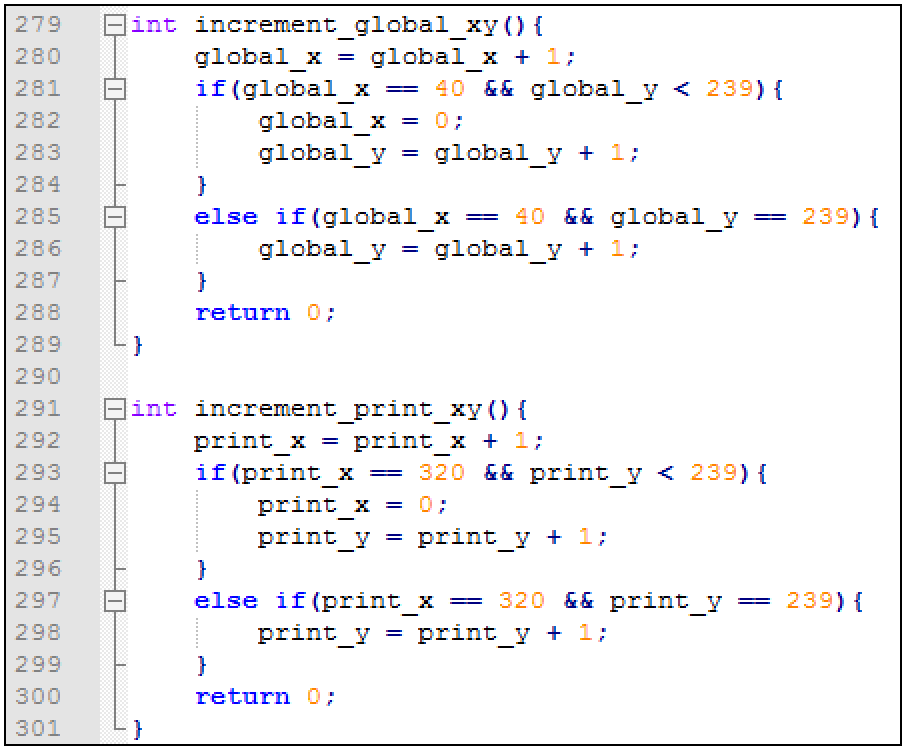


```c

        // DO STUFF HERE

        // if fifof is 1, do not send more data
        if(fifof == 0){     // if it is not full, send data

            // FIFO_IN_WRITE_REQ_PIO fifow  Asserted to write bitstream segment to FIFO in buffer. FIFO stores
            alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + FIFO_IN_WRITE_REQ_PIO_BASE, 1);

//          printf("Black=%d\n",picture_array[global_x][global_y]);

            alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + ODATA_PIO_BASE, picture_array[global_x][global_y]);
            increment_global_xy(); // update xy values for next address

            // FIFO_IN_WRITE_REQ_PIO fifow  Asserted to write bitstream segment to FIFO in buffer. FIFO stores
            alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + FIFO_IN_WRITE_REQ_PIO_BASE, 0);

//          printf("gx:%d, gy:%d\n",global_x,global_y);
        }
```



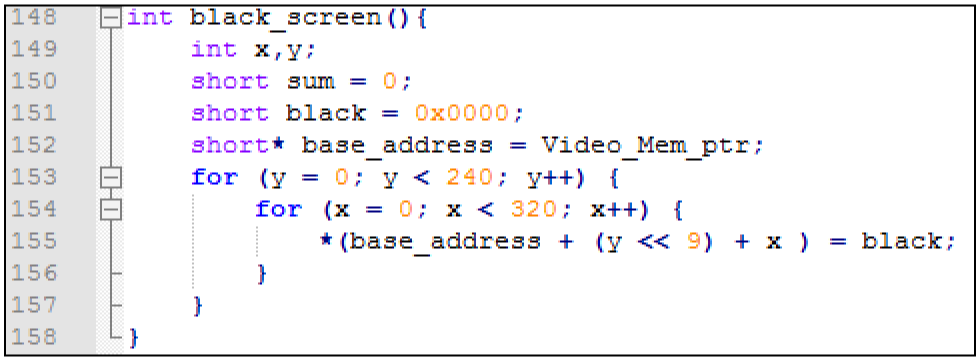
```c
        if(rready == 0) {// if data is ready to output

            // FIFO_OUT_READ_REQ_PIO fifor  Asserted when ARM wishes to read from the FIFO out. FIFO produces nex
            alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + FIFO_OUT_READ_REQ_PIO_BASE, 1);

//          idata = 0x0001FF;
            idata = alt_read_word(ALT_FPGA_BRIDGE_LWH2F_OFST + IDATA_PIO_BASE);

            // store to SDRAM
            *(SDRAM_start_ptr + SDRAM_count) = idata; // store SDRAM data
            SDRAM_count++; // increment SDRAM address
            bits = bits + (idata & 0x007FFFFF); // increment bits to make sure all output is read before exiting
//          printf("Bits:%d\n",bits);
//          printf("IDATA:%d\n",idata);

            // FIFO_OUT_READ_REQ_PIO fifor  Asserted when ARM wishes to read from the FIFO out. FIFO produces nex
            alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + FIFO_OUT_READ_REQ_PIO_BASE, 0);

//          printf("px:%d, py:%d\n",print_x,print_y);
        }

    } // run while loop ends

    // RLE_FLUSH_PIO flush      Used at the end of the bit-stream. RLE produces the final encoded data segment in
    alt_write_byte(ALT_FPGA_BRIDGE_LWH2F_OFST + RLE_FLUSH_PIO_BASE, 1);     // mark end of file

    return 0;
} // end of compress
```

