Sofia Velasquez Sierra
260967314
April 14th

LAB 3: Report

1. *Drawing things with VGA*

Approach: This part of the lab consisted of reading each pixel of the pixel buffer (320x240) and loading a colour into each one. A similar approach is used for the character buffer, which is read by the VGA controller.

To implement this, four driver functions were implemented. For VGA_draw_point_ASM, I began by checking if the coordinates x and y were valid; x < 320 and y < 240. If the conditions were met, a point was drawn on the screen with the determined colour c. This was done by determining the address of each pixel, which depends on the values of x and y: I implemented *lsl* of 1 for x and *lsl* of 7 for y, which were then both added to the base address. The colour c (in A3), was then stored into the computed address. The function VGA_clear_pixelbuff_ASM consists of iterating through each pixel of the pixel buffer, in order to call VGA_draw_point_ASM at each iteration and write a colour into it. I implemented this by initialising x, y and the colour c to 0. Then I implemented my two nested loops; the first loop *coordy* begins by checking if y is less than 240, the code continues onto the second loop if the condition is met. Next, the loop *coordx* checks if x is less than 320, it is not it is reset to 0, y is increased by 1 and it branches back to *coordy*. Otherwise, the loop branches to VGA_draw_point_ASM and when it returns adds 1 to x and branches to *coordx.*

The other two functions were implemented following the same structure as the first two. VGA_write_char_ASM checks that the coordinates x and y are valid; x < 80 and y < 60. It then computes the address for the ASCII code with the base address and the inputted x and y (shifted by 7). VGA_clear_charbuff_ASM initialises x, y and c to 0. Then, it enters the iteration loops, which are written in the same way as for VGA_clear_pixelbuff_ASM, but x is compared to 80, y is compared to 60 and the function branches to VGA_write_char_ASM.

Testing strategy: For his part my testing was done gradually as I wrote each function. I checked if each subroutine was compiling and then went line by line to check if everything was functioning as expected. Once the four functions were written and worked as expected, I did an integrated test on the entire code, including the given “hello world” code and made sure the pixel buffer displayed everything as expected.

Challenges faced: I did not face any significant challenges for this part of the lab.

2. *Reading keyboard input*

Approach: This part consists of writing a function that will receive the keyboard input and writes the corresponding PS/2 code on the VGA screen using the character buffer. The read_PS2_data_ASM function loads the address of the memory-mapped PS/2 data register. This is shifted by 15 in order to access the RVALID bit. If the bit is 1, the the keyboard data is outputted.

Testing strategy: To test this function, I first made sure that every line was working as I expected, i did this by going line by line. Then, I inputted different keyboard keys on the PS/2 keyboard and checked that the correct code was being displayed on the VGA screen.

Challenges faced: At first, I was not getting the correct output on the VGA screen because I had extra lines that were disturbing the right execution of the code. I modified a few lines and verifies that I did not have any extra steps, which fixed my issue.

Sofia Velasquez Sierra
260967314

Part 3. Putting everything together: Game of Life

Approach: This part of the lab consisted in implementing the game of life. It was split up in steps, each implementing a new function.

Getting started: drawing lines consisted of drawing the 16x12 grid in a certain color on the board. Two subroutines were implemented: VGA_draw_line_ASM and GoL_draw_grid_ASM. VGA_draw_line_ASM consists on drawing a line from pixel (x1, y1) to (x2, y2). The function takes four input parameters: x1, y1, x2, and y2, which specify the starting and ending points of the line to be drawn, and c, that specifies the color of the line. I start of by comparing x1 and x2, which determines if the line will be drawn vertically or horizontally. If x1 and y1 are equal, the value of y1 is moved to a new register (r4), which is then compared to y2, the current inputs for x1, y1 and c are stored and the code branches and links to VGA_draw_point_ASM. One the code links back, r4 is incremented by 1. While, y1<=y2 the code will stay in this subroutine drawing one point at a time in order to draw a line. Once the condition is not true anymore, the code goes on to the following subroutine that works in a similar way, but this time comparing x1 and x2.

