Group 14C: J. Lagasse, A. Sjogren, O. Onyenokwe, L. Wu
ECE 354: Lab 2 Report
3/7/2017

1. Introduction

In lab 2, our goal was to manipulate an image captured on a camera mounted on a DE1-Soc board. By the end of the lab we were able to understand the procedure involved in interfacing peripherals to an embedded system and the process of capturing image through a camera. We also learned to develop C programs for the ARM hard core processor using the Altera Monitor Program and researched basic image processing concepts. We also learned about how images are displayed on a monitor. Different functions were performed on a captured image including mirroring, rotation, grayscale, color inversion, and adding text such as a timestamp and photo counter.

2. Detailed Procedure

1. Our first objective was to research the components of the board, read their respective manuals, and review the electronics setup provided to us. We did all of these things but we did not go into as much detail as we could have, as we moved on to the next steps before fully understanding every detail of the hardware components.
2. We then familiarised ourselves with the Altera Monitor Program, which will be used to compile and load our C code onto our board. We practiced connecting the camera and setting up the monitor as well.
3. After we understood the physical setup, we began to examine the C code provided to us. We asked the TAs to explain some of the areas that we were confused about, and then read the DE1-SOC manual to find out how to properly code methods to manipulate images taken by the camera. From the manual, we learned about the pixel buffer for the location and color of pixels, and the character buffer to print text on top of the pictures.
4. With this knowledge, we then began to change the code C code provided to us to perform the following functions (along with the switch number used to implement each function):
    a. Take a picture (SW3): This function writes a command into the control register to stop the video feed, thereby freeing the frame and setting the data in the frame into the pixel buffer for us to use in manipulations. All other functions rely on this function working properly.
    b. Mirror (SW4): This function mirrors an image vertically thereby mirroring it across the y axis. We traverse the left half of the pixels of the image and swap pixels equidistant to the center of the photo along the y axis with each other to create the mirroring effect.
    c. Rotate (SW5): The rotate function is essentially the mirror function with an additional flip across the center of the photo along the x axis. First we run the loop described in the mirroring function. Then it runs another loop that traverses the top half of the image and swaps pixels that are equidistant to the center of the photo along the x axis. The result is a full 180 degree rotation.

d. Timestamp (SW6): This function takes a photo and then prints the date and time that the image was taken using the character buffer on top of the image.

e. Counter (SW7): The counter function simply counts the number of pictures taken. We increment this counter every time we call a function.

f. Grayscale (SW8): Grayscale converts our color photograph into a grayscale image by taking the red, blue, and green RGB value separately, computing the average, and replacing the RGB value with the average for each pixel in the photo. This works because the greater the average, the greater the darkness of the image. When all three RGB components of a color are equal, the color is considered to us to be either white, black, or a shade of gray.

g. Inverter (SW9): This function inverts the colors from grayscale by performing bitwise NOT on each pixel in the photograph.

h. Bonus (SW0): We did not have enough time to complete the two bonus methods, edge finder and difference finder. But, due to a misunderstanding, we did end up developing a method of our own that used grayscale, a gaussian filter, and color inversion in that order to create an effect similar to a pencil drawing.

3. Software

Our code contains specific methods to perform each of the functionalities outlined in the procedure. We will include a snapshot of each method of our code complete with a description underneath.

➔ Main Method

```c
int main(void){                                     // main method

    // initialize variables
    volatile int * KEY_ptr = (int *) KEY_BASE;
    volatile int * SW_ptr = (int *) SW_BASE;
    volatile int * Video_In_DMA_ptr = (int *) VIDEO_IN_BASE;
    volatile short * Video_Mem_ptr  = (short *) FPGA_ONCHIP_BASE;
    volatile short * Char_ptr    = (short *) CHARACTER_BASE;

    *(Video_In_DMA_ptr + 3) = 0x4;                  // Enable the video

    while (1){                                      // while loop

        switch(*SW_ptr){

            // TAKE PICTURE
            case 14:                                // Switch 3,2,1 high (Toggle 3)
                picture_counter++;                  // increment picture counter by 1
                clear_char(Char_ptr);               // clear text
                take_picture(Video_In_DMA_ptr);     // take photo
                while(*SW_ptr == 14){}              // While toggle 3 is high
                resume_video(Video_In_DMA_ptr);
                break;

            // MIRROR
            case 22:                                // Switch 4,2,1 high (Toggle 4)
                picture_counter++;                  // increment picture counter by 1
                clear_char(Char_ptr);               // clear text
                take_picture(Video_In_DMA_ptr);     // take photo
                mirror_horizontal(Video_Mem_ptr);
                while(*SW_ptr == 22){}              // While toggle 4 is high
                resume_video(Video_In_DMA_ptr);
                break;
```