Figure 2: Method to compress picture data from picture_array and send to RLE. Method also checked for data from RLE and saved it to SDRAM. Method runs until all data is sent from picture_array and all data is received from RLE and stored in SDRAM. It also send necessary output signals to the FIFO/RLE.



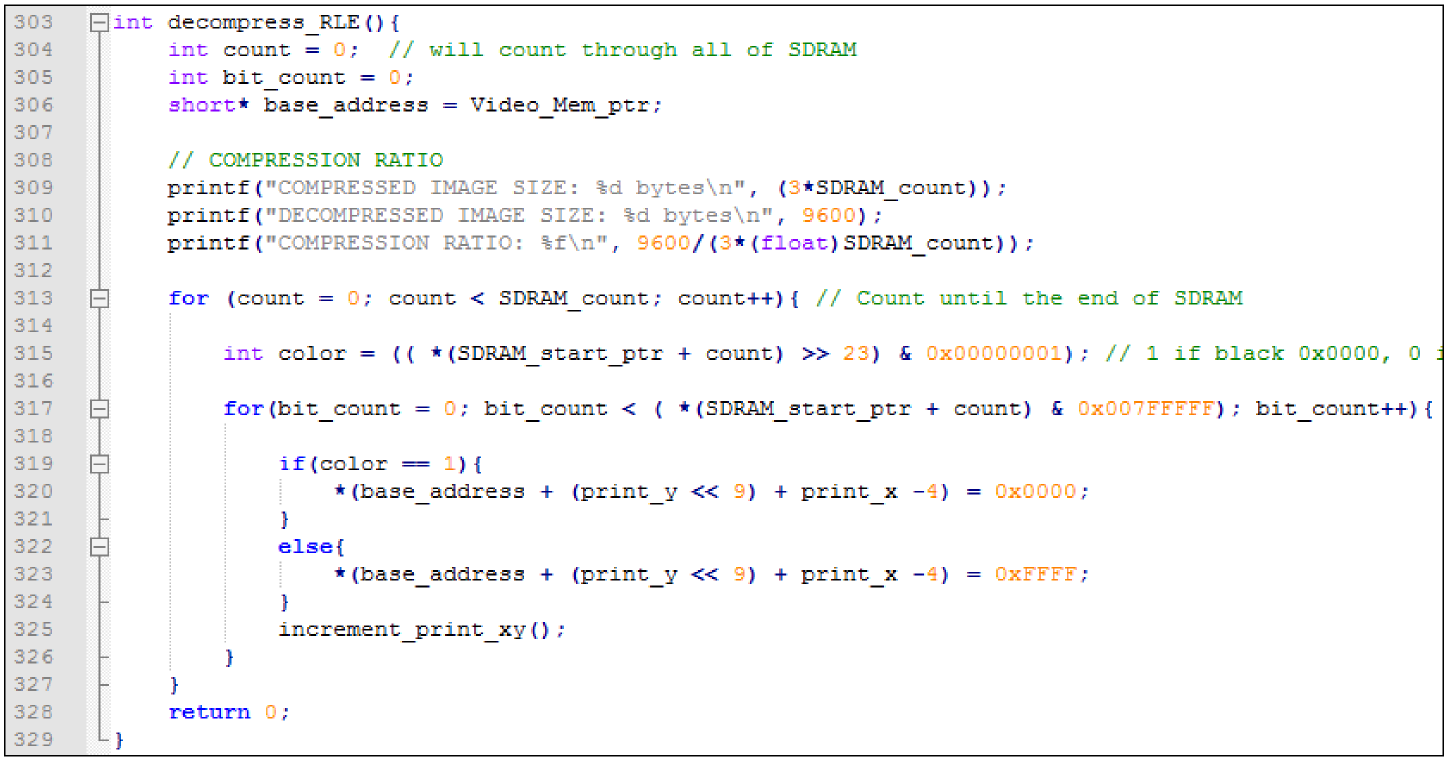
```c
int increment_global_xy(){
    global_x = global_x + 1;
    if(global_x == 40 && global_y < 239){
        global_x = 0;
        global_y = global_y + 1;
    }
    else if(global_x == 40 && global_y == 239){
        global_y = global_y + 1;
    }
    return 0;
}

int increment_print_xy(){
    print_x = print_x + 1;
    if(print_x == 320 && print_y < 239){
        print_x = 0;
        print_y = print_y + 1;
    }
    else if(print_x == 320 && print_y == 239){
        print_y = print_y + 1;
    }
    return 0;
}
```

