**ECSE 324 Lab 3 Report**

*Note: For my part 2 demo, due to Zoom recording latency, in the video, when I type multiple keys, it looks like their scan codes get outputted at the same time. That didn’t happen in real-time. You can try it yourself if you are interested.*

*Note: I had to re-record part 3. In the original cut, my flags loaded when I was doing the demo. But, the Zoom recording had some latency so in the video it looked like it failed to load. I’ve stitched my old part 1 and 2 with my new part 3.*

**Part 1: Drawing things in VGA**

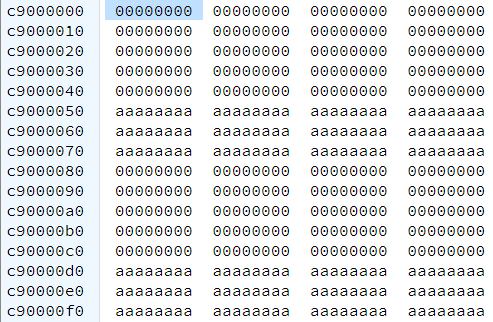
Approach

Before writing the functions, I located the addresses for the pixel and character buffers. This time, I tried renaming registers R0-3 with x, y, color/char, and mem with the “.req” directive. It made coding a lot more clearer, since in the past, I had to rely on looking at the comments to know what the registers meant. I was efficient with my registers, using only R0-3 in my functions, so I didn’t need to push and pop. I also didn’t need to push LR, since my functions don’t call other functions (input\_loop does call my functions but it does push LR). If it caused part 2 and 3’s code to malfunction, I would have included it, but they didn’t malfunction.

For drawing on the VGA display, I simply wrote the color half-word onto the right memory address: PIXEL\_BUFF | (y << 10) | (x << 1). Where 0 ≤ x ≤ 320 represents the column, and 0 ≤ y ≤ 240 represents the height. So for example, changing the pixel at x = 60 and y = 100 would mean going to this address:

PIXEL\_BUFF | (100 << 10) | (60 << 1)  
= 0xC8000000 | 102400 | 120  
= 0xC8000000 | 0x19000 | 0x78  
= 0xC8019078

For writing characters, it’s the same layout, except that it’s a character byte I write onto CHAR\_BUFF | (y << 7) | x. And x is up to 80 and y is up to 60.

Clearing the display just means setting every pixel to 0 (black). It can be done with two nested loops on x and y. I could have just called VGA\_draw\_point\_ASM to write the zeros, but unfortunately, I wrote the code for VGA\_clear\_pixelbuff\_ASM first. It would also be less code just setting every address to 0 instead of just the ones that map to a pixel. It would also require only one loop. Clearing the character buffers has the same layout.

By the way, if you look at the character buffer address (Figure 1), there are words that are filled with a’s instead of 0’s. These fields are unused addresses. They exist, because each buffer row is assigned 1 << 7 = 128 addresses, but there are only 80 columns. So there are 128 – 80 = 48 unused bytes each row.

Figure 1: Unused addresses in the character buffer

Challenges

Thanks to the starting code, there was a lot less code to write for this lab than the other ones. The only trouble I faced was that I got misalignment errors, which I later figured out was because I needed to use STRH and STRB, which I never needed to use before.

Shortcomings/Improvements

* Error checking for write character/pixel; what if the coordinates I put in are out of bounds?
* Increase portability: create constants for buffer width and height, then if I encounter a VGA display or character buffer that is of a different dimension, I can easily tweak the constants for it.
* Well obviously do more with it! There’s more to it than just hello world (oh wait that’s part 2 and 3).

Testing

Testing was as easy as running the code, seeing that the noise-filled screen turned black, before a gradient background appeared with the text “Hello World”. I compared it with the image in the lab handout, and it was the same.

**Part 2: Reading keyboard input**

Approach

This part is about reading keyboard input from the PS2\_DATA address. The two fields that matter are RVALID and DATA. The RVALID bit tells me if there’s a new value on the keyboard, and the DATA byte tells me what that value is. For the read\_PS2\_data\_ASM function, I am supposed to return whether there is valid data or not, and if there is, I should write the valid data to the pointer argument (from what I can tell at the starter code, that pointer is the stack pointer plus 7). Anyways, the procedure for my function is very simple:

1. Get the PS/2 Data address.
2. Get RVALID by using bit mask to get bit 15.
3. If RVALID is 0, return 0.
4. Otherwise, get the DATA field by “bitmasking” for bits 7..0.
5. Write into R0 those bits in step 4 and return 1.

It looks like the starter code clears the display and character buffer before polling for pressed/released keys to be written on the VGA in hexadecimal byte representation. It does the polling in “.input\_loop\_L9”. In L8, it checks to see that the buffer hasn’t overflowed by making sure that the memory to be written in is within rows 1-60. If it does overflow, it goes to L12 to exit the loop.

Challenges

This was the easiest part, since all I had to do was write one subroutine, with 13 instructions. At first, the VGA display wouldn’t stop printing 00, but after fixing a beginner’s bug, it worked like magic (Figure 2).

