Laboratory Exercise 7

Introduction to Graphics and Animation

The purpose of this exercise is to learn how to display images and perform animation. We will use the DE-series Media Computer and the Video Graphics Array (VGA) Digital-to-Analog Converter (DAC) on an Altera DE-series Board. This lab document is intended for DE1, DE2, DE2-70 and DE2-115 boards only.

Background

The DE-series Media Computer uses a number of circuits, called *cores*, to control the VGA DAC and display images on a screen. These include a VGA Pixel Buffer and a VGA controller circuit, which are used together with the SRAM memory and the SRAM controller to allow programs executed by the Nios II processor to generate images for display on the screen. The relevant portion of the DE-series Media Computer is shown in Figure 1.

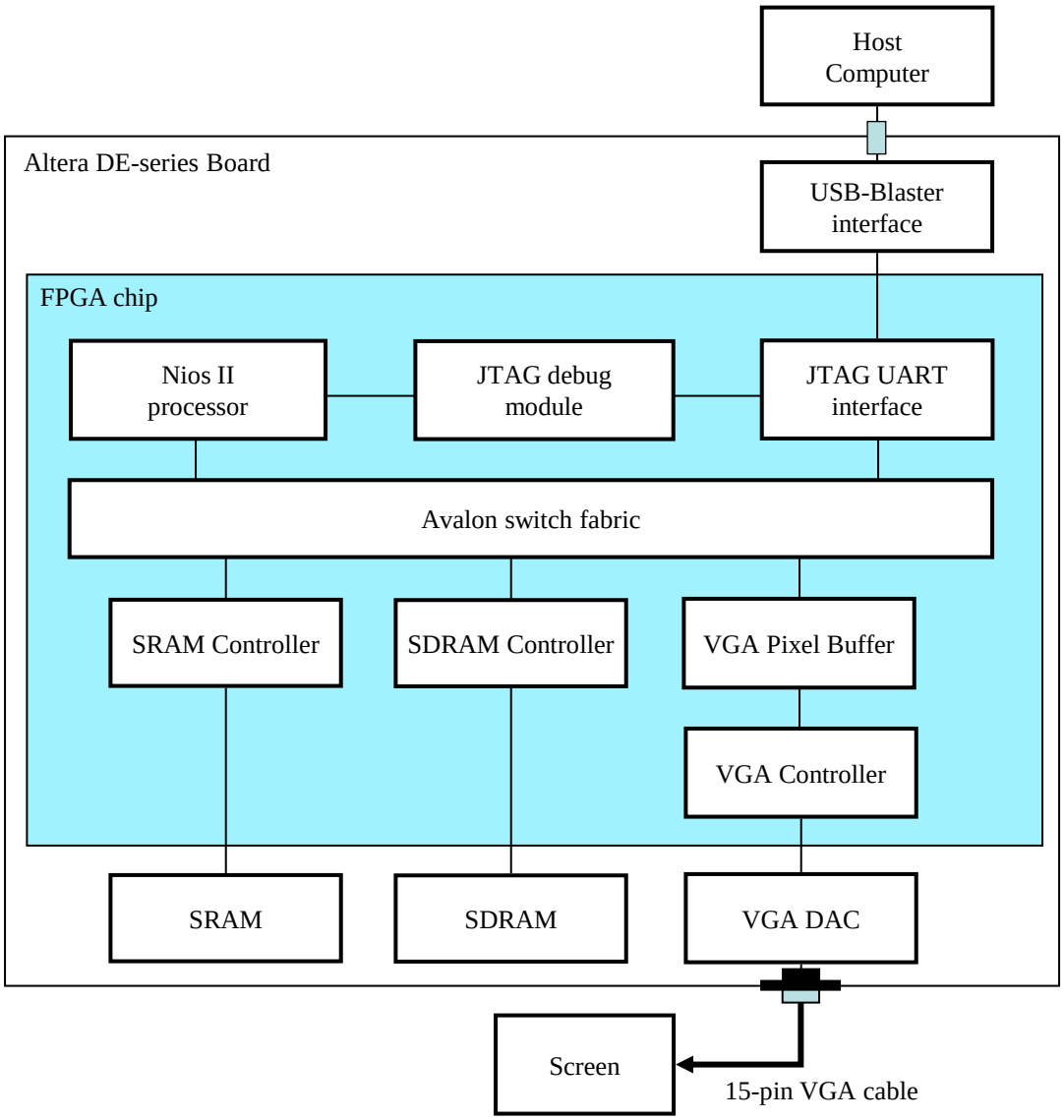


Figure 1. Portion of the DE-series Media Computer used in this exercise

The VGA pixel buffer serves as an interface between programs executed by the Nios II processor and the VGA controller. It gives the size of the screen and the location in the SRAM memory where an image to be displayed is stored. To display the image on the screen, the VGA pixel buffer retrieves it from the SRAM memory and sends it to the VGA controller. The VGA controller then uses the VGA Digital-to-Analog Converter to send the image data across the VGA cable to the screen.