Our main method calls individual functions using a case and switch statement. Each case is a different switch combination. For each switch flip, we increment the picture counter for the Count method, then clear the character buffer so that the menu goes away. We then take a picture by freezing the frame, and then call whatever method we need to perform. In the photo above, we are calling mirror_horizontal to perform the Mirror function. Then we use a while loop to hold the effect on the screen until the switch is flipped again, then we resume the video and then break the case statement.

- ➔ **Write Menu**

```
int print_menu(volatile short * Char_ptr){          // prints menu options on video feed
    int x1 = 8;                                     // set coordinates
    int x2 = 8;
    int x3 = 9;
    int x4 = 9;
    int x5 = 9;
    int x6 = 9;
    int y1 = 10;
    int y2 = 11;
    int y3 = 13;
    int y4 = 14;
    int y5 = 15;
    int y6 = 16;
    int offset1, offset2, offset3, offset4, offset5, offset6; // set offset vars
    char *line1_ptr = "Welcome to Group 14C Demo!"; // set menu text
    char *line2_ptr = "Flip switches to operate. ";
    char *line3_ptr = "3 Take picture, 4 Mirror  ";
    char *line4_ptr = "5 Rotate, 6 Timestamp     ";
    char *line5_ptr = "7 Counter, 8 Grayscale    ";
    char *line6_ptr = "9 Invert, 0 Bonus         ";
    /* Display a null-terminated text string at coordinates x, y. Assume that the text fits on one line */
    offset1 = (y1 << 7) + x1;
    offset2 = (y2 << 7) + x2;
    offset3 = (y3 << 7) + x3;
    offset4 = (y4 << 7) + x4;
    offset5 = (y5 << 7) + x5;
    offset6 = (y6 << 7) + x6;
    while ( *(line1_ptr) ){                         // print menu text
        *(Char_ptr + offset1) = *(line1_ptr);       // write to the character buffer line 1
        ++line1_ptr;
        ++offset1;
```

The print_menu method uses the code described in the DE1-Soc manual to print several hard coded statements to the screen. The int x and y are the coordinates for the on screen coordinates for each line of text. Each char line_ptr holds the actual string value to be printed. Then the offset is calculated, and the while loop prints out each character in each char line_ptr until they reach the end of the strings.

➔ Clear Screen

```c
int clear_char(volatile short * Char_ptr){          // clears all characters from screen
    int x=0;
    int y=0;
    int offset;
    char *clear_char_ptr = " ";
    for (y = 0; y < 59; y++) {
        for (x = 0; x < 79; x++) {
            offset = (y << 7) + x;
            *(Char_ptr+ offset) = *(clear_char_ptr); // clear
        }
    }
    return 0;
}
```

The clear_char method works the same way as the print_menu method, except it prints out a blank character to all positions in the character buffer, thereby removing all characters from the whole screen.

➔ Take a Picture

```c
int take_picture(volatile int * Video_In_DMA_ptr){  // disables the video to capture one frame
    *(Video_In_DMA_ptr + 3) = 0x0;                  // freeze frame
    return 0;
}

int resume_video(volatile int * Video_In_DMA_ptr){  // enables the video, disbales frame capture
    *(Video_In_DMA_ptr + 3) = 0x4;                  // unfreezes frame
    return 0;
}
```

The function take_picture writes into the control register to freeze the frames. The function resume_video also writes into the control register to unfreeze the frames. Video_In_DMA_ptr + 3 is the location of the DMA control register.

