LAB 4 REPORT

ECSE324

High level I/O - VGA, PS/2 Keyboard, and Audio

Boqi Chen 260727855

Fandi Yi 260722217

1. VGA

1.1 Implementation

In this part, VGA drivers were written and implemented. VGA_write_char_ASM and VGA_write_byte_ASM subroutines were written based on the other given code from last lab. VGA_write_char_ASM would receive input arguments that determined which coordinate to write the characters. The register R5 will be offset by input by R0 and added with R1 logical shifted left (LSL) by #1. The address in R5 will be the address of the coordinate to be assigned in. The character will be written in this address. The next subroutine VGA_write_byte_ASM uses a similar logic for the coordinates. The second character is simply offset by by 1 in the x axis. However, in order to determine the 2 characters passed as a byte in R2, it had to be AND with the numbers 15 (in binary 1111) for the first byte and 240 for the second byte. These are stored in R2 and R6 respectively. R2 and R6 will then be stored in their respective VGA addresses. Besides, we also apply two methods. They are VGA_clear_pixelbuff_ASM and VGA_clear_charbuff_ASM. As the name shown, these two subroutines are used to clear the words on the screen immediately after the screen is full. In the main.c file, we wrote the while if condition to call the subroutines. PB0 means button #0, It will check the slider switch is on or off once the PB0 is pressed. The off switch corresponded to test_byte, the on switch corresponded to test_char. Besides, the PB1 corresponded to color tests, PB2 corresponded to clear characters buffer and PB3 corresponded to clear pixel buffer.

1.1 Difficulties and Further Improvements

The VGA applications is considerably challenging for us, especially in the clear buffer subroutines. What we did what that, by observing that both of them had the same iterating structure (looking through the whole screen), and reset everything to 0, we decided to do have a subroutine for iterating the two dimensional space. However, there was still a difference between the two methods, as they did not use the same subroutine to write data. It took us a long time to figure out how to combine together. At the end, we passed the width and length of the space, along with the address of the subroutine that we wanted to use to write the resetting data (0). After realized this procedure was feasible, it took us some additional time to debug the program. But at the end, We felt that this was really an efficient and delegate way to implementing the clear routine.

2.Keyboard

2.1 Implementation

In this part, we need to show the words on the screen row by row by applying the keyboard. The PS/2 bus shown us the work principle of while key pressed and key released. For the basic idea of the of the subroutine was simple, firstly, input the keyboard's word. In order to achieve that, we need to check the bit in the position of 0x80 (8th bit) to see if there is any data ready from the keyboard ready for reading. If it is 1 meant that the keyboard was pressed and there was some data ready, 0 indicated there was no data. Then if there was any data ready, we read them by accessing the data part in the data register. Secondly, we needed to reading the word and using VGA driver to present the words on the screen. As for the main.c file, we also added the while if condition. While the keyboard is pressed, we will print the words on the screen. But for the x-y coordinates, they should have the limitation which is based on the size of the screen.(79x59).

2.1 Difficulties and Further Improvements

The keyboard applications is not really difficult and we set up the assembly file based on the VGA driver which we build up in the first part. However, while we testing this part, we found out that the screen was cleaned one row before the screen is full. The test is succeed after we changing y>=59 to y>59. Since if we use y>=59 which means the last row will be ignored and the screen will be cleaned before starting the last row. Also one improvement could be done by implementing the keyboard data reading with interrupt rather than the current polling style

3.Audio

3.1 Implementation

In this part, we need to use the audio driver to make the FPGA produce sounds by passing the arguments. In the assembly file, we read the Fifospace register of the audio device to check if we have enough space from both right and left sides of the Fifo. If there is no enough space, we just do not pass any data to the FIFO. However, if there is some space we will store the data passed to the Leftdata and Rightdata respectively. For the main.c file, firstly, we set two integers for high and low signals. When play the high signal at the beginning at interchange it between high and low. Secondly, we set up the while if condition to make the frequency of the sound to 100HZ (since the sample frequency is 48k per second, $\frac{48kHZ}{100\times2}$ =240 samples for half cycle). The audio port structure is shown below.

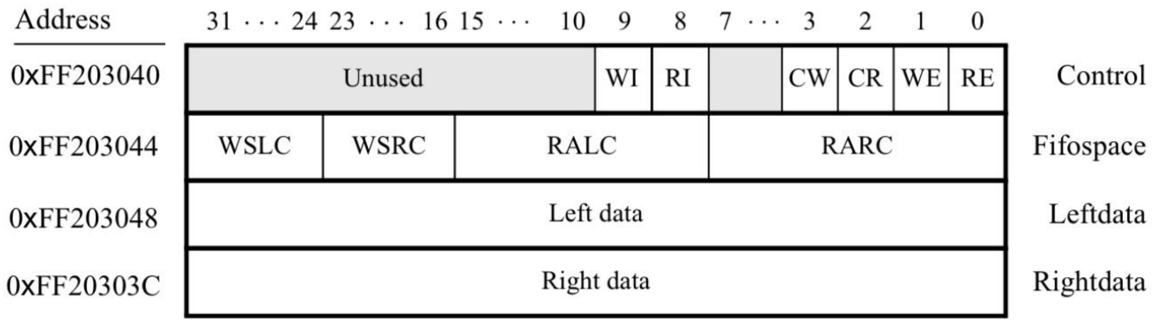

Figure. 1: Audio port registers

3.1 Difficulties and Further Improvements

During this testing, we found a bug related to the producing of sound. There is no sound we we were testing the device. We spent a lot try trying to find if there was anything wrong with the logic of the code. However, the subroutines and methods we used are literally correct. At the end, we found that the reason was that we used a wrong address for the Leftdata register when storing data. Of course, there would be no sound produced by the device. There is not much can be improved related to this specific lab, however, we find out that we should have everything regarding the main functionality in one file, rather than separated files as previous labs. So what we ended up doing was create only one single file with one main method, and each part of the lab becomes a method in the file. So that during the demo, we could only choose different method to run and there was no need for us to change the program setting anymore.