An image consists of a rectangular array of picture elements, called *pixels*. Each pixel appears as a dot on the screen, and the entire screen consists of 320 columns by 240 rows of pixels, as illustrated in Figure 2. Pixels are arranged in a rectangular grid, with the coordinate $(0, 0)$ at the top-left corner of the screen.

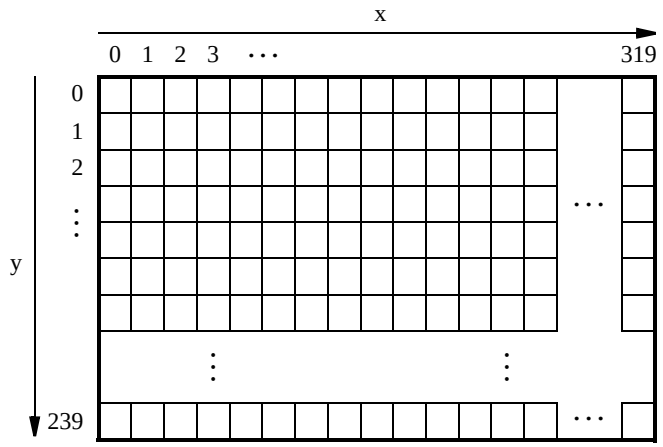
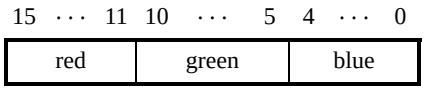
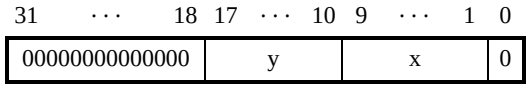


Figure 2. Pixel array.

The color of a pixel is a combination of three primary colors: red, green and blue. Any color can be created by varying the intensity of each primary color. We use a 16-bit halfword to represent the color of a pixel. The five most-significant and least-significant bits represent the intensity of the red and blue components, respectively, while the remaining six bits represent the intensity of the green color component, as shown in Figure 3a. For example, a red color would be represented by the value $(F800)_{16}$, a purple color by $(F81F)_{16}$, white by $(FFFF)_{16}$, and gray by $(8410)_{16}$.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 3. Pixel color and offset.

The color of each pixel in an image is stored at a corresponding address in a buffer in the SRAM memory. The address of a pixel is a combination of a *base* address and an (x, y) offset. In the DE-series Media Computer, the buffer is located at address $(08000000)_{16}$, which is the starting address of the SRAM memory. The (x, y) offset is computed by concatenating the 9-bit x coordinate starting at bit b_1 and the 8-bit y coordinate starting at bit b_{10}, as shown in Figure 3b. This computation is accomplished in C programming language by using the left-shift operator:

$$\text{offset} = (x << 1) + (y << 10)$$

To determine the location of each pixel in memory, we add the (x, y) offset to the base address. Thus, the pixel at location $(0, 0)$ has the address $(08000000)_{16}$, the pixel at $(1, 0)$ has the address *base* + $(00000002)_{16}$ = $(08000002)_{16}$, the pixel at $(0, 1)$ has the address *base* + $(00000400)_{16}$ = $(08000400)_{16}$, and the pixel at location $(319, 239)$ has the address *base* + $(0003BE7E)_{16}$ = $(0803BE7E)_{16}$.

To display images from a program running on the DE-series Media Computer, the VGA pixel buffer module contains memory-mapped registers that are used to access the VGA pixel buffer information and control its operation. These registers, located at starting address $(10003020)_{16}$, are listed in Figure 4.

Address	31 . . . 24	23 . . . 16	15 . . . 8	7 . . . 4	3	2	1	0	
0x10003020	front buffer address								Buffer register
0x10003024	back buffer address								Backbuffer register
0x10003028	Y		X						Resolution register
0x1000302C	m	n	Unused	B	Unused	A	S		Status register

Figure 4. VGA pixel buffer memory-mapped registers.

