Part 1: Drawing things with VGA

The goal of this first part of the lab was to implement drivers to control the VGA screen of the emulator. We used the provided starting code in the "vga.s" file and implemented the following described functions there. All functions follow the subroutine calling convention by using R0-R3 for arguments, pushing the used registers at the beginning of the function and popping them before using BX LR to return to the calling code. The first function we wrote was "VGA_draw_point_ASM" which is used to draw a point on the screen at the x-y coordinates specified in the first two arguments, the color of the point is specified in the third argument. We started the function by checking that the first two arguments given are valid: the screen is 320 pixels wide with a height of 240 pixels so if x is greater or equal to 320 or if y is greater than or equal to 240, we exit the function without displaying anything on the screen. After this check, we load the base address of the pixel buffer in R4 and access the pixel at the right x-y coordinate using the given formula: $0xC8000000 \mid (y \ll 10) \mid (x \ll 1)$. We save this resulting address in R4 and store the content of R2 (the color) in it. Figure 1 shows how we implemented this function.

The second driver we wrote was "VGA_clear_pixelbuff_ASM" which clears (set to black / '0') the entire pixel buffer. We used nested loops and the previously written function to implement this. Registers R0 and R1 keep track of the pixel coordinates and since we call "VGA_draw_point_ASM" and color black is '0', we put '0' in R2 before looping. The first loop loops over the x coordinate of the pixel buffer while the inner loop loops over the y-coordinate. At the beginning of the second loop, we call our "VGA_draw_point_ASM" subroutine. Then we compare R1 to 240 (which is above the highest y-coordinate possible for the VGA screen). If R1 is strictly less than this limit, we start the inner loop again after incrementing its value. Once R1 reached this limit, we get out of the inner loop to increment R0. If R0 is strictly less than 320 (so 319 is the maximum x-coordinate possible for the VGA screen), we loop back to the beginning of the outer loop. Otherwise, it means that we arrived to (240, 320) which is not a valid coordinate, so we exit the function. Notice that this function clears the screen pixel "column by column". Figure 2 shows the nested loop implementation.

```
14 VGA_draw_point_ASM:
15     PUSH {R4, R5, R6,
16     // Check if the cc
17     CMP R0, #320
18     BGE EXIT1
19     CMP R1, #240
20     BGE EXIT1
21     LDR R4, =PIXBUF
22     LSL R5, R0, #1
23     LSL R6, R1, #10
24     ORR R4, R4, R5
25     ORR R4, R4, R6
26     STRH R2, [R4]
```

Figure 1: Assembly code of the "VGA_draw_point_ASM" subroutine.

```
45 Loop1:
46     Loop2:
47     BL VGA_draw_point_ASM
48     CMP R1, #240
49     ADDLT R1, #1
50     BLT Loop2
51     MOV R1, #0
52     CMP R0, #320
53     ADDLT R0, #1
54     BLT Loop1
55     POP {LR}
56     BX LR
```

Figure 2: Assembly code for the loop clearing the VGA screen.

The third subroutine was "VGA_write_char_ASM" which writes the ASCII code passed in the third argument to the screen at the x-y coordinates given in the first two arguments. To do so, we used the character buffer which is continuously read by the VGA controller. We started the function by checking that the first two arguments are valid: the buffer is 80 characters wide with a height of 60 characters so if x is greater or equal to 80 or if y is greater than or equal to 60, we exit the function. We then implemented the function by loading the base address of the character buffer in R4 and accessing the character at the right x-y coordinate using the given formula: $0xC9000000 \mid (y \ll 7) \mid x$. We saved this resulting address in R4 and stored the content of R2 (the ASCII code of the character to display) in it. Figure 3 shows how we implemented this function.

```
65 VGA_write_char_ASM:
66     PUSH {R4, R5, LR}
67     // Check if the cc
68     CMP R0, #80
69     BGE EXIT2
70     CMP R1, #60
71     BGE EXIT2
72     LDR R4, =CHARBUFF
73     LSL R5, R1, #7
74     ORR R4, R4, R5
75     ORR R4, R4, R0
76     STRB R2, [R4]
77 EXIT2:
78     POP {R4, R5, LR}
79     BX LR
```

Figure 3: Assembly code of the "VGA_write_char_ASM" subroutine.

