High Level I/O - VGA, PS/2, Keyboard, and Audio - Laboratory #4 Report

ECSE 324 - Computer Organization

November 7th, 2018

Sarah Swanson - 260660847
Bryan Jay - 260738764
Group 19

Introduction

The purpose of this laboratory is to introduce the high level I/O capabilities of the DE1-SoC computer. The tasks which we are challenge to complete are: use the VGA controller to display pixels and characters, use the PS/2 port to accept input from a keyboard, and use the audio controller to play generated tones.

Part 1 - VGA

The purpose of part 1 of the laboratory was to use the VGA controller to display pixels and characters. The data that is displayed onto the screen comes from the FPGA on-chip memory sections: the pixel buffer and character buffer.

We were challenged to write five subroutines that make the display function. We were given VGA_clear_pixelbuff_ASM and VGA_draw_point_ASM and were told to write VGA_clear_charbuff_ASM, VGA_write_char_ASM(int x, int y, char c) and VGA_write_byte_ASM(int x, int y, char byte). For the first subroutine, VGA_clear_charbuff_ASM, we implemented this to be very similar to VGA_clear_pixelbuff_ASM but it clears all memory locations in the character buffer and the ‘y’ starts at the 7th bit for the char character. We also had to implement VGA_write_char_ASM(int x, int y, char c) which writes ASCII code that is passed into the third argument at the coordinates (x,y). First we checked if the coordinates were in range and if not, leave the subroutine. Then the subroutine will store the value in the location given in the array. Lastly we wrote VGA_write_byte_ASM(int x, int y, char byte) which takes two address points since it is a byte. It writes the hexadecimal version of the value on the screen. This prints two characters at the coordinates x,y on the screen. We checked if the coordinates were in range first, and exited the subroutine if not. Then we stored the value into the given location in the array.

We were also challenged to build a C based application in order to test the functionality of the VGA driver’s subroutines. We had three subroutines we had to test. The first was if pushbutton 0 was pressed, then check if any of the slider switches were on and if yes, call the test_byte() function which executes the VGA_write_byte_ASM subroutine. Otherwise, call the test_char() function which executes the VGA_write_char_ASM subroutine. We implemented this using a while loop with if statements. The second was if the pushbutton 1 was pressed, then call the test_pixel() function which executed the VGA_draw_point_ASM subroutine for the x and y’s in range [0,319] and [0,239]. This was implemented in the same while loop using an else if statement. The third was if the third pushbutton was pressed, then clear the character buffer by calling the VGA_clear_charbuff_ASM subroutine. This was also placed in the

while loop with an else if statement. Finally if fourth pushbutton was pressed, clear the pixel buffer by calling VGA_clear_pixelbuff_ASM subroutine. Again, an else if was used in the while loop to continuously check if the buttons were pressed.

A challenge faced while doing this laboratory was integrating the subroutines together and taking the code from laboratory three and integrating it all together. Besides integrating all the code, the way we implemented the subroutines and application pretty simple therefore there are no improvements.

Part 2 - Keyboard

The purpose of part 2 of the laboratory was to use the PS/2 port to accept input from a keyboard. The PS/2 bus is what provides data about the keystroke events. It does this by sending hexadecimal numbers (called scan codes) which vary from 1-3 bytes in length. The make code is a scan code that is sent when a key on the PS/2 keyboard is pressed. The break code is a scan code that is sent when the key is released. Typematic delay is the initial delay after the first make code is sent, and is then continuously sent. The typematic rate is the rate at which the make codes are sent after the initial delay between the first two make codes.

We were challenged to implement a subroutine read_PS2_data_ASM that checks the RVALID bit in the PS/2 data register. If valid, data from the same register is stored at the address in the char pointer argument and the subroutine returns 1 to denote valid data. If RVALID is not set, subroutine will return 0. We implemented this by checking first if RVALID bit is zero. If zero, then return 0 and branch to end. If not, continue. Then we got the data from the first 8-bits and stored the value into the given address and returned 1.

We were then challenged to write an application that reads raw data from the keyboard and displays it to the screen only if valid. We had to keep track of the x,y coordinates where the byte was being written. A gap of 3 x coordinates were given because each byte displays two characters and one space between each byte. We implemented this application by using the VGA_write_byte_ASM subroutine. We checked if there was an input from the keyboard and if yes return 1, else return 0. If the data was not empty, we wrote the value to the display using the VGA_write_btye_ASM subroutine. We then increment the index by 3 in order to write the next value onto the screen. When the index was completed for a row, resent the x and increment the y in order to go to the next row. When the index reached the end of the array, we reset all values and cleared the display using the VGA_clear_charbuff_ASM() subroutine.

A challenge faced in the part of the laboratory was understanding how the make and break code works and to be able to handle this within our code and our main method. An improvement could have been to have cleaner code because our comments are mixed in between the code. Besides that, our code was simple and could not be made simpler to our knowledge.

Part 3 - Audio

The purpose of part 3 of the laboratory was to use the audio controller to play generated tones. We were challenged to write a driver for the audio port following the same procedure already introduced. The subroutine AUDIO_write_ASM took one integer argument and wrote it to both the left and right FIFO only if there was space in both FIFOs. To implement this, we loaded all of the audio data first. Then if the bit is filled in the left FIFO (WLSC), return 0. If in the right FIFO (WSRC), return 0. This was to show no data was written to the FIFOs. We then stored the signal into the right and left data spots in the registers and returned 1 to say data was written in FIFO. Either 1 (0x00FFFFFF) or 0 (0x00000000) would indicate the signal depending if data was written in the FIFO.

We were also challenged to find the sampling rate from the manual and to calculate the number of samples for each half cycle of the square wave. To do this we had to find the number of samples that were required for a 100Hz (100 cycles/sec) square wave. We found (in the De1-SoC computer manual) that you can find this by dividing 48,000 samples/sec by 100 cycles/sec to result in needing 480 samples/sec. For samples 0-239, the signal was 1 and for samples greater than 239, the signal was toggled from 0-1.

To play the 100Hz square wave on the audio out port, we wrote a program that checks if the board received the signal using the subroutine AUDIO_write_ASM(signal) which plays the 100Hz square wave on the audio out port if a signal was received. Then increment the sample. If the sample reached 240, reset it because it was at the end. If the signal was 0 before, restore the signal to 1. Otherwise, put the signal to 0 if there was 1 before.

A challenge in this part of the laboratory was finding the number of samples. The De1-SoC computer manual was helpful for this. An improvement for this part could have been to have a routine that checks the WSLC and WSRC bits at the same time. This would cause the subroutine to be less repetitive.