➔ Mirror

```c
int mirror_horizontal(volatile int * Video_Mem_ptr){// "mirror" in demo, mirror horizontally
    int x,y;
    short* base_address = Video_Mem_ptr;
    for (y = 0; y < 240; y++) {
        for (x = 0; x < 159; x++) {
            long temp1 = *(base_address + (y << 9) + x );
            long temp2 = *(base_address + (y << 9) + (319-x));
            *(base_address + (y << 9) + x ) = temp2;
            *(base_address + (y << 9) + (319-x)) = temp1;
        }
    }
    return 0;
}
```

The mirror_horizontal function uses the two for loops to traverse the left half of the on-screen image. The picture is 240x320 pixels, therefore making the x axis loop stop at 158 makes it traverse half the image. Then, we create two long variables that hold the pixel buffer data of the point we are on and the point we would like to mirror. Then we rewrite the pixel buffer with the data of the opposite points. Note that the y location needs to be shifted over 9 bits to calculate the right address, and that 319-x computes the pixel on the other side of the center axis.

➔ Rotate

```c
int mirror_vertical(volatile int * Video_Mem_ptr){  // "rotate" in demo, mirror vertically or flip 180 deg
    int x,y;
    short* base_address = Video_Mem_ptr;
    for (y = 0; y < 240; y++) {
        for (x = 0; x < 159; x++) {
            long temp1 = *(base_address + (y << 9) + x );
            long temp2 = *(base_address + (y << 9) + (319-x));
            *(base_address + (y << 9) + x ) = temp2;
            *(base_address + (y << 9) + (319-x)) = temp1;
        }
    }
    for (y = 0; y < 119; y++) {
        for (x = 0; x < 320; x++) {
            long temp3 = *(base_address + (y << 9) + x );
            long temp4 = *(base_address + ((239 - y) << 9) + x );
            *(base_address + (y << 9) + x ) = temp4;
            *(base_address + ((239 - y) << 9) + x ) = temp3;
        }
    }
    return 0;
}
```

The mirror_vertical method is essentially two instances of the mirror method, where the first double for loop is identical to the rotate method and the second for loop is mirroring but for the opposite direction. The bottom method traverses for all of the x axis but only traverses the top half of pixels as shown by it stopping at 118 pixels. The value 239-y computes the pixel on the other side of the center axis, which is then shifted by 9 to get the correct address value.

➔ Timestamp

```c
265 int timestamp(volatile short * Char_ptr){          // print timestamp
266     time_t t = time(NULL);
267     struct tm tm = *localtime(&t);
268     tm.tm_hour = tm.tm_hour - 5;
269     int x = 8;
270     int y = 13;
271     int offset;
272     char *timestamp_ptr = asctime(&tm);
273     offset = (y << 7) + x;
274     while ( *(timestamp_ptr) ){
275         *(Char_ptr + offset) = *(timestamp_ptr);
276         ++timestamp_ptr;
277         ++offset;
278     }
279     return 0;
280 }
```

The timestamp function uses time_t to fetch the current time from the computer. Then we use line 268 to subtract five hours from the time, therefore converting the time from GST to our time zone. The rest of the method is the character buffer code explained in the print_menu method.

➔ Counter

```c
int display_counter(volatile short * Char_ptr, int picture_counter){    // display the value stored in the counte
    // try spritf to get rid of null pointer extra character
    int x = 15;
    int y = 13;
    int num_offset;
    char array[16];
    snprintf(array, sizeof(array), "Count: %i\n", picture_counter);
    char *num_ptr = &array;
    num_offset = (y << 7) + x;
    while ( *(num_ptr) ){
        *(Char_ptr + num_offset) = *(num_ptr);
        ++num_ptr;
        ++num_offset;
    }
    return 0;
}
```

The display_counter method again works like the print_menu method, except it only prints one line, which is the text "Count:" and the current value stored in the variable picture_counter. The variable picture_counter starts at zero at the beginning of the program and is then incremented by one every time we perform a function. We use the C method snprintf to convert the int variable picture_counter to a string so that the code can print correctly.

