Giuseppe Lipari (260742911)
Alex Masciotra (260746829)
Group 45

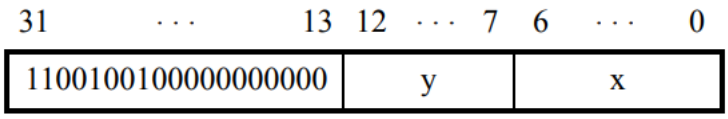
ECSE 324 Lab 4

Introduction:

This lab contained three different sections. The first being to use the VGA controller to display pixels and characters. The second part was to use the PS/2 port to accept inputs from a keyboard. Finally, the last part was to use an audio controller to play generated tones.

VGA:

For the first part of this lab, we needed to display pixels and characters by implementing a vga driver. There were three different routines that needed to be created to run this section, two were given. One of the clears is for the pixels and the other is for the characters. This was done by setting the respective register for pixels or characters to 0. Below is the character buffer register which was loaded and worked on.



(b) Character buffer addresses

To do the clear, we created an outer loop which loops through the Y values and an inner loop going through the X values. To attain the memory locations of the character buffer, we left shifted the Y values by 7. Afterwards, we got the complete memory location by ORing the base address for each register with the shifted values of X and Y. Afterwards, we stored the byte 0x00 at the computed memory location, which cleared the screen.

The third, fourth and fifth routine were the write_char, write_byte and draw_point. They verify that the target memory location is valid. To do this they check if the X and Y inputs are within the allowable buffer ranges (79 for X and 59 for Y). To find the appropriate offset for the write char and write byte, we stored the x offset from R0 into a register and then added the y offset from R1 shifted 7 bits to the left into the same register. Then the corresponding asci code in R2 was written to the offset address. For the write byte, the code was very similar except for first making room by ANDing with FF00 and 00FF to find which bits were actually on. Then the appropriate character was stored one next to eachother.

In order to test our VGA code, we also needed to implement our Pushbutton code from lab3 in order to have the four different buttons either clear, write pixel or write character. If a switch was left on, the monitor would display the bytes.

The largest challenge that we faced in this first part of the lab was to compute the ascii code for the characters to be displayed in the write_byte routine. After discussing with other groups and looking online, we were able to figure it out.

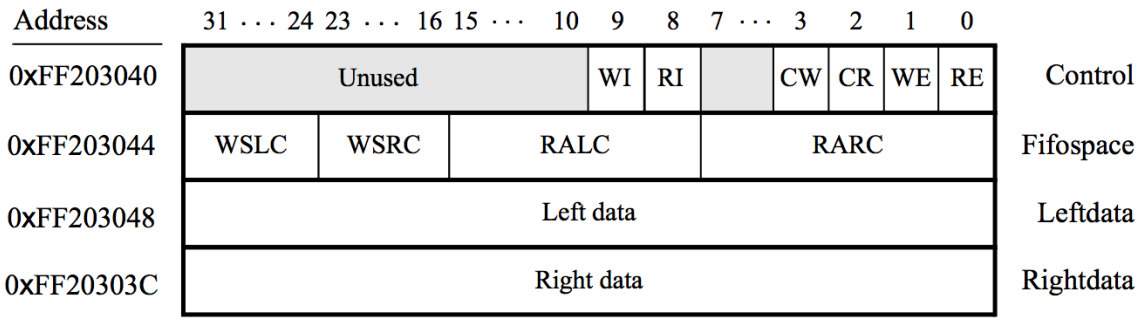
Keyboard:

The second part of this lab was the keyboard. This was implemented using the PS/2 input. We were tasked with having to implement a code that reads the keyboard and prints the make code to the screen. The way the routine worked was that it would receive a character pointer which would be stored in the FIFO. At first, we isolated the RVALID bit by ANDing with 0x8000 to check to see if there's space in the queue it loads the values of the keyboard data register. If the RVALID bit is set to zero, we can simply return 0 meaning we didn't get any new data. If it's not, then we apply a mask to the register value with value 0xFF to get only the data of the key pressed. We later stored this byte at the memory address specified by the argument.

We tested this code by creating a C program. In order to do this, we kept track of where the cursor was (where the next byte will be written). We did this by having an X value increment by 3 every time something is written to the screen. If the X value is greater than the maximum allowed, we set X=0 and increment the Y value by one to jump to the next line.

Audio:

The third and last section was to write a driver for the audio port. In essence, what we had to do was write data to the leftdata and rightdata registers. The first thing that needed to be done was to read the values of the fifospace register with the WSLC and WSRC. This contains the number of words available by applying mask values of 0xFF000000 and 0x00FF0000 to the fifospace value. If at least one of these two values is 0, then there is no space for the data and the routine returns 0, which means it was not able to write the data to the audio port. If there is space, the routine stores the data in the leftdata or rightdata registers, which are offset by 0xC and 0x8 respectively from the control register and returns 1, which means data was written to the audio port. An appears below which shows the leftdata and rightdata registers:



We tested this driver by running a C program that played a 100Hz square wave. The first thing that the program does is calculate after how many samples it should switch from a high value to low value or low to high. This was calculated using the following formula:

Number of samples = sampling rate/(2*desired frequency)

Afterwards, in a while loop, every iteration tried to write data to the audio port. If it was successful while looping, then it would update its sample count. Once the max count was reached, it would reset the sampling count and change the audio value from low to high or high to low.

The largest challenge we faced was finding the number of samples we needed to write before changing value from low to high or high to low. Once we discovered the formula above, it became quite simple.

To conclude, this was a very interesting lab with three parts. We learned how to use the VGA controller to display pixels and characters. In addition, we learned how to accept input from a keyboard as well as how to use an audio port to play generated tones.