Laboratory Exercise 8

Introduction to Graphics and Animation

The purpose of this exercise is to learn how to display images and perform animation. We will use the ARM A9 processor, in the DE1-SoC Computer. Graphics will be displayed on a VGA terminal by using the DE1-SoC Computer's video-out port. To do this exercise you need to know how to use C code with the ARM processor, and how to use the video-out port. For Part IV of the exercise you also need to know how to use C code with interrupts for the ARM processor. Recommended tutorials include *Introduction to the ARM Processor* and *Using the ARM Generic Interrupt Controller*. You should be familiar with the material in the DE1-SoC Computer documentation that pertain to the use of exceptions and interrupts with C code, in addition to the material about the video-out port.

Background Information

The DE1-SoC Computer includes a video-out port with a VGA controller that can be connected to a standard VGA monitor. The VGA controller supports a screen resolution of 640 × 480. The image that is displayed by the VGA controller is derived from two sources: a *pixel* buffer, and a *character* buffer. Only the pixel buffer will be used in this exercise, hence we will not discuss the character buffer.

Pixel Buffer

The pixel buffer for the video-out port holds the data (color) for each pixel that is displayed by the VGA controller. As illustrated in Figure 1, the pixel buffer provides an image resolution of 320 × 240 pixels, with the coordinate 0,0 being at the top-left corner of the image. Since the VGA controller supports the screen resolution of 640 × 480, each of the pixel values in the pixel buffer is replicated in both the *x* and *y* dimensions when it is being displayed on the VGA screen.

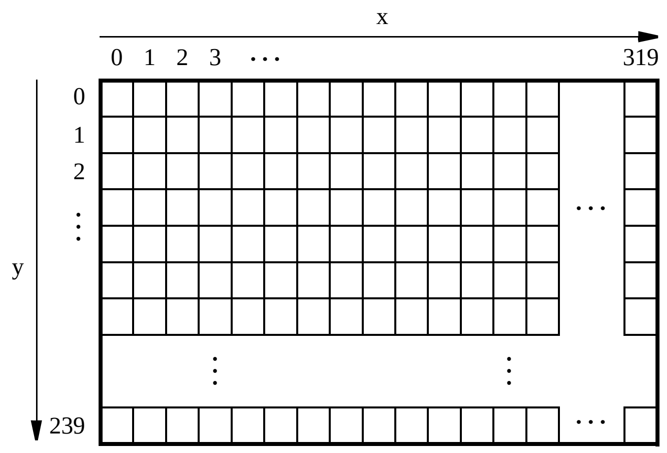
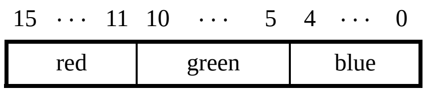
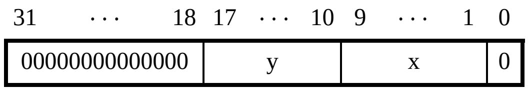


Figure 1: Pixel buffer coordinates.

Figure 2a shows that each pixel color is represented as a 16-bit halfword, with five bits for the blue and red components, and six bits for green. As depicted in part b of Figure 2, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE1-SoC Computer the default address of the pixel buffer is 0xC8000000, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at location 0,0 has the address 0xC8000000, the pixel 1,0 has the address *base* + $(00000000\ 000000001\ 0)_2$ = 0xC8000002, the pixel 0,1 has the address *base* + $(00000001\ 000000000\ 0)_2$ = 0xC8000400, and the pixel at location 319,239 has the address *base* + $(11101111\ 100111111\ 0)_2$ = 0xC803BE7E.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 2: Pixel values and addresses.

You can create an image by writing color values into the pixel addresses as described above. A dedicated *pixel buffer controller* reads this pixel data from the memory and sends it to the VGA display. The controller reads the pixel data in sequential order, starting with the pixel data corresponding to the upper-left corner of the VGA screen and proceeding to read the whole buffer until it reaches the data for the lower-right corner. This process is then repeated, continuously. You can modify the pixel data at any time, by writing to the pixel addresses. Writes to the pixel buffer are automatically interleaved in the hardware with the read operations that are performed by the pixel buffer controller.