➔ Grayscale

```c
299 int grayscale(volatile int * Video_Mem_ptr){        // grayscale
300     int x,y;
301     short* base_address = Video_Mem_ptr;
302     for (y = 0; y < 240; y++) {
303         for (x = 0; x < 320; x++) {
304             long pixel_ptr = *(base_address + (y << 9) + x );
305             unsigned int blue = ( pixel_ptr & 0x1F );
306             unsigned int green = ( ( pixel_ptr >> 6 ) & 0x1F );
307             unsigned int red = ( ( pixel_ptr >> 11 ) & 0x1F );
308             unsigned int average = (blue+red+green)/3;
309             *(base_address + (y << 9) + x ) = average + (average<<6) + (average<<11);
310         }
311     }
312     return 0;
313 }
```

The grayscale function traverses each pixel in the image using the for loops, takes the RGB data in it (line 304), then separates the value for blue, red, and green by bitwise ANDing the 5 bits for each color (shifted out from their location in the data register). Then we add all these values together, take the average, and set each RGB value to the average, therefore creating only white, gray, or black pixels

➔ Inverter

```
315 int invert(volatile int * Video_Mem_ptr){          // inverts colors from grayscale
316     int x,y;
317     short* base_address = Video_Mem_ptr;
318     for (y = 0; y < 240; y++) {
319         for (x = 0; x < 320; x++) {
320             long pixel_ptr = *(base_address + (y << 9) + x );
321             unsigned int blue = ( pixel_ptr & 0x1F );
322             unsigned int green = ( ( pixel_ptr >> 6 ) & 0x1F );
323             unsigned int red = ( ( pixel_ptr >> 11 ) & 0x1F );
324             unsigned int average = (blue+red+green)/3;
325             average = ~average;
326             *(base_address + (y << 9) + x ) = average + (average<<6) + (average<<11);
327         }
328     }
329     return 0;
330 }
```

The invert function is identical to the grayscale function except for line 325, where the average value has the bitwise NOT performed on it to get the complementary value.

➔ Bonus

```
332 int blur_filter(volatile int * Video_Mem_ptr){          // gaussian filter to blur
333     int x,y;
334     double blur_corner = 0.01;  // gaussian 0.01
335     double blur_edge = 0.08;    // gaussian 0.08
336     double blue_center = 0.64;  // gaussian 0.64
337     short* base_address = Video_Mem_ptr;
338     for (y = 1; y < 239; y++) {
339         for (x = 1; x < 319; x++) {
340 
341             // row 1
342             long r1c1 = *(base_address + ((y-1) << 9) + (x-1) );
343             long r1c2 = *(base_address + ((y-1) << 9) + x );
344             long r1c3 = *(base_address + ((y-1) << 9) + (x+1) );
345             // row 2
346             long r2c1 = *(base_address + (y << 9) + (x-1) );
347             long r2c2 = *(base_address + (y << 9) + x );
348             long r2c3 = *(base_address + (y << 9) + (x+1) );
349             // row 3
350             long r3c1 = *(base_address + ((y+1) << 9) + (x-1) );
351             long r3c2 = *(base_address + ((y+1) << 9) + x );
352             long r3c3 = *(base_address + ((y+1) << 9) + (x+1) );
353 
354             long blur = (int)(r1c1*blur_corner+r1c2*blur_edge+r1c3*blur_corner+r2c1*blur_edge+r2c2*blue_center+
355 
356             *(base_address + (y << 9) + x ) = blur;
357         }
358     }
359     return 0;
360 }
```

We obviously did not do the correct bonus method, however what we did do is a gaussian filter to blur an image out. This works by taking the value of the pixel we are looking at the the values of the 8 pixels surrounding it (lines 342-352). Then we multiply each value by a certain coefficient depending on its location and add the values together into a variable that we call blur. Then we set the value of the pixel we are currently on to the blur value. However this program has a fatal flaw where we immediately write back the values to the pixel buffer, which causes a chain reaction that causes the other pixels to be over blurred. This could've been solved by creating an array to store the the original image and and saving the resulting values to the pixel buffer.

```
int edge_det(volatile int * Video_Mem_ptr){         // gaussian filter to blur
    int x,y;
    short* base_address = Video_Mem_ptr;
    short array[319][239];
    for (y = 0; y < 240; y++) {
        for (x = 0; x < 320; x++) {
            array[x][y] = *(base_address + (y << 9) + x );
        }
    }

    bw(base_address);
    short hit_color = 0x6666;
    unsigned int cell[8];

    for (y = 1; y < 239; y++) {
        for (x = 1; x < 319; x++) {

            // row 1
            cell[0] = *(base_address + ((y-1) << 9) + (x-1) );
            cell[1] = *(base_address + ((y-1) << 9) + x );
            cell[2] = *(base_address + ((y-1) << 9) + (x+1) );
            // row 2
            cell[3] = *(base_address + (y << 9) + (x-1) );
            cell[4] = *(base_address + (y << 9) + x );
            cell[5] = *(base_address + (y << 9) + (x+1) );
            // row 3
            cell[6] = *(base_address + ((y+1) << 9) + (x-1) );
            cell[7] = *(base_address + ((y+1) << 9) + x );
            cell[8] = *(base_address + ((y+1) << 9) + (x+1) );

            if ( cell[0]!=cell[4] || cell[1]!=cell[4] || cell[2]!=cell[4] || cell[3]!=cell[4] || cell[5]!=cell[4]
            }
            else{
                array[x][y] = hit_color;
            }
        }
    }
    *(base_address + (y << 9) + x ) = array[1][1];

    for (y = 0; y < 240; y++) {
        for (x = 0; x < 320; x++) {
            *(base_address + (y << 9) + x ) = array[x][y];
        }
    }

    return 0;

}
```