The *Buffer* and *Backbuffer* registers store the location in the memory where two image buffers are located. The first buffer, called the *front buffer*, is the memory where the image currently visible on the screen is stored. The second buffer, called the *back buffer*, is used to draw the next image to be displayed. Initially, both registers store the value $(08000000)_{16}$. We will discuss how to use these buffers in Part III of the exercise.

The *Resolution* register holds the width and height of the screen in terms of pixels. The 16 most-significant bits give the vertical resolution, while the 16 least-significant bits give the horizontal resolution of the screen. The *Status* register holds information about the VGA pixel buffer. We will discuss the use of these registers as they are needed in the exercise.

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 they resemble a line as closely as possible. Consider the example in Figure 5.

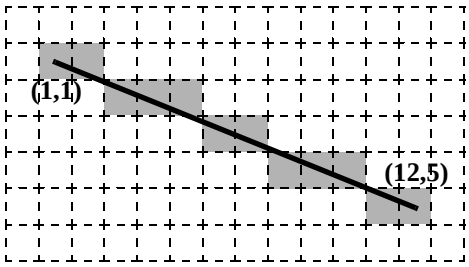


Figure 5. Drawing a line between points $(1, 1)$ and $(12, 5)$.

We want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares represent pixels that can be colored. To draw a line using pixels, we have to follow the line and for each column color the pixel closest to the line. To form a line between points $(1, 1)$ and $(12, 5)$ we color the shaded pixels in the figure.

We can use algebra to determine which pixels to color. This is done using the end points and the slope of the line. The slope of the 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:

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

Thus, for column $x = 2$, the y location of the pixel is $\frac{4}{11} + 1 = 1\frac{4}{11}$. Because 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 $y = 1$. We perform this computation for each column between x_1 and x_2.

The approach of moving along the x axis has a drawback when a line is steep. A steep line spans more rows than columns, so if the line-drawing algorithm moves along the x axis to compute the y coordinate for each column there will be gaps in the line. For example, a vertical line has all points in a single column, so the algorithm would fail to draw it properly. To remedy 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. The pseudo-code for the algorithm is shown in Figure 6.

```
1  draw_line(x0, x1, y0, y1)
2
3      boolean is_steep = abs(y1 - y0) > abs(x1 - x0)
4      if is_steep then
5          swap(x0, y0)
6          swap(x1, y1)
7      if x0 > x1 then
8          swap(x0, x1)
9          swap(y0, y1)
10
11     int deltax = x1 - x0
12     int deltay = abs(y1 - y0)
13     int error = -(deltax / 2)
14     int y = y0
15     if y0 < y1 then y_step = 1 else y_step = -1
16
17     for x from x0 to x1
18         if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
19         error = error + deltay
20         if error >= 0 then
21             y = y + y_step
22             error = error - deltax
```

Figure 6. Pseudo-code for a line-drawing algorithm.

The original algorithm uses floating-point operations to compute the location of each pixel in the line. Since floating-point operations are usually much slower to perform, most implementations of this algorithm are altered to use integer operations only. For example, the code shown in Figure 6 is optimized to use only integer operations.

Write a C-language program that draws a few lines on the screen using this algorithm. Do the following:

1. Write a C-language program that implements the line algorithm.
2. Create a new project for the DE-series Media Computer using the Altera Monitor Program.
3. Download the DE-series Media Computer onto the DE-series board.
4. Connect a 15-pin VGA cable to the VGA connector on the DE-series board and the monitor.

   The VGA cable can be connected to the screen in two ways. If the screen has multiple VGA input ports, you can connect the VGA cable to an unused port. Then, using the buttons on the screen change the video source to the corresponding port. If the screen has only a single VGA port, a KVM (Keyboard-Video-Mouse) switch is needed. Using the KVM switch, you can connect multiple video sources to a single screen. This device will allow you to choose which video source will be displayed.
5. Compile and run your program.

Part II

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing the same object at different locations on the screen. To move an object on the screen we must display it at one position first, and then at another later on. A simple way to achieve this is to draw the object at one position, and then erase it and draw it at another position.

The key to animation is timing, because to realize animation it is necessary to move objects at regular time intervals. The time intervals depend on the graphics controller. This is because the controller draws images onto a screen at regular time intervals. The VGA controller in the DE-series Media Computer redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, this will be the unit of time.

