# ECSE 324: Computer Organization Lab 3: Keyboard and VGA

Hongtao Xu 260773785

**Part 1: Drawing things with VGA**

Brief description:

In this lab, we will use the high-level I/O capabilities of the DE1-SoC simulator to display pixels and characters using a VGA controller and accept keyboard input via the PS/2 port.

In part 1, we are going to simply display a colorful image with the words “Hello World!” on the VGA pixel buffer, which is 320 pixels wide and 420 pixels high. To draw things with VGA, we are going to create a VGA driver. I need to create four-driver subroutines to support this application: VGA_draw_point_ASM, VGA_clear_pixelbuff_ ASM, VGA_write_char_ASM, and VGA_clear_charbuff_ASM.

Approach taken:

The DE1-SoC computer has a built-in VGA controller that can render pixels, characters, or a combination of both. Regarding rendering pixels, the VGA controller reads the pixel buffer from region 0xc8000000 which contains the color value of every pixel on the screen, in detail, 0-4 is a blue channel, 5-10 is a green channel, and 11-15 is a red channel. Also, there are x and y coordinates to access the individual pixel colors. We can also render characters using character buffer, which is analogous to the pixel buffer while for the characters. The characters themself are a buffer of byte-sized ASCII at 0xc9000000, and we can access individual characters through x and y coordinate.

There are four functions mentioned in the brief description controlling the display over VGA pixel buffer. Firstly, VGA_draw_point_ASM draws a point on the screen with the color indicated above, by accessing only the pixel buffer memory. In detail, x and y are used for indicating location with the indicated color. In the simulator, w use R0, R1, and R2 to represent x, y, and c respectively. In order to access a specific position on pixel buffer, we use LSL to shift x and y coordinates left respectively, then combine them together to form the offset of the memory location. We put the initial location of VGA pixel buffer 0xC8000000 to this offset. Finally, we store the color value kept in r2 into this calculated memory location by using STRH. Secondly, VGA_clear_pixelbuff_ASM is used to clear (sets to 0) all the valid memory locations in the pixel buffer. To be specific, we are filling all the memory locations with 0 to make the entire buffer dark. I implement this function by calling VGA_draw_point_ASM with a color value of zero for every valid location on the screen. For each pixel in VGA, we fill in r0 and r1 with its coordinates and r2 with 0, and we increase the pixel coordinates after clearing each pixel. Next, VGA_write_ char_ASM writes the ASCII code passed in the third argument (r2) to the screen at the (x, y) coordinates given in the first two arguments (r0 and r1). It is pretty similar to what is done for VGA_draw_point_ASM, character buffer is similar to the pixel buffer, instead of a 320*240 sized memory location, it is an 80*60 sized memory location for writing characters. In detail, the subroutine will store the value of the third argument at the address calculated from the first two arguments. To make sure that the input memory location is valid, we check if r0(x) and r1(y) are smaller or equal to 80 and 60 respectively. I also use LSL to shift r1 to the left for 7 bits and the initial location #0xC9000000. We use STRB in this subroutine. Lastly, VGA_clear_ charbuff_ASM is to clear (sets to 0) all the valid memory locations in the character buffer, which takes no argument and returns nothing. I call VGA_write_char_ASM with a character value of zero for every valid location on the screen to implement this function. In detail, we put r0 and r1 with the location of coordinates inside a loop and r2 with 0, after we clear all valid locations inside a character buffer, we increase the pixel coordinates.

Challenges and possible improvement:

In this part, I am confused about how to display the correct colors on the VGA pixel buffer and find out which provided registers I should use here. After reading and analyzing the code, I found that all colors have been provided and found the right register to implement it.

**Part 2: Reading keyboard input**

Brief description:

Part 2 is about a high-level description of the PS/2 keyboard protocol. Part 2 is pretty simple comparing to part 1, we can use the subroutines we wrote in part 1. Our goal is to display a whole black image on the VGA pixel buffer, and then we print on it with the MAKE key and BREAK key of the corresponding key when we pressed or released it. I type in the PS/2 keyboard as input. To be specific, there is a keystroke event sending hexadecimal numbers called scan codes, varying from 1-3 bytes in length. There are two parameters involved here, the typematic delay and the typematic rate. When a key on the PS/2 keyboard is pressed, a unique scan code called make code is sent, then a break code is sent when the key is released, another scan code called the break code is sent. The make code would stop be sent if the key is released or another key is pressed. There is only one new subroutine we need to deal with: read_PS2 _data_ASM.

Approach taken:

As I can copy the most relative subroutines from part1, this part becomes much easier for me. The only new approach I have taken in this part is read_PS2 _data_ASM. In detail, DE1-Soc receives keyboard input from a memory-mapped PS/2 data register at address 0xff200100. There is input argument and output argument, input is the memory address in which the data is read from the PS/2 keyboard and stored, and output is checking RTVALID valid or not. When RVALID is valid, the data from the same register should be stored at the address in the pointer argument, and the subroutine should return 1 to denote valid data. The lowest 8 bits of the PS/2 data register are corresponding to a byte of keyboard data, and all the rest bits showing 0. The RVALID bit can be accessed by shifting data register 15 to the right and extracting the lowest bit by using the LSR method. If the RAVLID bit is not set, the subroutine should simply return 0. The obtained result is the RVALID bit, which is moved to R0. In this way, the character buffer could receive and display our data. Also, the write_bit subroutine is provided, which allows me to type in “Type here” and check if the PS/2 driver working correctly. And I did not meet many challenges in this part, everything is pretty straightforward.

**Part 3: Putting everything together: Vexillology**

Brief description:

In this part, I am going to create an application that a gallery of flags. I need to draw three flags, one fixed flag, one real-life flag (China’s), and one imaginary flag (Mines). If I press A on the “type here”, I can see the China flag, and if I press D on the “type here”, I can see the imaginary flag. Generally, this part is the verification of part 1 and part 2. We can use all the subroutines provided from the previous two parts, and there are two new subroutines: draw_rectangle and draw_star. We use those two subroutines to draw rectangles and stars, also defining their size, the center location, and colors.

Approach taken:

The approach in part 3 is pretty simple. I will mainly talk about two new subroutines since the other subroutines are provided from part 1 and part 2. For draw_rectangle, it is to draw a rectangle, which takes five arguments. The first four arguments are stored in r0 though r3, and the fifth argument is stored on the stack at the address [sp]. For draw_star, it is to draw a star, which takes four arguments, stored in r0 through r3. When I draw those flags, the only task is to set the parameters into the corresponding locations, colors, and sizes of each shape and call the corresponding subroutines. For the real-lift flag, I draw the Chinese flag and for the imaginary flag, I draw three different colors rectangles side by side.

Challenges and possible improvement:

I did not meet too many challenges here, the only difficulty is to find the correct position of those tiny stars while drawing the Chinese flag.