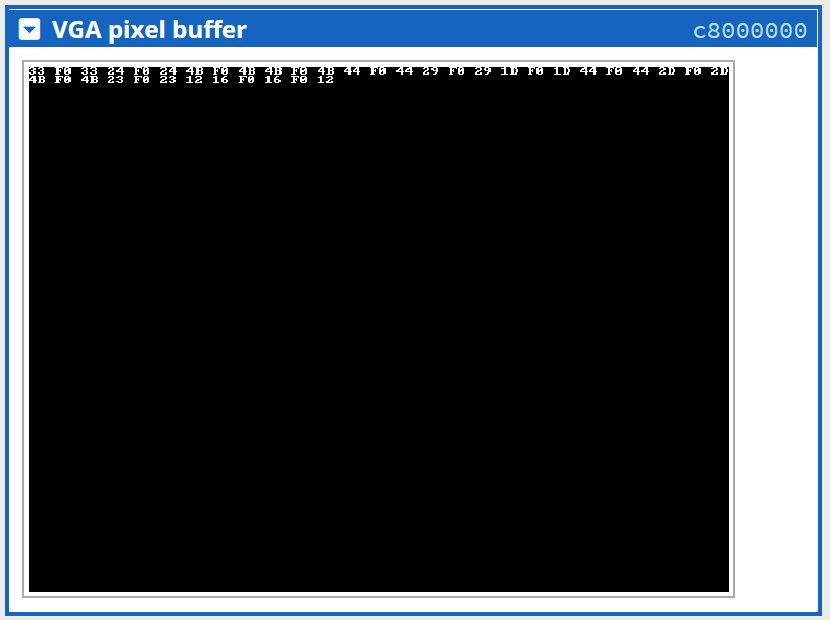


Figure 2: Typing Hello World! outputs these scan codes.

Shortcomings/Improvements

* as an alternate part of the lab, it would be cool to write onto the VGA display the numbers/letters themselves instead of make and break codes
* give the break codes a different color

Testing

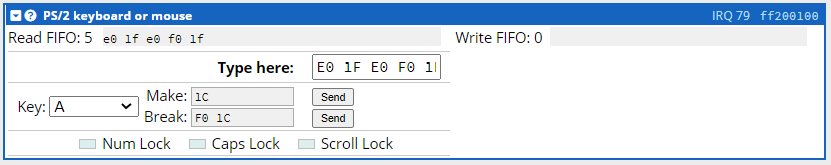
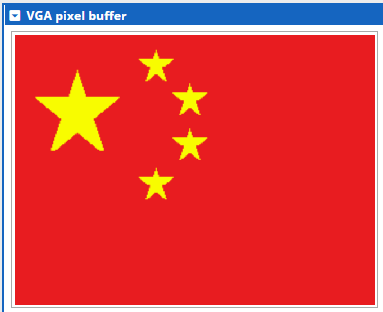


Figure 3: Emulator PS/2 keyboard device. Here I typed the Windows key just to see what the codes would be.

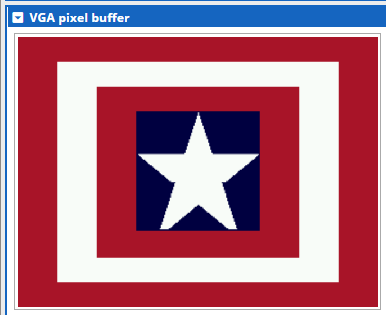
Testing was straightforward, just type some characters into the 0xFF200100 keyboard device (Figure 3) and see the codes get outputted. I tried pressing and releasing a key quickly, and also holding down a key before letting go to see repeated codes. Then I checked that the make codes match the ones in the lab handout. I checked that every time I released a key, a break code was generated. Finally, I tried to overflow the character buffer to see that it branches to end.

**Part 3: Vexillology**

Approach

With the functions that I wrote for part 1 and 2, I had to draw a real life and imaginary flag. Functions for drawing rectangles and stars were given to us. With them, I chose to draw the flag of China (PRC) (Figure 4), since the flags of Hong Kong and Taiwan are more complicated. I also drew a flag based off of Captain America’s shield. By observing the Texan flag code, I saw that they stored the draw\_rectangle’s 4th parameter in the stack. They also stored the color argument in memory, since it is too large to be an immediate value. With that in mind, for the Chinese flag, I simply made function calls to the red rectangle background and 5 stars. For the shield flag, I made successive calls to smaller rectangles, then finally to a star at the center (Figure 5).

Figure 4: The PRC flag in VGA pixel buffer

How did I get the colors right? What I did was get the proper 24-bit RGB colors online, then I shifted the red and blue bits to the right by 2 and for the green bits by 3 to get 16-bit color in 5:6:5 form.

As for what the starter code is doing, it looks like input\_loop first clears VGA, next draws the Texan flag, then it polls for the letters A or D in “.flags\_L39”. If the right key is pressed, it will clear the display then draw the next or previous flag. Then it keeps polling.

Figure 5: Captain America's shield in the form of a flag

Challenges

This part was very fun, the hard part was deciding what flags to do! I spent a lot of time getting the Chinese and shield flag to look authentic, there was lots of trial and error repositioning and resizing stars and rectangles. I was a little confused as to why I needed to subtract the stack pointer by 8 instead of 4; turns out that is because of the way draw\_rectangle function was coded, which is a bit inefficient.

Shortcomings/Improvements

* the Chinese flag’s little stars should be rotated for added authenticity
* the Captain America shield flag would look better as a circle
* create a helper function that takes in a 24-bit rgb color in three inputs and outputs the corresponding 5:6:5 color

Testing

Again very straightforward, press A or D to see the different flags pop up. When drawing the flags, it was just a cycle of writing code, compiling code, and seeing if the VGA flag looks like the actual flag.