Figure 3: Methods to handle and keep track of reading and writing to screen. All variables used are global variables. Increment_global_xy is used for reading and increment_print_xy is used for printing. Because the picture_array is stored in a char array, it retrieves one byte at a time, which is why global_x goes up to 40. The other method print_x goes to 320 because we print each pixel individually to the screen.



```c
int black_screen(){
    int x,y;
    short sum = 0;
    short black = 0x0000;
    short* base_address = Video_Mem_ptr;
    for (y = 0; y < 240; y++) {
        for (x = 0; x < 320; x++) {
            *(base_address + (y << 9) + x ) = black;
        }
    }
}
```

Figure 4: Method to make the screen black in between displaying the original and decompressed photo.



```c
int decompress_RLE(){
    int count = 0;  // will count through all of SDRAM
    int bit_count = 0;
    short* base_address = Video_Mem_ptr;

    // COMPRESSION RATIO
    printf("COMPRESSED IMAGE SIZE: %d bytes\n", (3*SDRAM_count));
    printf("DECOMPRESSED IMAGE SIZE: %d bytes\n", 9600);
    printf("COMPRESSION RATIO: %f\n", 9600/(3*(float)SDRAM_count));

    for (count = 0; count < SDRAM_count; count++){ // Count until the end of SDRAM

        int color = (( *(SDRAM_start_ptr + count) >> 23) & 0x00000001); // 1 if black 0x0000, 0 i

        for(bit_count = 0; bit_count < ( *(SDRAM_start_ptr + count) & 0x007FFFFF); bit_count++){

            if(color == 1){
                *(base_address + (print_y << 9) + print_x -4) = 0x0000;
            }
            else{
                *(base_address + (print_y << 9) + print_x -4) = 0xFFFF;
            }
            increment_print_xy();
        }
    }
    return 0;
}
```

Figure 5: Method to handle decompressed data from SDRAM and print to the screen properly. Outer loop runs each output stored in SDRAM, while the inner loop prints out the number and type of pixel stored in the SDRAM data to the screen. This method also prints the compression ratio.

5. Problems Encountered and Solutions:

One of the main problems that we encountered was that many of the signals from the RLE were not producing values that we expected. Our IDATA_PIO always returned zero and our RESULT_READY_PIO always returned 1 (when it should've been 0). To try to fix this, we redid the hardware several times, which did not fix the problem and took us many hours over the course of several days to do due to the compilation time. Ultimately, we went through the hardware setup with TA Sachin from scratch to verify that we were following all the directions properly, to which the hardware still did not work properly so we were given a sof file generated by TA Sachin to use. Once we were able to get working hardware, the software quickly fell into place.

We also did not realize that we needed to save the output data to the SDRAM until towards the end of the project. This change did not take super long to implement but it was a bit confusing at first because we did not use SDRAM explicitly in any of the previous labs. We at first set the pointer for SDRAM to be a char on accident. We changed the pointer type to be an int so that we would be able to store and access each 24 bit output segment separately.

During the demo, it was brought to our attention that our code for counting how many bytes we saved to SDRAM (SDRAM_count) would not work properly when we took several pictures in a row. We tried to find the source of the error, to which we noticed that some of the print statements seemed to execute multiple times, potentially meaning that the switch's analog position was causing the code to run several times and mess up the values of SDRAM_count. We tried moving SDRAM_count to different places in the code and tried setting it to 0 in strategic locations but we could not figure out the issue. With more time we might have been able to catch it, but we were more or less satisfied with this project despite the small error.

6. Results and Conclusion:

Overall, our project was successful. We succeeded in understanding the core concepts of the lab, as we demonstrated in the demo. We enjoyed putting together the other three projects and seeing how they all combine to make one functional system, which was rewarding. We feel that we have a much better understanding of how traditional processors and FPGAs can work together and what each one's strengths and weaknesses are. For example, anything in C code will take several clock cycles to execute due to its pipeline, while verilog code will execute each clock cycle, implementing things much faster than C code. However, hardware can be trickier to debug than C code, as we had a hard time catching hardware errors due to the complexity of the FPGA software QUARTUS / QSYS. We also were very happy that we were able to overcome most of the problems we encountered along the way, however we wish we had caught the SDRAM_count error before the demo.