

ECSE 324

Computer Organization

Lab 4

**Authors:**

Tyrone Wong, (260601565)

Yian Cheng , (260873155)

Odero Otieno, (260664284)

## Introduction

As a follow up from the previous lab, we will be further exploring the I/O capabilities of the DE1-SoC computer.We will be writing high level I/O programs that involve the VGA display controller, the Ps/2 Keyboard and an Audio controller. We will be able to see how to use C and Assembly to interact with the devices and determine how to receive data and output data.

## VGA

Description

In this lab component, we will be exploring the built in VGA controller located on the De1-Soc computer. The display functionality is separated into 2 components, a pixel buffer and a character buffer, both are on chip memory sections on the FPGA chip. The pixel buffer is responsible for displaying colors and the character buffer will store characters to display using ASCII. We were tasked with completing 2 types of functions, one type to write out on the display by storing into the buffers and the other to clear the memory locations for each buffer. We wrote 3 writing functions:

**VGA\_write\_char\_ASM(x,y,c++):** Displays a character given and writes it to the char buffer

**VGA\_write\_byte\_ASM(x,y,c++):** Displays a set of bytes given and writes it to the char buffer to display the bytes in hexadecimal form

**VGA\_draw\_point\_ASM(x,y, colour++):** Displays the color specified in terms of Hexadecimal RGB

And to clear the displays we wrote 2 similar functions:

**VGA\_clear\_charbuff\_ASM():** Will clear the values stored in the charbuffer memory space

**VGA\_clear\_pixelbuff\_ASM():** Will clear the values stored in the pixelbuffer memory space

The Approach taken

To test the functionality of our writing and clearing of the 2 buffer memory locations, we wrote a main method that involves a continuous loop to see what input button was pressed and adjusted what was displayed if a switch button was activated. If a writing button was pressed, we would check the slider switch to determine which writing function to execute. Once determined,we would then execute a nested forloop to iterate through the entire x and x plane and store either a byte or char at that corresponding location. To display at either memory location, it was simply a matter of specifying the right address and shifting the input coordinates to match the memory format specified by the manual.

i.e) To write at position x,y = (0,1) to the pixel buffer, indicate base address 0xC8000000 and then offset by a specified position to specify the 0,1 coordinate in the pixel buffer address location and write to the content of that location. In the case of 0,1 the address offset would be (00000001 000000000 0) so we would take argument y and shift it by 10 bits and add it to the base address to get 0xC8000400.

The approach from char and byte are almost identical, they only vary in initial base address, shifting and slight alteration in coordinate handling (i.e saving 2 adjacent x locations for byte display and then skipping a spot vs filling up all x spots with char).

If the user clicked a clear button in our main method, we would initiate a clear method depending on the button chosen. To clear the display, we iterated through each coordinate starting off by clearing a y column and then incrementing the x row. Both methods for byte and char are very similar and involve the same algorithm of shifting the coordinates correctly to specify the correct memory location corresponding with the coordinate. Both methods just involved resetting the value after going through each memory location specified by covering all x and y values.

Challenges Faced

The largest challenge faced was the memory manipulation and the conversion of the input char or bytes. We had to pay a lot of attention to the data and how to convert it via ASCII or keep track of what data size we were using, half words or bytes. Additionally, we spent a large amount of time debugging the code that was functional but seemed unfunctional due to the screen settings! i.e) Chars weren’t showing up when stored at the initial address due to the chars appearing off screen. Note for future: Specify to reset the screen to factory settings.

Improvements

I think that we can potentially make the clear functions into 1 singular method. We would have that method receive arguments for which type to clear (byte or char) and we would indicate the boundaries of the x and y coordinates. This way we would have a reusable clearing method for both buffers instead of having 2 very similar methods.

## PS/2 Keyboard

Description

In this component, we will be exploring the use of a keyboard that will store information and how we will be collecting information received from the keyboard in a FIFO (First In, First Out) manner. With this program, we created a main method that constantly checks for for keyboard input and would display a byte on the screen representing when the key was pressed and when it was released.

Approach Taken

The approach we took was very similar to the vga method. We made use of a continuous while loop to constantly check for new inputs from the keyboard. To read the keyboard input, we take the PS2 address and we retrieve the MSB at memory position 15-16 to verify the R-Valid memory location. If the location is a valid, we take the input data and we insert it into the memory location of R0 (Our argument from the main method that is the char pointer). Then we set the R0 register to be either 0 or 1 to indicate if the input is valid or not. If the input was valid, the program would use the write\_vga\_byte method to display the character and then increment x (horizontally by 3 to leave a space in between 2 byte chars) and y accordingly.

Challenges Faced

The challenges faced with the keyboard component was setting up the main method with the pointer. Manipulating C pointers in the c program and then writing the registers according to the input was a bit confusing. We spent a lot of time debugging and checking the memory content to see what was being stored, which register was the pointer, and whether we were dealing with a value or an address.

