Lab3 – High Level IO

ECSE-324 Computer Organization

Sen Wang (260923645)

Friday December 4 2020

McGill University
Montreal Quebec, Canada

Part 1: Drawing Things with VGA

Brief Description:

This assembly program contains four main subroutines to implement. The first one is VGA_draw_point_ASM. This subroutine draws a point on the screen with the color as indicated in the third argument, by accessing only the pixel buffer memory. The second one is VGA_clear_pixelbuff_ASM. It clears all the valid memory locations in the pixel buffer. It takes no arguments and returns nothing. The third subroutine is VGA_write_char_ASM. It writes the ASCII code passed in the third argument to the screen at the (x, y) coordinates given in the first two arguments (r0 and r1). Finally, VGA_clear_charbuff_ASM clears all the valid memory locations in the character buffer. It takes no arguments and returns nothing.

Approach Taken:

The approach taken for VGA_draw_point_ASM is straightforward. We know that the pixel buffer is 320 pixels wide and 240 pixels high and individual pixel colors and be accessed at 0xc8000000 | (y << 10) | (x << 1). To draw a point at a specific color, we just need to LSL X by 1 and Y by 10, add the two values and store the result in the pixel buffer. Similarly, VGA_clear_pixelbuff_ASM is implemented by setting all the valid memory location in the pixel buffer to 0 by using a nested for loop (x and y). VGA_write_char_ASM implementation is also simple. Here, we use the character buffer, which is analogous to the pixel buffer, but for characters. The devices' VGA controller continuously reads the character buffer and renders its contents as characters in a built-in font. The character buffer itself is a buffer of byte-sized ASCII character at 0xc9000000. The buffer has a width of 80 characters and a height of 60 characters. An individual character can be accessed at 0xc9000000|(y <<7) x. Finally, the implementation of VGA_clear_charbuff_ASM is done by calling VGA_write_char_ASM with a character value of zero for every valid location on the screen.

Challenges Faced and Solutions

The logic used in this lab is relatively simple and I did not face any major challenges or difficulties. The main part that was the most time consuming was the nested for loop. A few internet searches and readings in Piazza helped me solve this problem.

Possible Improvement to the Program

Several improvements could be made to this program. I believe that my implementation of the nested for loop is not the most optimal one because it uses a lot of registers. Some registers were only being used to store intermediate calculation values. Reducing the number of registers used could make the program more flexible in the future if we want to make modification on it.

Part 2: Reading Keyboard Input

Brief Description:

This assembly program contains the same four subroutines for part 1 of this lab. The first one is VGA_draw_point_ASM. This subroutine draws a point on the screen with the color as indicated in the third argument, by accessing only the pixel buffer memory. The second one is VGA_clear_pixelbuff_ASM. It clears all the valid memory locations in the pixel buffer. It takes no arguments and returns nothing. The third subroutine is VGA_write_char_ASM. It writes the ASCII code passed in the third argument to the screen at the (x, y) coordinates given in the first two arguments (r0 and r1). The fourth subroutine is VGA_clear_charbuff_ASM. It clears all the valid memory locations in the character buffer. It takes no arguments and returns nothing. Finally, we have an additional subroutine called read_PS2_data_ASM. A memory address in which the data is read from the PS/2 keyboard will be stored as input and an integer that denotes whether the data read is valid or not is the output.