Finally, the fourth function we wrote was "VGA_clear_charbuff_ASM" which was very similar to "VGA_clear_pixelbuff_ASM". It sets to '0' all memeory locations in the character buffer. We also used nested loops to loop over all possible x-y coordinates in the buffer and used "VGA_write_char_ASM" to write a '0' in all these locations. Instead of comparing R0 and R1 to 320 and 240, we compared them to 80 and 60 respectively. This function also clears the character buffer "column by column". Figure 4 shows how we implemented this function.

```
VGA_clear_charbuff_ASM:
    PUSH {LR}
    // R0 and R1 keep track of
    MOV R0, #0
    MOV R1, #0
    MOV R2, #0
Loop3:
    Loop4:
        BL VGA_write_char_ASM
        CMP R1, #60
        ADDLE R1, #1
        BLE Loop4

    MOV R1, #0
    CMP R0, #80
    ADDLE R0, #1
    BLE Loop3

    POP {LR}
    BX LR
```

Figure 4: Assembly code of the "VGA_clear_charbuff_ASM" subroutine.

When implementing this first part, I only experience one minor issue which was the logical shift left that I implemented using a logical shift right. None of my functions worked until I fixed this issue. The correct output was then produced after compiling and running my program.

Part 2: Reading keyboard input

In this second part of the lab, we used the provided starting code in the file "ps2.s" which implements a program that read keystrokes from the keyboard and writes the PS/2 codes to the VGA screen. We therefore copied our previously written VGA drivers and then implemented the "read_PS2_data_ASM" function which takes as argument the memory address at which the data read from the PS/2 keyboard will be stored. This function also returns '1' if the data read is valid, otherwise it returns '0'. The subroutine also respects the subroutine calling convention by using R0 as an argument, pushing the used registers at the beginning of the function, using R0 to return the value and popping the used registers at the end before branching back to the caller.

The function starts by loading the PS/2 data register address in R4 (0xff200100) and loading the content of this register in R6 using another load instruction. We know that this register has a bit called "RVALID" that states whether the content of the register represents a new value from the keyboard. The subroutine checks this bit by using the given formula: $RVALID = \left((*\ (volatile\ int\ *)0xff200100\right) \gg 15)\ \&\ 0x1$. We therefore shifted the content of the data register stored in R6 by 15 to the right and stored the result in R5. We then used the "AND" instruction to extract the lowest bit of R5. We then compared this result also stored in R5 to '1' by using the "CMP" instruction. If "RVALID" was '0', the function moves '0' in R0 and exists the function because the data was not valid. Otherwise, we get the value stored in the PS/2 data register by ANDing the content of R6 with '0xFF' because the data register stores its data in its 8 lower bits (see DE1-SoC manual). We then store this retrieved value in the address stored in R0 (the argument). Since the "RVALID" bit was set, the data was valid so we then move '1' in R0 and exit the function. Figure 5 shows how we implemented this function.

```
read_PS2_data_ASM:
    PUSH {R4, R5, R6, LR}
    LDR R4, =PS2        /
    LDR R6, [R4]

    LSR R5, R6, #15     /
    AND R5, R5, #0x1    /
    CMP R5, #1          /

    MOVNE R0, #0        /
    BNE Exit4

    AND R6, R6, #0xFF   /
    STRB R6, [R0]
    MOV R0, #1

Exit4:
    POP {R4, R5, R6, LR}
    BX LR
```

Figure 5: Assembly code of the "read_PS2_data_ASM" subroutine.

