ECSE 324 – Computer Organization (Fall 2018)
Lab 4: High level I/O - VGA, PS/2 Keyboard, and Audio
Group 12: Alain Daccache-260714615 ; Michel Abdel Nour-260725050

Part I. VGA:

This part consisted of implementing the subroutines defined in the header file.

Pixel Buffer:

VGA Clear Pixel: We iterate through all the possible (x, y) combinations, making sure we stay in the range [0, 319] for x and [0, 239] for y. Therefore, we implement a for loop inside of a for loop i.e. for each x from 0 to 319 { for each y from 0 to 239 } in order to iterate through all the values of y for each x.

Every pixel has a unique address where to store the corresponding data. The address is assembled by using a base address and the x and y coordinates. To do so, we know that the base address of the pixel buffer is 0xC8000000. Then, the x coordinate is shifted left by 1 bit, and the y coordinate is shifted left by 10 bits because the address for x starts after the first bit, and the address of y starts after the 10th bit. After shifting x's value, we store it in a register, and then add that register with the shifted y's value into a new register. Then we can add the base address and the (x, y) offset to get the correct address of the pixel.

To clear, we simply move into a register the immediate value #0 then store it in the contents pointed to by the address that we computed earlier.

VGA Draw Point: This part mainly consisted of storing the colours into registers. We needed to consider the correct offsets to make space for the X and Y bits After we shifted the X bits by one bit to make space for the offset and the Y bits by 10 for the same reason, we were able to write the colour at the address.

Character Buffer:

VGA Clear Char: A similar approach than the VGA Clear Pixel subroutine, but a different base address was used (0xC9000000) and a different (x, y) offset was used (no shift for x, and left shift by 7 for y instead of 10). Also, the range of x coordinates is [0, 79] and that of y is [0, 59]

VGA Write Char: First, we take care of the edge cases by using compare operations to make sure the input for the x and y coordinates are within the range of acceptable values (between 0 and 79 for x and 0 and 59 for y). Then, we shift left y by 7 bits and add to x then to the base address. Finally we store the value inputted (R2) into the computed address.

VGA Write Byte: This part was similar to Write Char, The only difference is that a byte is equivalent to 2 characters, so we had to adjust the input registers accordingly. The subroutine was called twice to fill each 4 char.
We used a lot of the code we wrote from before.

Simple VGA Application

After writing all the subroutines, we needed to incorporate them into the C program to test the VGA driver.

This code consisted of 4 if statements, one for each button pressed (1,2,3,4). The code would simply state that if read Push Button Data was equal to one of these numbers (in binary one-hot encoded) the code should do something as stated in the manual. For example, if the PB data = 1, we were to write a code that if slider switches were 0 then we would call the test char subroutine written previously, else we would call the test byte subroutine. This consisted of a while loop, a nested if statement and an else statement and was simple to write. For the other 3 statements the code would simply call the test pixel, clear char buffer, clear pixel buffer subroutines respectively. No issues faced here.

Challenges and Improvements:

- There are a lot of repetitive parts among the subroutines. For instance, we always generate an address offset (both pixel and char buffers) by either taking an input from the user (write, draw) or using hardcoded combinations of x and y (clear), then shifting appropriately and adding to the base address. We can implement a subroutine that returns the address taking arguments for x, y, and whether they are coordinates for working with pixels or chars.
- Both VGA Clear Char and Pixel are similar in nature. The challenge was in implementing a for loop inside of another for loop, until we noticed that we can add the conditions for loop 1 after the conditions for loop 2, as follows [for clear pixel]: ADD R1, R1, #1, CMP R1, #240, BLT PIXEL_LOOPY, ADD R0, R0, #1, CMP R0, #320, BLT PIXEL_LOOPX

Part II. Keyboard

The PS/2 port of the FPGA uses the PS2_Data register. We used the following fields of the register

- The Data field holds the key pressed on the keyboard. Specifically, when a key is pressed on the keyboard, a make code is created and stored in a FIFO register. When the key is released, a break code is stored.
- The RVALID bit, which, if set to 1, reading from this register provides the data at the head of the FIFO in the Data field (i.e. the last 8 bits), and decrements RVALID by 1.

The function *read_PS2_data_ASM(data)* passes the argument data, a char pointer, in which the data that is read will be stored. We use a pointer in order to “pass by reference” and save the data from the Data field after the function returns. The approach is fairly simple:

- In the subroutine, we check the status of the RVALID bit. To do so, we load the PS2_Data register (which contains RVALID), then we AND it with 0x8000 in order to get the bit at the 15th position (RVALID). We return this value in R0.
- Finally, we get the Data field by performing an AND operation with 0xFF to get the last 8 bits (where the key to be read resides), and store that data in R0.

main.c application: We continuously call the read_PS2_data_ASM function (polling) until it returns 1. Then, we call VGA_write_byte_ASM(x,y, *data) [x and y initialized at 0] in order to display the data read from the keyboard. Because we are writing bytes, the x coordinate has to be incremented by 3 instead of 1 since each byte will display two characters, and one more for a space between each byte. When we reach the end of a line (i.e. x > 79), we reset x and increment y, in order to start at a new line.

Challenges and Improvements:

- The previous lab gave us enough experience manipulating bits of a register to get the value at some meaningful bit(s). Therefore, we did not incur any major challenge. The description of what was required from us (the subroutine and the main function) was very clear and provided the logic behind implementing the functions.
- No improvements for this part as we have hardware constraints (PS/2 port) that has functionality that will be used for similar purposes (display key typed, on screen). We are simply accessing the RVALID bit, storing the value of the Data field in the contents of a char pointer passed by reference, then displaying it to the screen by using an already implemented VGA subroutine.

Part III. Audio

This part is similar to Part II of the lab. Registers hold the output before outputting the data. We keep track of how much space there is in FIFO using another register that monitors the amount of space left. Left fifo and right fifo correspond to the the left and right audio channels L_FIFO and R_FIFO.
The amount of space remaining in fifo is stored in two 8-bit sections of the fifo-space (FIFO). They are labelled WSLC and WSRC (Left and Right respectively).

In audio.s, we keep the value in WSLC and WSRC. If there is space, we write values in both registers.
In main.c, we had to use the default sampling rate of the CODEC,found in the DE1-SoC manual to be 48K sample/sec and determine what rate we had to write values to the registers to output a 100Hz audio signal, a square wave. 100Hz = 100 /sec and the CODEC samples at 48K/sec. From the lab manual we were told that if the codec sampled at 100 samples/sec and we want a 2Hz = 2/sec signal then two full cycles would need to occur in 100 samples. That is 50 samples per cycle therefore 25 are ‘hi’ and 25 are ‘lo’. To get our desired 100Hz signal at 48K samples/sec we do the same math. 48K/100 = 480 samples per cycle. Therefore 240 ‘hi’ and 240 ‘lo’. Our C code accomplishes this by running two loops counting up to 240 that write 0x00FFFFFF, ‘hi’, and 0x00000000, ‘lo’. Each loop they will check whether there is space, if there is then the write

will be successful, otherwise try again by reducing the counter and trying to write. The loops check whether a write was successful depending on the value returned by audio_write_ASM(), a 1 indicates a successful write, and a 0 indicates a failed write.

Challenges and Improvements: The main challenge face was checking whether or not WSRC and WSLC were full. We tried writing them using offset addresses but the debugger would freeze when activated. Eventually, we rewrote it to use ‘load byte’ and ‘compare’ and wrote to the left and right data registers using their exact addresses which worked.