Approach Taken:

The approach taken for this part of the lab is very similar to part 1 since they mostly contain the same subroutines. The approach taken for VGA_draw_point_ASM is straightforward. We know that the pixel buffer is 320 pixels wide and 240 pixels high and individual pixel colors and be accessed at 0xc8000000 | (y << 10) | (x << 1). To draw a point at a specific color, we just need to LSL X by 1 and Y by 10, add the two values and store the result in the pixel buffer. Similarly, VGA_clear_pixelbuff_ASM is implemented by setting all the valid memory location in the pixel buffer to 0 by using a nested for loop (x and y). VGA_write_char_ASM implementation is also simple. Here, we use the character buffer, which is analogous to the pixel buffer, but for characters. The devices' VGA controller continuously reads the character buffer and renders its contents as characters in a built-in font. The character buffer itself is a buffer of byte-sized ASCII character at 0xc9000000. The buffer has a width of 80 characters and a height of 60 characters. An individual character can be accessed at 0xc9000000|(y <<7) x. Finally, the implementation of VGA_clear_charbuff_ASM is done by calling VGA_write_char_ASM with a character value of zero for every valid location on the screen. Finally, the approach taken for read_PS2_data_ASM check the RVALID bit in the PS/2 Data register. If it is valid, then 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. If the RVALID bit is not set, then the subroutine should simply return 0.

Challenges Faced and Solutions:

I did not face any main challenges for this part of the lab. They were solved in part 1.

Possible Improvements to the Program:

Several improvements could be made to this program. I believe that my implementation of the nested for loop is not the most optimal one because it uses a lot of registers. Some registers were only being used to store intermediate calculation values. Reducing the number of registers used could make the program more flexible in the future if we want to make modification on it.

Part 3: Putting Everything Together, Vexillology

Brief Description:

This assembly program contains the same subroutines for part 2 of this lab. The first one is VGA_draw_point_ASM. This subroutine draws a point on the screen with the color as indicated in the third argument, by accessing only the pixel buffer memory. The second one is VGA_clear_pixelbuff_ASM. It clears all the valid memory locations in the pixel buffer. It takes no arguments and returns nothing. The third subroutine is VGA_write_char_ASM. It writes the ASCII code passed in the third argument to the screen at the (x, y) coordinates given in the first two arguments (r0 and r1). The fourth subroutine is VGA_clear_charbuff_ASM. It clears all the valid memory locations in the character buffer. It takes no arguments and returns nothing. We also have a subroutine called read_PS2_data_ASM. A memory address in which the data is read from the PS/2 keyboard will be stored as input and an integer that denotes whether the data read is valid or not is the output. We have two additional subroutines. One is draw_rectangle and it takes five arguments. The first four arguments are stored in registers R0 through R3. The fifth argument is stored on the stack at address [sp]. Finally, we have draw_star that takes four arguments, stored through registers R0 through R3.

Approach Taken:

The approach taken for this part of the lab is the same as part 2 since they mostly contain the same subroutines. The approach taken for VGA_draw_point_ASM is straightforward. We know that the pixel buffer is 320 pixels wide and 240 pixels high and individual pixel colors and be accessed at 0xc8000000 | (y << 10) | (x << 1). To draw a point at a specific color, we just need to LSL X by 1 and Y by 10, add the two values and store the result in the pixel buffer. Similarly, VGA_clear_pixelbuff_ASM is implemented by setting all the valid memory location in the pixel buffer to 0 by using a nested for loop (x and y). VGA_write_char_ASM implementation is also simple. Here, we use the character buffer, which is analogous to the pixel buffer, but for characters. The devices' VGA controller continuously reads the character buffer and renders its contents as characters in a built-in font. The character buffer itself is a buffer of byte-sized ASCII character at 0xc9000000. The buffer has a width of 80 characters and a height of 60 characters. An individual character can be accessed at 0xc9000000|(y <<7) x. Finally, the implementation of VGA_clear_charbuff_ASM is done by calling VGA_write_char_ASM with a character value of zero for every valid location on the screen. Finally, the approach taken for read_PS2_data_ASM check the RVALID bit in the PS/2 Data register. If it is valid, then 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. If the RVALID bit is not set, then the subroutine should simply return 0.

Challenges Faced and Solutions:

I did not face any main challenges for this part of the lab. They were solved in part 1.

Possible Improvements to the Program:

The main improvements that could be made for this program are the same as part 2 of this lab.