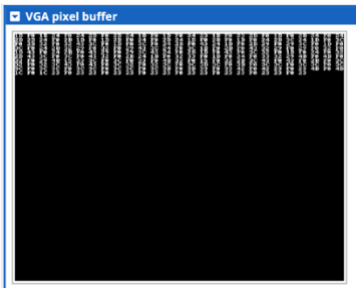

Figure 6: Output generated by running the ps2.s file which uses the "read_PS2_data_ASM" method we wrote

When working on this second part of the lab, I did not encounter any specific challenges and my code worked directly after I implemented it. I was just confused about the $(*\ (volatile\ int\ *))$ part of the formula to get the "RVALID" bit but reading the DE1-SoC manual helped me a lot. After compiling and running the program, I tested my code by typing into the PS/2 keyboard device which showed on the VGA screen the hexadecimal ASCII codes corresponding to the letters one after the other as shown on Figure 6.

Part 3: Putting everything together: Vexillology

The goal of this third part of the lab was to create an application that paints a gallery of flags on the VGA screen of the emulator. We used the provided code in "flag.s" that implemented an input loop that reads keystrokes from the keyboard using the PS/2 driver we wrote in part 2. If 'A' is pressed the VGA screen will display the previous flag, and if 'D' is pressed it will show the next one. The code also included a function that draws the flag of Texas and subroutines to draw rectangles and stars on the screen. Our objective was to write two other functions: "draw_real_life_flag" that draws a real flag and "draw_imaginary_flag" that draws a non-existing flag.

For the real flag, I decided to implement the European flag because it only required stars and a rectangle for the background which corresponded to the given methods “draw_rectangle” and “draw_star”. To draw a rectangle, we can simply call the “draw_rectangle” subroutine which uses the following arguments: R0 and R1 to specify the x and y coordinates of the top left corner of the rectangle, R2 and R3 for the width and height of the rectangle respectively and a fifth argument passed through the stack at address [sp] that denotes the color of the rectangle. The “draw_star” subroutine uses 4 arguments: R0 and R1 to store the x and y coordinate of the center of the star, R2 for the radius of the star and R3 for its color.

The function therefore starts by drawing a blue rectangle for the background. Since we want to cover the entire VGA screen, the coordinates of the top left corner of this rectangle are (0,0) so we move ‘0’ in R0 and R1 and the width and height are the same as the screen: 320 and 240, so we move ‘320’ in R2 and ‘240’ in R3. We want the background to be blue, so we must pass the corresponding 16-bit color value of blue to the “draw_rectangle” function through the stack. To do so, we used the blue color already defined in the program for the texas flag which is stored as a word in memory. We therefore subtract ‘8’ to the stack pointer to store the word value of the blue color. The function then uses the “BL” instruction to call the “draw_rectangle” subroutine as shown on Figure 7.

```
138 draw_real_life_flag:
139     push    {r4, lr}
140     // Draw a rectangle for
141     sub     sp, sp, #8
142     ldr     r3, .flags_L32
143     str     r3, [sp]
144     mov     r3, #240
145     mov     r2, #320
146     mov     r1, #0
147     mov     r0, r1
148     bl      draw_rectangle
```

Figure 7: Beginning of the “draw_real_life_flag” method that draws the blue background

Now that the background of the flag is set, we need to draw the 12 stars in circle in the middle of the flag. We used a separate photo editing software, where we imported a picture of the European Union flag on a canvas of 320x240 pixels. We then used lines to locate the coordinates of the center of each star as shown on Figure 8. Afterwards, we called the “draw_star” subroutine 12 times to draw these 12 stars, with R0 and R1 storing the corresponding coordinates and a constant radius of ‘20’ (stored in R2). We initialized a global variable storing the 16-bit color value of yellow which we stored in R3 every time we called the helper function as shown on Figure 9.

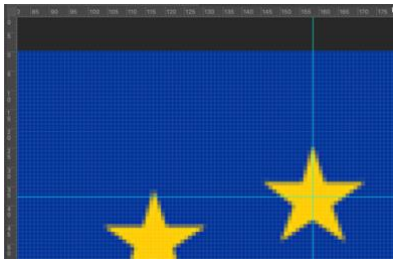

Figure 8: Method used to locate the center of each star on the European Union flag.

```
150     // Draw all stars of the european flag (12 total)
151     // Use the yellow variable storing the 16 bit value for yellow
152     // Coordinates were calculated using a photo editing software (see report)
153     // Use a radius of 20 for all stars
154     LDR r3, =yellow
155     mov r2, #20
156     mov r0, #158
157     mov r1, #38
158     bl draw_star
```

Figure 9: One of the call to the “draw_star” method in “draw_real_life_flag” function.

