Carlo D'Angelo 260803454
Mostafa Abdelwahab 260551876
March 29th, 2019
ECSE 324 – Computer Organization

# Lab 4: High level I/O - VGA, PS/2 Keyboard, and Audio

## *VGA*

VGA_clear_charbuff_ASM is a subroutine that clears the data from the VGA character buffer. VGA has 80*60 registers, each used to display a character depending on the value stored in the register (corresponding to the ASCII code of the character). VGA_clear_charbuff_ASM goes through each of those registers and set their values to zero. To do so, we begin from the character buffer base address, and then clear the registers row by row. We first set the address value with a specific y increment to the base address (the y coordinate has to be left shifted 7 bits first then we "ORR" to the base address") then we go to the addresses withal possible x values (0 to 79- ORRing it to the address we have). We then store 0 at the register with obtained address.

VGA_clear_pixel_ASM is a similar subroutine to the previous one, for the pixels of the VGA. It uses the same logic to go through the 320*240 registers and sets the value in each register to zero.

VGA_write_char_ASM is a subroutine that takes 3 arguments. The first 2 (int x and y ) should be within the accepted ranges for the VGA character coordinates [0;79] and [0;59] respectively, otherwise the subroutine returns to the caller without writing any values. It then goes to the corresponding register by using the ORR instruction to get the specific address from the character buffer base address, the x coordinate, and the y coordinate (the y coordinate has to be left shifted 7 bits first). Once we have the desired address, we write to it the values of the third argument (the character ASCII code). This will display the specified character at the specified coordinates on the screen

VGA_write_byte_ASM is a subroutine that prints 2 consecutive characters on the screen starting at the specified coordinates. It takes 3 arguments. The first 2 (int x and y ) should be within the

accepted ranges for the VGA character coordinates [0;79] and [0;59] respectively and (x,y)≠(79,59), otherwise the subroutine returns to the caller without writing any values. It then goes to the corresponding register (using the same logic as VGA_write_char_ASM ). It then prints a second character at the following coordinate (same row, column to the right or lower row, first column if first character was at last column).

Each of the 2 characters is one of the following set [0..9,A..F]. The third argument is a byte (8 bits). We isolate the higher 4 bits (right shift 4 bits) then write the ASCII code to the first character address depending on the hexadecimal representation of these 4 bits. We do the same to the lower 4 bits to the second character address.

VGA_draw_point_ASM takes 3 arguments. The first 2 (int x and y ) should be within the accepted ranges for the VGA pixel coordinates [0;319] and [0;239] ), otherwise the subroutine returns to the caller without writing any values. It then goes to the corresponding register by using the ORR instruction to get the specific address from the character buffer base address, the x coordinate(left shifted 1 bit first), and the y coordinate (the y coordinate has to be left shifted 10 bits first). It then stores to this address, the value of the third argument: a short representing the color to be draw at this pixel.

## *P/2 Keyboard*

The read_PS2_data_ASM subroutine accesses the PS/2 port (0xFF200100 base memory address) and reads the RVALID bit in the PS2_Data register to check the validity of the data in its *Data* field. If this data is valid (i.e., RVALID bit is 1), then it is stored at the address in the char pointer argument, and the subroutine returns 1 to denote valid data. If the RVALID bit is not set (i.e., is 0), then the subroutine simply returns 0.

Since the PS/2 bus provides data about keystroke events by sending hexadecimal numbers called *scan codes*, the data must be stored in a queue (i.e., FIFO). This fact is important because, when the RVALID bit is 1 and we read the from the PS2_Data register, it provides the data at the head of this FIFO in the *Data* field.

In this part of the lab, we must write the data stored at the address in the char pointer argument to the VGA by using the VGA_write_byte_ASM subroutine. In other words, when a key on the PS/2 keyboard is pressed, the VGA displays the key's corresponding *make code* and *break code*.

### *Audio*

The play_audio subroutine accesses the Audio In/Out Port port (0xFF203040 base memory address) and checks the *WSLC* and *WSRC* fields in the Fifospace register to see if there is space in the left and right audio-out FIFOs. The subroutine takes one integer argument and, if the data stored in the *WSLC* and *WSRC* fields are not 0, writes this integer to both the left and the write FIFOs (by writing to the Leftdata and Rightdata registers respectively). The subroutine also returns an integer value of 1 if the data was written to the FIFOs, and returns 0 otherwise.

This part of the lab plays a 100 Hz square wave on the audio out port by writing 0x00FFFFFF (i.e., high pitch) and 0x0000000 (i.e., low pitch) to the audio-out FIFOs using the play_out subroutine. Since the default setting for the sample rate provided by the audio CODEC is 48K samples/sec, we had to write the high pitch to the FIFOs 240 times and write the low pitch to the FIFOs 240 times (and repeat this process 100 times in that order) because 100 complete cycles of the wave had to be contained in 48K samples (i.e., 100 Hz square wave).