Group 30
Zhuonan Wang 260714137
Suhui Shen 260705934

ECSE 324 Lab 4 Report

Lab 4 introduces the high level I/O capabilities of the DE1 Soc board. In particular, we used VGA controller to display pixels and characters. Also, we used PS/2 keyboard to accept keyboard input. Lastly, we used audio controller to play generated tone.

1 VGA

VGA Driver

We need first to create a VGA.h file that record all the functions written in assembly file. Those functions include VGA_clear_charbuff_ASM() that clears the char buffer in memory. Similarly, VGA_clear_pixelbuff_ASM() clears the pixelbuffer in memory. VGA_write_char_ASM(int x, int y, char c) writes the character c to the coordinates (x, y) on the screen. VGA_write_byte_ASM(int x, int y, char byte) write a byte, which is 2 characters on screen and each byte is separated by a byte of blank space in between. VGA_draw_point_ASM(int x, int y, short colour) writes RGB color in pixel on the screen. Before write operation, it should check if the coordinates are within the range.

Then we need to write assembly file that actually implement the functions. For each method, we first push the registers we will be using and the link register to the stack and pop them out after the result is returned. For VGA_clear_charbuff_ASM(), we have two counters, one for x and another for y and both initiated to be 0. We clear the char memory at (0,0) first by setting it to 0. We keep increment y counter for a given x so that a column is cleared. At end of each column, we increment the x counter by 1 so we start to clear a new column. VGA_clear_pixelbuff_ASM() uses the same logic but the memory cleared is the pixel buffer. For VGA_write_char_ASM(int x, int y, char c), we first check if x is between 0 and 79 and y is between 0 to 59, if it is outside the range, simply branch back to the link register. If x and y are within the range, we calculate the final address by adding the base address to x and y which is shifted 7 position. We then store the third argument, the char into R5, which is now pointing to the address corresponds to (x, y). VGA_write_byte_ASM(int x, int y, char byte) uses the same logic as write_char before. A byte is 8 bits long so we can write two chars to the screen and a space between bytes. Now the issue is to use ASCII code so that it can be displayed. We initiated ASCII constant that shows the corresponding ASCII code of the hexadecimal number so that we can just convert to ASCII code easily. The last one VGA_draw_point_ASM(int x, int y, short colour) uses the same logic as other write functions but this time is the pixel stored in memory.

Simple VGA Application

In this simple c application, we call the three assembly functions written before in three different methods named test_char, test_byte, and test_pixel, which would let the whole screen display characters, bytes, and color pixels, respectively. In the main method, we first enter an infinite loop. Inside the loop we check the press button as well as the slider switches if needed, and call the corresponding functions as described in the lab manual.

The challenge we met in this part was about logic shifting the counters to the correct place. Also when incrementing the counters, we need to shift them back. Therefore, we write the ADD and LSL in the same line to make sure that the counter doesn't change after this.

2 Keyboard

In this part, we made use of the PS/2 keyboard to control the display. There is only one function in this program named read_PS2_data_ASM. When calling this function, the scan code corresponding to the key that is pressed will be shown on the display. The header file contains this function, which takes a char pointer variable as the argument and stores the value at the address.

In the assembly file named ps2_keyboard.s, there is one subroutine, read_PS2_data_ASM. We first load the address of PS2_DATA and get the value in this address. Since we only consider the bit at 15 in the address of PS2_DATA to determine whether it's valid or not, there is a constant value of 0x8000 for us to get the value at that bit. After that we AND the value in the address with the constant value. If the result is zero, which means the bit at 15 is zero, the data read is invalid so the subroutine returns 0. Otherwise, the data read is valid, so we store the data to R0 and return 1 as described in the instruction. Note that here the Data is in bits 0-8, so we need to AND the value in PS2_DATA with #11111111 before storing it.

In the C file of this program, we first clear all the displays on the screen by calling VGA_clear_charbuff_ASM() and VGA_clear_pixelbuff_ASM() that we used before to clear the characters and the pixels. Then we initialize two variables x and y for the horizontal and vertical display, respectively. After that we enter an infinite while loop to start displaying. If the data read is valid, or equivalently, the method read_PS2_data_ASM(&ch) returns 1, then we call VGA_write_byte_ASM method with the arguments x, y, and ch to display the scan code on the screen. We need to increase x by the gap of 3 after this because we need to shift to the new place where the next byte is to be displayed. However, we also need to consider whether a row is full or not. Specifically, if x reaches 80, which means the row is full, we need to move on to the next row by setting x to 0 again and increase y by 1. Whenever y reaches 60, the screen is full, so we clear the screen by calling VGA_clear_charbuff_ASM().

The challenge we met in this part was about understanding the usage of each bit inside the PS2_Data address, and we also needed to check that we took the correct bits corresponding to RVALID or Data.

3 Audio

For this part, we need to write a program that could generate a 100 Hz square wave audio. Again in the header file, we need only one method named write_audio_ASM that takes one integer argument. The corresponding subroutine should take this integer argument and write it to both the left and the right FIFO only if there is space in both the FIFOs.

In the assembly file we first load the data in FIFO_SPACE to R1 and load the address of FIFO_LEFT as well as FIFO_RIGHT. Then we need to get the value of WSRC and WSLC. To get WSRC, we shift R1 by 16 bits and then load the last byte of what is left, because WSRC lies in bits 16-23. To get WSLC, we shift R1 by 8 bits again and then load the last byte, because WSLC lies in bits 24-31. There is no space if either WSRC or WSLC is zero, so in the branch NONSPACE, we return 0. Otherwise, the integer argument R0 is stored in both right and left FIFO address and then we return 1.

In the C file for this part, we first need calculate the number of samples for each half cycle of the square wave. With a sampling rate of 48K samples/sec and a frequency of 100 Hz, the sampe number for each half cycle would be 48K / (2*100), which is 240. Therefore we initialize an integer i to be 0 and would increment as a counter. Then we write 0x00FFFFFF to the FIFO for 240 times and also write 0x00000000 for 240 times. Whenever the space is full, we decrement the counter by 1.

4 Possible Improvements

We found two possible improvements for this lab. The first one is for the subroutines of the VGA. We pushed a lot of registers at the beginning of the subroutine, including some that we didn’t actually use in the program. We could only push the used registers so that we don’t need to use a lot of spaces in the stack.

The second one is for the C program of the audio part. We simply used an infinite while loop and write the two values to FIFO to control the audio. It is possible that we make use of the TIM program to control it, but the code would be more difficult.