It is also possible to prepare a new image for the VGA display without changing the content of the pixel buffer, by using the concept of *double-buffering*. In this scheme two pixel buffers are involved, called the *front* and *back* buffers, as described below.

Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the VGA screen. This pixel buffer controller includes a programming interface in the form of a set of registers, as illustrated in Figure 3. The register at address 0xFF203020 is called the *Buffer* register, and the register at address 0xFF203024 is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the VGA screen. As mentioned above, in the default configuration of the DE1-SoC Computer this Buffer register is set to the address 0xC8000000, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also 0xC8000000, which means that there is only one pixel buffer. But software can modify the address stored in the Backbuffer register, thereby creating a second pixel buffer. An image can be drawn into this second buffer by writing to its pixel addresses. This image is not displayed on the VGA monitor until a pixel buffer *swap* is performed, as explained below.

A pixel buffer swap is caused by writing the value 1 to the Buffer register. This write operation does not directly modify the content of the Buffer register, but instead causes the contents of the Buffer and Backbuffer registers to be swapped. The swap operation does not happen right away; it occurs at the end of a VGA screen-drawing cycle, after the last pixel in the bottom-right corner has been displayed. This time instance is referred to as the *vertical synchronization* time, and occurs every 1/60 seconds. Software can poll the value of the S bit in the *Status* register, at address 0xFF20302C, to see when the vertical synchronization has happened. Writing the value 1 into the Buffer register causes S to be set to 1. Then, when the swap of the Buffer and Backbuffer registers has been completed S is reset back to 0. The *Status* register contains additional bits of information, shown in Figure 3, but these bits are not needed for this exercise. Also, the programming interface includes a *Resolution* register, shown in the figure, that contains the X and Y resolution of the pixel buffer(s).

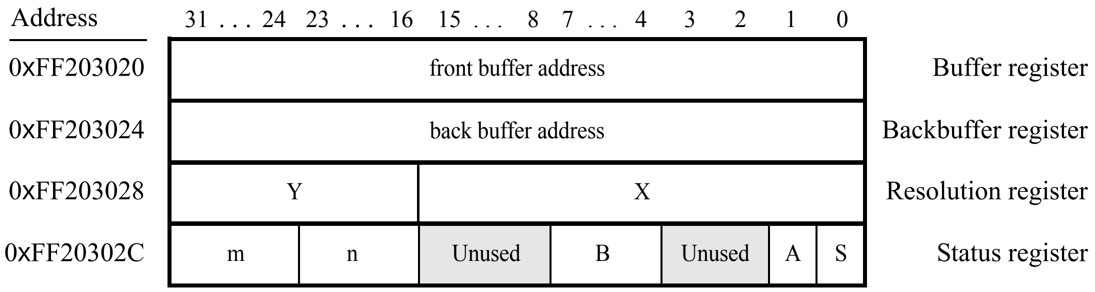


Figure 3: Pixel buffer controller registers.

In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, a pixel buffer swap is performed. Then, the pixel buffer that is now pointed to by the Backbuffer register, which was already displayed, is cleared and the next new image is drawn. In this way, the next image to be displayed is always drawn into the "back" pixel buffer, and the "front" and "back" buffer pointers are swapped when the new image is ready to be displayed. Each time a swap is performed software has to synchronize with the VGA controller by waiting until the S bit in the Status register becomes 0.

Part I

In this part you will learn how to implement a simple line-drawing algorithm.

Drawing a line on a screen requires coloring pixels between two points (x_1, y_1) and (x_2, y_2), such that the pixels represent the desired line as closely as possible. Consider the example in Figure 4, where we want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares in the figure represent the location and size of pixels on the screen. As indicated in the figure, we cannot draw the line precisely—we can only draw a shape that is similar to the line by coloring the pixels that fall closest to the line's ideal location on the screen.