To ensure that we change the image only once every $1/60^{th}$ of a second, we use the VGA pixel buffer to synchronize a program executing on the DE-series Media Computer with the redraw cycle of the VGA controller. This is accomplished by writing the value 1 into the *Buffer* register and waiting until bit b_0 of the *Status* register in the VGA pixel buffer becomes equal to 0. This signifies that a $1/60^{th}$ of a second has passed since the last time an image was drawn on the screen.

Write a C-language program that moves a horizontal line vertically across the screen and bounces it off the top and bottom edges of the screen. Your program should first clear the screen, by setting all pixels to black color, and then repeatedly draw and erase (draw the same line using the black color) the line during every redraw cycle. When the line reaches the top, or the bottom, of the screen it should start moving in the opposite direction.

Part III

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

Write a program that animates eight small filled rectangles on the screen. These rectangles are moving continuously and bouncing off the edges of the screen. They are connected together with lines to form a chain. An illustration of the animation is shown in Figure 7. Figure 7a shows an instant where the rectangles are moving in the direction of red arrows, and Figure 7b shows the instant after the rectangles have moved.

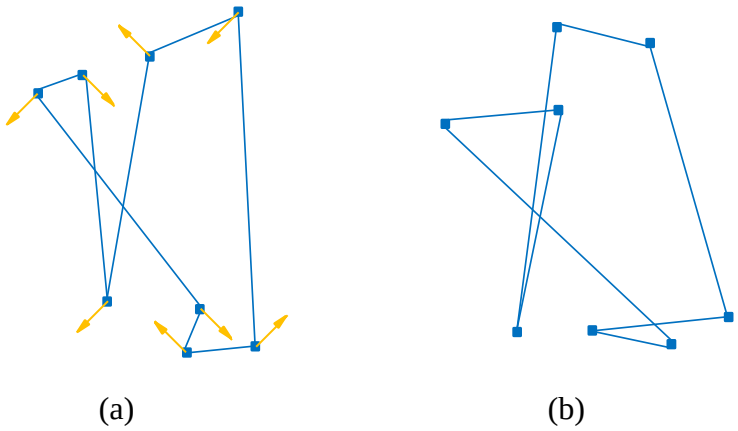


Figure 7. Two instants of the animaion.

In order to make the anmiation look different each time you run it, use *rand()* function for the initial positions of the rectangles and the directions of their movement.

You can enhance your program by using switches as an input to make your animation controllable. By toggling the switches, you can control the speed of the movement of rectangles, and stop/start the movement of some rectangles on the screen.

When you run your program, you may notice that the animation appears to flicker. This happens when the computer takes too much time to render a new image on the screen. This looks bad and is undesirable in computer

animation.

A commonly used technique called double buffering can remedy this problem. Double buffering uses two buffers, rather than just one, to render an image on the screen. One of the buffers is visible on the screen and the other is hidden. The visible buffer is called the front buffer and the hidden buffer is called the back buffer. To create an animation using two buffers, we draw an image in the back buffer. When the image is ready and the VGA controller is about to draw a new image on the screen, we swap the front and the back buffers. This operation makes a seamless transition from one image to another. Once the buffers are swapped, the back buffer becomes the front buffer and vice versa.

The VGA pixel buffer in the DE-series Media Computer supports double buffering. The location of the buffers in memory is stored in registers *Buffer* and *Backbuffer*, shown in Figure 4. Initially, the location of both buffers is the same, thus only one buffer is used. To enable double buffering, we need to separate the front and the back buffers by setting the *Backbuffer* register to address $(08040000)_{16}$. This will cause half of the SRAM memory in the DE-series Media Computer to be used for the front buffer, and the other half for the back buffer.

Each time you want to display the image in the back buffer, you need to write 1 to the *Buffer* register. By doing this, the VGA pixel buffer will swap the address stored in *Buffer* register and *Backbuffer* register. The swapping can take up to $1/60^{th}$ second because it must wait until the front buffer has been completely drawn. You should continuously poll bit b_0 of the *Status* register for a value 0 before you can draw a new image in the new back buffer.

When using double buffering, your program should only draw images in the back buffer and erase the back buffer after every buffer swap. Also, the program should use two pointers pointing to the front buffer and the back buffer. When you ask the VGA pixel buffer to swap its buffers, you should swap the pointers as well.

Preparation

The recommended preparation for this laboratory exercise includes C code for Parts I to III.