Before returning at the end of the function, since we subtracted ‘8’ to the sp register to pass an argument to “draw_rectangle” we must add ‘8’ to the sp register to pop the right registers at the end of the function. Figure 10 shows the resulted flag obtained after the Texas flag if we skip to the left by pressing ‘D’ in the PS/2 keyboard input.

Figure 10: Output of the “draw_real_life_flag” method.

When designing my imaginary flag, I wanted a white contour at the border of the flag, with red background and 4 superposed stars in the middle. I also wanted the stars to alternate color between white and black.
The "draw_imaginary_flag" subroutine starts with the white border by drawing a rectangle that takes the entire VGA screen as we did for the blue background in the European flag. The subroutine then draws a second red rectangle above the white one, letting a border of 20 pixels on each side. Therefore, when we called the "draw_rectangle" subroutine, we set the x-y coordinate of the top left corner of that rectangle to be (20,20) so we moved '20' in both R0 and R1. The width and height of the rectangle is therefore 320-20-20=280 and 240-20-20=200 pixels respectively, so we move '280' in R2 and '200' in R3. We used the red color of the Texas flag already declared in the program as the red for the background. Figure 11 shows how the function draws these two rectangles. An important note is that since we subtract '8' to the stack pointer value twice to draw both rectangles, we need to add '8' to sp before we pop our registers at the end of the function.

```
231 draw_imaginary_flag:
232     push    {r4, lr}
233     // Draw a rectangle for
234     sub     sp, sp, #8
235     ldr     r3, =white
236     str     r3, [sp]
237     mov     r3, #240
238     mov     r2, #320
239     mov     r1, #0
240     mov     r0, r1
241     bl      draw_rectangle
242
243     // Draw a rectangle for
244     sub     sp, sp, #8
245     ldr     r3, .flags_L32+8
246     str     r3, [sp]
247     mov     r3, #200
248     mov     r2, #280
249     mov     r1, #20
250     mov     r0, r1
251     bl      draw_rectangle
```

Figure 11: Beginning of the "draw_imaginary_flag" method that draws the two rectangles

Afterwards, the function draws the four stars in the middle, starting from the outside one (biggest one) to the smallest inner one. We therefore make four different calls to the "draw_star" subroutine, with the first two arguments always storing the coordinates of middle pixel in the VGA screen: R0 stores 320/2 = 160 and R1 stores 240/2=120. We use a radius of 100 for the biggest star (set in R2 before calling the function), 80 for the second one, 60 for the third one and 40 for the inner one. Finally, we alternate white and black colors between stars by loading in R3 the 16-bit color values of white and black that we initialized in global variables at the beginning of the program. Figure 12 shows how we called the "draw_star" helper function for the smallest black star.

```
271     mov r0, #160
272     mov r1, #120
273     mov R2, #40
274     LDR r3, =black
275     bl draw_star
276     add sp, sp, #8
277     add sp, sp, #8
278     pop     {r4, lr}
279     BX LR
```

Figure 12: Call to the "draw_star" helper method in the "draw_imaginary_flag" to draw a small black star in the middle of the flag.

When writing this third part of the lab, the main challenge was to determine the coordinates of the stars in the European union flag I decided to show on the VGA screen. Eventually, using this photo editing software helped me a lot, but I could have written a function in assembly to determine these coordinates. After compiling and running the code, the flag presented on Figure 13 is shown if we press 'A' from the Texas flag (or 'D' twice).

Figure 13: Output of the "draw_imaginary_flag" method.

A possible improvement of this code could be to use PUSH or POP instructions instead of manually subtracting and adding '8' to the stack pointer when we call the "draw_rectangle" subroutine.