We can use algebra to determine which pixels to color. This is done by using the end points and the slope of the line. The slope of our example line is $slope = (y_2 - y_1)/(x_2 - x_1) = 4/11$. Starting at point $(1, 1)$ we move along the x axis and compute the y coordinate for the line as follows:

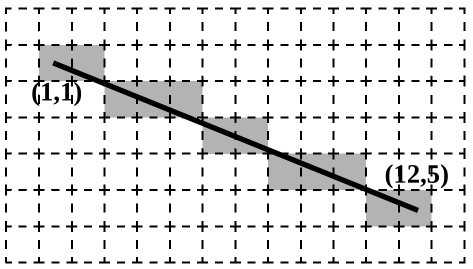


Figure 4: Drawing a line between points $(1, 1)$ and $(12, 5)$.

$$y = y_1 + slope \times (x - x_1)$$

Thus, for column $x = 2$, the y location of the pixel is $1+\frac{4}{11}\times(2-1) = 1\frac{4}{11}$. Since pixel locations are defined by integer values we round the y coordinate to the nearest integer, and determine that in column $x = 2$ we should color the pixel at $y = 1$. For column $x = 3$ we perform the calculation $y = 1 + \frac{4}{11} \times (3 - 1) = 1\frac{8}{11}$, and round the result to $y = 3$. Similarly, we perform such computations for each column between x_1 and x_2.

The approach of moving along the x axis has drawbacks when a line is steep. A steep line spans more rows than it does columns, and hence has a slope with absolute value greater than 1. In this case our calculations will not produce a smooth-looking line. Also, in the case of a vertical line we cannot use the slope to make a calculation. To address this problem, we can alter the algorithm to move along the y axis when a line is steep. With this change, we can implement a line-drawing algorithm known as *Bresenham's algorithm*. Pseudo-code for this algorithm is given in Figure 5. The first 15 lines of the algorithm make the needed adjustments depending on whether or not the line is steep. Then, in lines 17 to 22 the algorithm increments the *x* variable 1 step at a time and computes the *y* value. The *y* value is incremented when needed to stay as close to the ideal location of the line as possible. Bresenham's algorithm calculates an *error* variable to decide whether or not to increment each *y* value. The version of the algorithm shown in Figure 5 uses only integers to perform all calculations. To understand how this algorithm works, you can read about Bresenham's algorithm in a textbook or by searching for it on the internet.

Perform the following:

1. Write a C-language program that implements Bresenham's line-drawing algorithm, and uses this algorithm to draw a few lines on the screen. An example of a suitable main program is given in Figure 6. The code first determines the address of the pixel buffer by reading from the pixel buffer controller, and stores this address into the global variable *pixel_buffer_start*. The main program clears the screen, and then draws four lines. An example of a function that uses the global variable *pixel_buffer_start* is shown at the end of Figure 6. The function *plot_pixel ()* sets the pixel at location *x*, *y* to the color *line_color*.
2. Create a new Monitor Program project for the DE1-SoC Computer to use with your C code.
3. Connect a VGA monitor to the DE1-SoC board, and compile and run your program.

```
draw_line(x0, x1, y0, y1)

    boolean is_steep = abs(y1 − y0) > abs(x1 − x0)
    if is_steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)

    int deltax = x1 − x0
    int deltay = abs(y1 − y0)
    int error = −(deltax / 2)
    int y = y0
    if y0 < y1 then y_step = 1 else y_step = −1

    for x from x0 to x1
        if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
        error = error + deltay
        if error ≥ 0 then
            y = y + y_step
            error = error − deltax
```

Figure 5: Pseudo-code for a line-drawing algorithm.

Part II

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing this same object at different locations on the screen. A simple way to "move" an object is to first draw the object at one position, and then after a short time erase the object and draw it again at another nearby position.

To realize animation it is necessary to move objects at regular time intervals. The VGA controller in the DE1-SoC Computer redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, it is reasonable to control an animation using this unit of time.