Improvements

The way we wrote our code is very easy to understand but not entirely efficient. We used more registers than necessary just to make it clear so that we could uniquely identify content according to the registers. This was because we had trouble distinguishing between addresses and content. An improvement would be to optimize the register use and recycle any registers that have no relevant value in them.

**3.**​ ​**Audio**

*Our Approach*

We were required us to write​ ​a​ ​driver​ ​consisting ​of​ ​a​ ​single​ ​routine​ for​ ​the​ ​audio​ ​port.​ ​For the audio port registers, we needed to write data​ ​to​ ​the​ ​leftdata​ ​and​ ​rightdata​​. ​

Reading​ ​the​ ​value​ ​of​ ​the​ ​fifospace​ ​register is the first step. The fifospace register ​is​ ​at​ ​an​ ​offset​ ​of​ ​0x4​ ​from​ ​the​ ​control​ ​register​ ​of​ ​the​ ​audio​ ​port.​ ​To​ ​verify​ ​if​ ​it​ ​can write​ ​data​ ​to​ ​the​ ​data​ ​registers,​ ​it​ ​reads​ ​the​ ​WSLC​ ​and​ ​WSRC​ ​bits​ ​of​ ​the​ ​fifospace​ ​register. They​ ​contain​ ​the​ ​number​ ​of​ ​words​ ​available on the left and on the right respectively.

We then applied​ ​masks​ ​with​ ​values​ ​0xFF000000 and​ ​0x00FF0000​ ​respectively​ ​to​ ​the​ ​fifospace​ ​value.​ ​If​ ​at​ ​least​ ​one​ ​of​ ​these​ ​two​ ​numbers​ ​is 0,​ ​then​ ​there​ ​is​ ​no​ ​space​ ​for​ ​the​ ​data​ ​and​ ​the​ ​routine​ ​returns​ ​0,​ ​meaning​ ​it​ ​failed​ ​to​ ​write​ ​the data​ ​to​ ​the​ ​audio​ ​port.​ ​Else​ ​if​ ​there​ ​is​ ​space​ ​for​ ​the​ ​data,​ ​the​ ​routine​ ​stores​ ​the​ ​input​ ​value​ ​in the​ ​leftdata​ ​and​ ​rightdata​ ​registers,​ ​which​ ​are​ ​at​ ​an​ ​offset​ ​of​ ​0xC​ ​and​ ​0x8​ ​respectively​ ​of​ ​the control​ ​register​ ​and​ ​returns​ ​1,​ ​meaning​ ​it​ ​successfully​ ​wrote​ ​the​ ​data​ ​to​ ​the​ ​audio​ ​port.

We​ then ​wrote​ ​a​ ​C​ ​program​ ​that​ ​plays​ ​a​ ​100Hz​ ​square​ ​wave.​ ​The​ ​C​ ​program​ ​calculates​ ​the number of ​samples​ ​it​ ​should​ ​switch​ ​between low and high values according to the formula: *number of samples* = *sampling rate* / (2 \* *desired frequency*) .​ ​Then​ ​it​ ​looped​ ​forever,​ ​every iteration​ ​trying​ ​to​ ​write​ ​data​ ​to​ ​the​ ​audio​ ​port.​ ​If​ ​it​ ​succeeded,​ ​then​ ​it​ ​would​ ​update​ ​its​ ​sample

count​ ​and​ ​if​ ​this​ ​count​ ​reached​ ​the​ ​computed​ ​maximum​ ​count,​ ​then​ ​it​ ​would​ ​reset​ ​the sampling​ ​count​ ​and​ ​change​ ​the​ ​audio​ ​value​ ​from​ ​low​ ​to​ ​high​ ​or​ ​high​ ​to​ ​low.

*Some Challenges We Faced*

​We had some challenges with calculating​ ​the number​ ​of​ ​samples​ ​we​ ​needed​ ​to​ ​write​ ​before​ ​changing​ ​value​ ​from​ ​low​ ​to​ ​high​ ​or​ ​high​ ​to low.​ ​Dividing​ ​the​ ​sampling​ ​rate​ ​by​ ​the​ ​desired​ ​frequency​ ​gave​ ​us​ ​the​ ​number of​ ​sample​ ​needed​ ​for​ ​a​ ​whole​ ​period after which ​we​ ​only​ ​had​ ​to​ ​divide​ ​this​ ​result​ ​by​ ​two​ ​to​ ​get​ ​the number​ ​of​ ​sample​ ​for​ ​each​ ​half​ ​period.

*Possible Improvements*

Despite facing some challenges in this part, figured out that we could extend this driver to play ​sine or​ ​sawtooth​ ​waves also.​ ​We noticed that while for the square wave we have to change the value at ​every​ half ​sample,​ the improvement would​ require ​changing​ ​the​ ​value​ ​of​ ​the​ ​data​ ​we​ ​write​ ​to the​ ​audio​ ​port​ ​at every half period.