1.0 VGA DRIVER

1.1 VGA Clear Char & Clear Pixel

The objective of this part of the lab was to write two subroutines which would set all the memory locations in the character buffer and the pixel buffer to 0. To do this, two loops were used, iterating through the x and y axis of the screen. For the clear char subroutine, we were to iterate the x axis from 0-79 and the y from 0-59. For the pixel we were to iterate through the x axis from 0-319 and the y axis from 0-239. Every pixel or char of the 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. For the clear char subroutine, the y coordinate is shifted to the left by 7 bits and "ORRed" with the x coordinate then added to the base address. For the clear pixel, the y coordinate is shifted left by 10 bits and the x coordinate is shifted left by 1 bit.
This part of the lab was fairly simple and straightforward, and both clear pixel and clear char were extremely similar to each other. No challenges were faced when writing this code.

1.2 VGA Write Char

This part of the lab was also fairly simple to write, to start with registers were pushed on to the stack for future use. Two registers were used to keep track of the length and width of the screen in terms of pixels (x=80 and y=60). Firstly, we would use the CMP operation to compare current X, Y (R0,R1) to check if they are eligible values, basically in between 0-80 and 0-60 respectively. Then we would offset y by 7 bits, in order to make room for x for when they are added together. Finally we would store the value inputted into the address allowing to write a char. No major challenges were faced here other than simple debugging due to mis use of ARM instructions.

1.3 VGA Write Byte

This part of the lab was fairly identical to Write char, however as a byte is 2 chars, we used the write char subroutine twice for the write byte method and adjusted the input registers accordingly. Each of the calls to write char writes 4 bits from the input. This subroutine was fairly simple due to the ability to reuse the existing code.

1.4 VGA Draw Byte

This part of the lab was also fairly simple, firstly, the colour was stored into a register. Later, the Y(value which determine Y coordinate on screen) was offset by 10 to make space for the X bits, and the X bits were shifted by one bit to make space for the offset value as shown in the computer manual. The shifted Y and X were then added to the offset, and finally added to the address. Finally the colour was stored at the address.

2.0 Simple VGA Application

Finally, we were to write a simple c program which will test the functionality of our 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) the code should do something as stated in the manual. For 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 extremely 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.

3.0 Keyboard

The keyboard component of this lab uses the PS/2 port of the FPGA. A lot of the functionality is provided by the FPGA registers. There are two registers that are used to implement the PS/2 port, since we do not use interrupts for this lab we only use one of the registers, PS2_Data. When a key is pressed on the keyboard a make code is created a stored to a FIFO register and when the key is released a break code is stored. The ARM code checks whether the bit 15, RVALID, of the PS2_Data register is set to 1, which indicates a that there is data to be read from the head of the FIFO stack. The head of the FIFO stack is pointed to by the last 8 bits of the PS2_Data register, called Data, and reading from here decrements RVALID by 1.

Our implementation loads the PS2_Data register and checks the status of RVALID, and if it is 1 reads the data at the last 8 bits of the register. This data is simply stored as a byte at the pointer passed to the subroutine. The ARM code returns 1 when there is a successful read of the PS2_Data register and returns 0 otherwise (when RVALID is 0).

The C code for the keyboard is simple and similar to the C code for VGA_write_byte_ASM(). It works on a polling method that continuously attempts to read the data from the keyboard into a variable called ‘value’ when it succeeds the code calls the VGA_write_byte_ASM() subroutine with the value read and the next x and y coordinates. Because we are writing bytes the x coordinate has to be incremented by 3 instead of 1. When the x and y coordinates get to the bottom right of the screen the entire screen is cleared. Our C implementation is a little different from normal in that we create a char variable called value and then pass its address. This is instead of creating a char pointer and passing that, then writing the dereferenced pointer. We believe this is a clearer implementation.

It was a very straightforward implementation for this section due to the fact that we could utilize out working VGA_clear_charbuff_ASM() and VGA_write_byte_ASM() subroutines. The only challenges were that once again our debugger would freeze. Luckily the read_PS2_data_ASM() subroutine was easy enough to implement.

4.0 Audio

The audio component of this lab is very similar to the keyboard section. It utilizes registers to hold the data being output before actually outputting the data. The registers that hold the input/output data has another register that stores how much space there is. The two registers that hold the output data correspond to the left and right audio channels and are thusly named

Leftdata and Rightdata.The amount of remaining space in the output registers is stored in two 8-bit sections of the Fifospace register labelled WSLC and WSRC, respectively.

Our task for the ARM code for this section was to check the values in WSRC and WSLC, and if both represent free space in the Leftdata and Rightdata registers then write a value to both of the data registers. For the C code, 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 ‘high’ and 25 are ‘low’. To get our desired 100Hz signal at 48K samples/sec we do the same math. 48K/100 = 480 samples per cycle. Therefore 240 ‘high’ and 240 ‘low’. Our C code accomplishes this by running two loops counting up to 240 that write 0x00FFFFFF, ‘high’, and 0x00000000, ‘low’. 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 write_audio_data_ASM(), a 1 indicates a successful write, and a 0 indicates a failed write.

The challenges we faced were trying to read whether WSRC and WSLC were full. We originally tried to read them loading the register and shifting right by 16 bits to get rid of the RARC and RALC bits, then comparing the remaining value to a constant. We would then write to the two registers by using offset addresses. It was not clear which part of that did not work as our debugger would always freeze when activated. Then we tried to check each of WSRC and WSLC individually by ANDing them with a constant. That also did not work, it was not clear why even after consultation with other lab groups and the TA who all believed it should work. 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.