To ensure that you change the image only once every $1/60^{th}$ of a second, use the pixel buffer controller to synchronize with the vertical synchronization cycle of the VGA controller. As we discussed in the background section of this exercise, synchronizing with the VGA controller can be accomplished by writing the value 1 into the *Buffer* register in the pixel buffer controller, and then waiting until bit S of the *Status* register becomes equal to 0. For this part of the exercise you do not need to use a back buffer, so ensure that the *Buffer* and *Backbuffer* addresses in the pixel buffer controller are the same. In this approach, a pixel buffer "swap" can be used as a way of synchronizing with the VGA controller via the S bit in the *Status* register.

```
volatile int pixel_buffer_start;                  // global variable

int main(void)
{
    volatile int * pixel_ctrl_ptr = (int *) 0xFF203020;
    /* Read the location of the pixel buffer from the pixel buffer controller */
    pixel_buffer_start = *pixel_ctrl_ptr;

    clear_screen ( );
    draw_line( 0, 0, 150, 150, 0x001F);       // this line is blue
    draw_line (150, 150, 319, 0, 0x07E0);    // this line is green
    draw_line (0, 239, 319,239 , 0xF800);    // this line is red
    draw_line (319, 0, 0, 239, 0xF81F);      // this line is a pink color
}

··· code not shown for clear_screen() and draw_line() subroutines

void plot_pixel(int x, int y, short int line_color)
{
    *(short int *)(pixel_buffer_start + (y << 10) + (x << 1)) = line_color;
}
```

Figure 6: Main program for Part I.

Perform the following:

1. Write a C-language program that moves a horizontal line up and down on the screen and "bounces" the line off the top and bottom edges of the display. Your program should first clear the screen and draw the line at a starting row on the screen. Then, in an endless loop you should erase the line (by drawing the line using black), and redraw it one row above or below the last one. When the line reaches the top, or bottom, of the screen it should start moving in the opposite direction.
2. Make a new Monitor Program project to test your code. Notice how long it takes for the horizontal line to move through the 240 lines of the VGA display. It should take $240 \times 1/60 = 4$ seconds.

Part III

Having gained the basic knowledge about displaying images and animations, you can now create a more interesting animation.

You are to create an animation of eight small filled rectangles on the screen. These rectangles should appear to be moving continuously and "bouncing" off the edges of the screen. The rectangles should be connected with lines to form a chain. An illustration of the animation is given in Figure 7. Part a of the figure shows one position of the rectangles with arrows that indicate the directions of movement, and Figure 7b shows a subsequent position of the rectangles. In each step of your animation each of the rectangles should appear to "move" on a diagonal line: up/left, up/right, down/left, or down/right. Move the rectangles one row and one column at a time on the VGA screen.

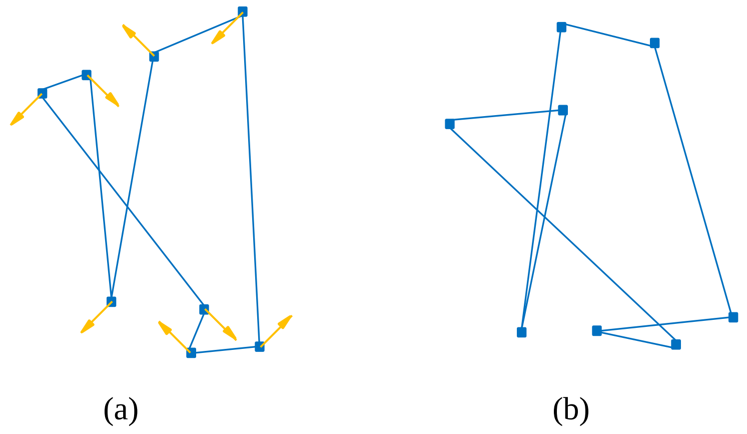


Figure 7: Two instants of the animation.