GoL_draw_grid_ASM draws the grid lines on the VGA screen for the GoL board, by calling VGA_draw_line_ASM to draw each individual line. The function is divided into two parts; one to draw all the vertical lines and one to draw the horizontal lines. First, x1, x2 and y1 are set to 0 and y2 is set to 240. The loop *x_axis* is entered, and will only execute while x2 is less than 320, the maximum x in the grid. If the condition is met and the loop is entered the code branches and links to VGA_draw_line_ASM to draw a line. When the code links back, it increments x1 and x2 by 20 and branches to *x_axis.* The lines will be drawn until the maximum x (320) is reached. When this happens, we move on to the second part of the function; x1, y1 and y2 are set to 0 and x2 is set to 320. The loop *y_axis* is entered while y1 is less than 240, the maximum y value. Similarly to *x_axis,* the code will branch and link to VGA_draw_line_ASM and increment y1 and y2 by 20 when linking back. The loop will continue doing this until y1 is no longer smaller than the maximum y (240).

Next steps: drawing rectangles consists on drawing rectangles on the board implementing two functions: VGA_draw_rect_ASM and GoL_fill_gridxy_ASM. VGA_draw_rect_ASM draws a rectangle from two specified points (x1, y1) and (x2, y2). These coordinates are specified at a previous subroutine that calls VGA_draw_rect_ASM. This function uses a loop to branch and link to VGA_draw_line_ASM while incrementing x1 by 1 at each iteration, thus drawing lines from the first point to the second. The loop is only exited when x2 becomes less than x1 the loop is exited.

GoL_fill_gridxy_ASM fills an area of the grid with the colour c. It is called by GoL_draw_board_ASM to draw a rectangle on the screen. The function calculates the coordinates of the rectangle using the input arguments and calls the VGA_draw_rect_ASM function to draw the rectangle. The coordinates are computed by multiplying the given x1, y1 by 20. For x2, y2, I added 20 to x1 and y1. Once these were established, I branched and linked to VGA_draw_rect_ASM in order to draw the rectangle corresponding to the computed coordinates.

Game logic: initialization begins writing the game logic. Here, the environment of the game is initialized. The board is saved in memory as a 2D structure with dimensions of 16x12. This is done by implementing the function GoL_draw_board_ASM that will fill a grid if GoLBoard[y][x] == 1. I implemented this using three counters initialised to 0. Then, I entered the first loop *iter* which iterates through each position of the GolBoard in memory. Each value

is loaded into R7 and compared to 1 at each iteration. If R7 is equal to 1, the code branches to GoL_fill_gridxy_ASM using the current values of x and y. When linking back, a3 and a1 (x) are both incremented by 1, and in the case that a1 reaches 16 it is reset to 0 and a2 is incremented by 1. To exit the loop a3 has to be equal to 192, so after the loop has executed 192 iterations.

Testing strategy:

For this part of the lab, I tested each part of the sections as I did them. I conducted unit testing for each of them, making sure every subroutine was working as expected. When I encountered an issue, I would go line by line to debug the subroutine in question. I did integration testing up until the part I coded, making sure everything worked well together. If I had done the rest of this part, my testing would have been more complex since I would have had to take into account all the edge cases and unit tested more subroutines.

Challenges:

The main challenge I encountered was not having time to implement the entire game.

Shortcomings/improvements:

I was not able to finish coding the game but if I could have implemented it, it would have been in the following way:

First of all, for *Game logic: changing the playing field* I would create a function that would simply draw the cursor at any location on the board. This could be done by specifying the coordinates and calling VGA_draw_rect_ASM. Then I would create another subroutine that would wait, using polling, for a keyboard input (w, a, s, d, spacebar) which would change the position of the cursor and activate/deactivate a rectangle on the grid. When an input is detected, I would branch to another subroutine that would check what key was pressed and branch to another subroutine that would execute the corresponding cursor movement if it was one of the wanted keys. The subroutine corresponding to the spacebar would have to change the colour of the current rectangle to change its status.

Then, *Game logic: state update* consisted of implementing the rules for updating playing field state. I would start of by creating a loop that would repeatedly call two subroutines. The first one would iterate through every cell of GolBoard and branch to new subroutine that checks the amount of neighbours that this cell has. The second subroutine iterates through each cell of the grid and would load the value of the cell and the amount of neighbours it has. Then, I would implement subroutines to determine wether to activate, deactivate or not do anything to a cell depending on the number of neighbours it has.