This was our attempt at the edge finder bonus function. The idea was to save the original image to an array, turn the image in the pixel buffer black and white (we wrote a separate method for black and white), then use a technique similar to our gaussian blur tech to get the value of a pixel and the values of the eight surrounding pixels and check if they are all black or white. If all surrounding pixels match the current pixel, then we would save the value in the array for that pixel to be a certain color, hit_color. Then we traverse the image and set all data values in the pixel buffer to the values in the array. This in theory would work, but our program would freeze each time we tried to run it. We believe this is because this is a very memory intensive method, and this method caused our program to run out of memory in the FPGA. To fix this, we could've learned how to save the array to the SDRAM but we ran out of time to implement this fix.

4. Problems & Solutions

The first major problem that our group encountered occurred in step 2 of the procedure when we first attempted to set up the board, camera, and external monitor properly. The live video feed on the screen experienced color distortion was rotated at 180 degrees. This was not fixed by rerunning and re-programming the board. The solution to this problem involved reconnecting the camera and ensuring that the camera was connected by all pins, because we would accidentally insert the camera without attaching all the pins.

All of our other problems occurred in step 4 of the procedure as we were developing methods in the C program. We first started the project by creating a menu print out and working with the character buffer, in which we ran into problems where the character buffer was too small to print out all the text we wanted to have on each line. This required us to split up the lines of text into smaller segments and print them out separately. Our next task was printing the timestamp, where we did not know the best way to get the current time from the computer to print out. This required us to do some research on available C functions until we found a method that would retrieve the time. Unfortunately the time was printed in GST, which is not our time zone, so we had to subtract the time returned by 5 hours to get it into our time zone. Next we worked on the counter, where we had a pointer error that caused our counter to consistently return 32 regardless of how many pictures were taken. This was solved by using print statements to debug our code and fix our pointer mistake.

From there we began to manipulate the pixel buffer. The first method we implemented that required using the pixel buffer were the mirror and rotate methods. How to use the pixel buffer correctly stumped us for a while, as we got a bit confused with the pointers and which variable types to use. We figured out a working combination by debugging through print statements until we found out which variable type was the proper one. Once we figured this out, it was easy to code the logic of mirroring and rotating the pixels. We also were stumped by the pixel buffer for the color data for a while for the same reason, which we also solved through print statement debugging until we understood which combination worked successfully.

We made an attempt to do the bonus, however we ran out of time to complete them. We tried the edge detector in particular, and out main problem with this is that the FPGA would run out of memory and we did not learn how to allocate memory to the SDRAM in time. In general, one could consider our lack of programming efficiency in some areas as a problem, as our code is a bit longer and more complicated than it needed to be. For example, we should've created a method to print a particular string on the monitor screen instead of hard coding all lines of text that we wanted to print. This would've improved our efficiency and potentially could have solved our issue with the bonus question, as there might have been enough memory for our code to work without implementing SDRAM if our code was written more efficiently.

5. Conclusion

After completing this lab we are now able to understand how to capture and manipulate and images using an embedded system. Thanks to some research and TA Sachin's explanations, we also better understand the how the FPGA and ARM microprocessor work together to perform these functions. We were able to complete all functions required in this project, but were sadly unable to complete the bonus in time, even though we enjoyed the challenge. We will prioritize understanding the hardware over the software for the next lab, as we feel that that was an area we could've improved in. We will also try to be more efficient with our code in general.