To make the animation look slightly different each time you run it, use the C library function *rand ()* to help calculate initial positions for each of the rectangles, and to determine their directions of movement.

Perform the following:

1. Write a C-language program to implement your animation. Use both a front and back buffer in your program, so that you can avoid making changes to the image while it is being displayed by the pixel buffer controller. An example of a suitable main program is given in Figure 8. The code sets the location in memory of both the front and back pixel buffers—the front buffer is set to the start of the FPGA on-chip memory, and the back buffer to the starting address of the SDRAM chip in the DE1-SoC Computer. In each iteration of the while loop the code clears the entire screen, draws the rectangles and lines, and then updates the locations of rectangles. At the bottom of the while loop the code calls the function *wait_for_vsync ()*, which synchronizes with the VGA controller and swaps the front and back pixel buffer pointers.
2. Make a new Monitor Program project to test your code.
3. Experiment with your code by modifying it to use just a single pixel buffer (simply change the address of the back buffer to be the same as the front buffer). Explain what you see on the VGA screen as a result of this change.

Part IV

For this part of the exercise you are to enhance the animation from Part III so that during the animation the following changes can take place:

1. The speed of movement of the rectangles can increased or decreased
2. The number of rectangles can be increased or decreased
3. The lines between rectangles can be drawn or not drawn

```
volatile int pixel_buffer_start;                      // global variable

int main(void)
{
    volatile int * pixel_ctrl_ptr = (int *) 0xFF203020;
    ··· declare other variables (not shown)
    ··· initialize location and direction of rectangles (not shown)
    /* initialize the location of the front pixel buffer in the pixel buffer controller */
    *(pixel_ctrl_ptr + 1) = 0xC8000000;               // first store the address in the back buffer
    /* now, swap the front and back buffers, to initialize front pixel buffer location */
    wait_for_vsync ();
    /* initialize a pointer to the pixel buffer, used by drawing functions */
    pixel_buffer_start = *pixel_ctrl_ptr;
    clear_screen( );                                  // pixel_buffer_start points to the pixel buffer
    /*    Set a location for the pixel back buffer in the pixel buffer controller */
    *(pixel_ctrl_ptr + 1) = 0xC0000000;
    pixel_buffer_start = *(pixel_ctrl_ptr + 1);       // we draw on the back buffer

    while (1)
    {
        /* Erase any boxes and lines that were drawn in the last iteration */
        clear_screen( );                              // pixel_buffer_start points to the pixel buffer

        ··· code for drawing the boxes and lines (not shown)
        ··· code for updating the locations of boxes (not shown)

        wait_for_vsync ();                            // swap front and back buffers on VGA vertical sync
        pixel_buffer_start = *(pixel_ctrl_ptr + 1);   // update back buffer pointer
    }
}

··· code for subroutines (not shown)
```

Figure 8: Main program for Part III.

In Part III the speed of animation was set by the 1/60 seconds VGA vertical synchronization time. One way to control the speed of animation is to make use of a timer. In this scheme, the main program would draw the next step of the animation each time the timer expires. Lengthening the timeout would produce a slower animation, and shortening the timeout would speed up the animation. The maximum speed of the animation would be limited by the 1/60 seconds VGA synchronization time, as it was in Part III. To cause the animation to appear to move more quickly than in Part III, you have to increase the screen-distance that the rectangles move in each step of the animation.

Perform the following:

1. Implement the speed control discussed above for the animation. The speed of animation should approximately double when you press pushbutton KEY_0, and it should reduce by the same amount when you press KEY_1. Pressing KEY_2 should cause the program to display one fewer rectangle, down to a minium of one, and pressing KEY_3 should increase the number of rectangles to some maximum of your choosing. You can process the pushbutton *KEYs* using either polled I/O or using interrupts. Finally, when any slide switch SW_{9-0} is set to the 1 position the lines between rectangles should not be drawn; only when all *SW* switches are set to the 0 position should the lines appear.
2. Make a new Monitor Program project to test your code.
3. Add any other animation features that you may find interesting.