ECSE 324 Computer Organization
Lab 3
Kua Chen 260856888

Introduction

In this lab, we explored the high-level I/O capabilities of the DE1-SoC simulator, including the VGA controller and PS/2 port. We first tried to interact with VGA by drawing things. After that we moved to understand keyboard input and PS/2 protocol. Finally, we combined two I/O capabilities to create an application which could dynamically update the VGA by pressing certain keys.

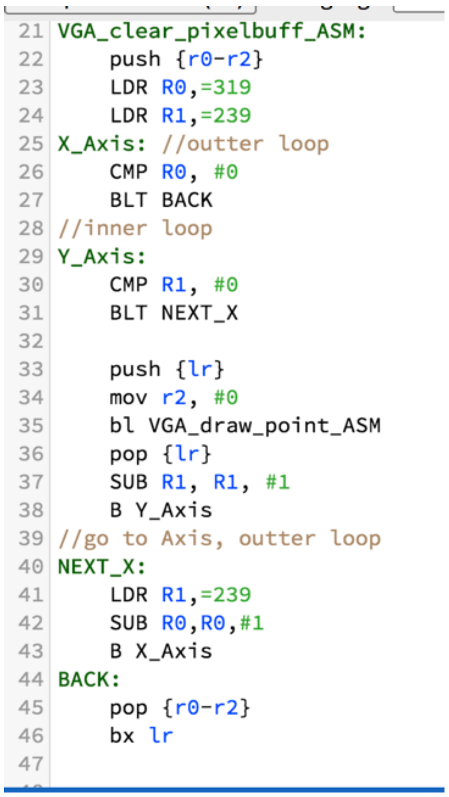
Part 1: vga.s

In this part, I was required to write a VGA driver to control pixel and character buffers.

VGA_draw_point_ASM: this function changes a single pixel on the screen to a designed color. Three input parameters are needed. Horizontal coordinate (R0) and vertical coordinate (R1) to locate the position of the pixel. R2 represents the destinated color code. The pixel buffer is 320 pixels wide and 240 pixels high. Individual pixel colors can be updated by STRH R2, [Address]. Address = 0xC8000000 + (y << 10) + (x << 1), where x is R0 and y is R1.

I encountered one small problem here about “memory address misaligned”. This was simply I used STR R2, [Address] at first. However, R2 is a 16-bit integer (half-word) and I should use STRH instead of STR.

VGA_clear_pixelbuff_ASM: this method clears pixel buffer by setting color code 0 to every pixel. A single pixel has horizontal coordinate and vertical coordinate. Therefore, if we want to clear all pixels, two for-loop statements are needed. X_Axis is the outer loop and Y_Axis is the inner loop. Inside the Y_Axis, subroutine **VGA_draw_point_ASM** was called to clear a single pixel.



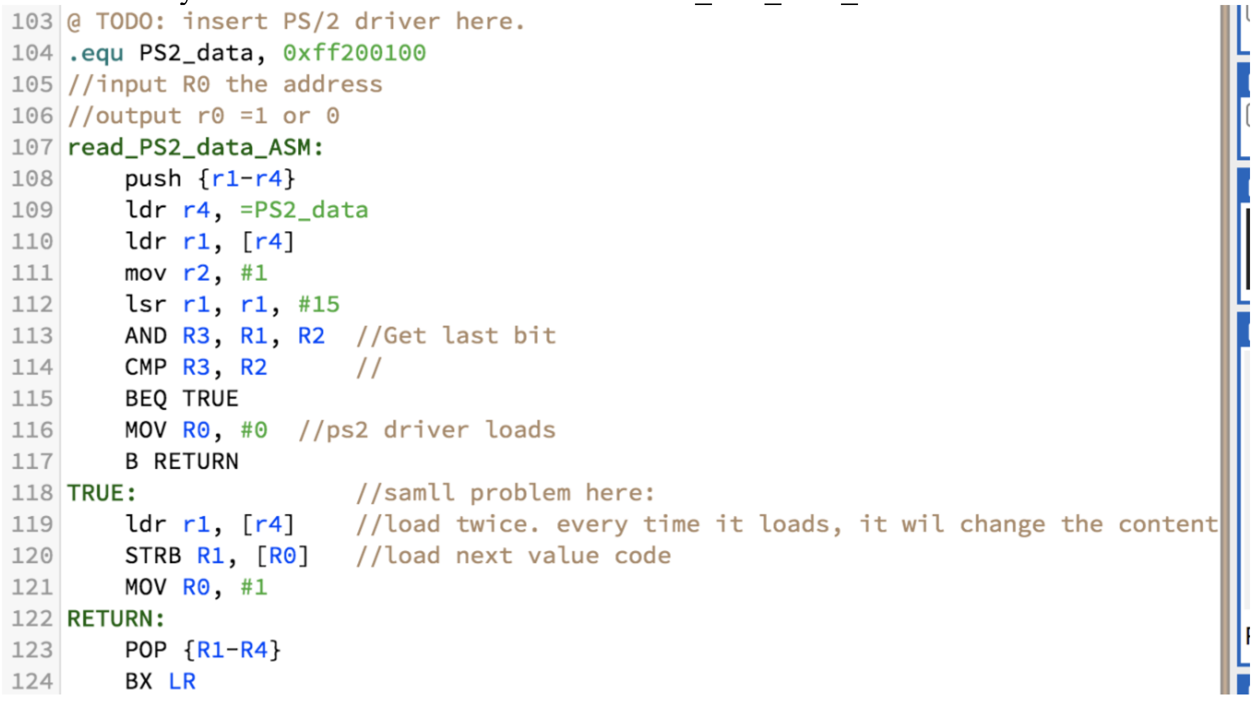
```
VGA_clear_pixelbuff_ASM:
    push {r0-r2}
    LDR R0,=319
    LDR R1,=239
X_Axis: //outter loop
    CMP R0, #0
    BLT BACK
//inner loop
Y_Axis:
    CMP R1, #0
    BLT NEXT_X

    push {lr}
    mov r2, #0
    bl VGA_draw_point_ASM
    pop {lr}
    SUB R1, R1, #1
    B Y_Axis
//go to Axis, outter loop
NEXT_X:
    LDR R1,=239
    SUB R0,R0,#1
    B X_Axis
BACK:
    pop {r0-r2}
    bx lr

```

One mistake I made was that when I reach the end of Y_Axis loop, I forgot resetting the Y value (R1) back to 239 again. The consequence was that this method could only clear one column of pixels.

VGA_write_char_ASM: this method writes a character to the screen with (R0, R1) coordinates. The character is represented by the ASCII code through R2. The base address for character buffer is 0xC9000000. To write a character on the screen, we should use STRB R2, [Address]. The buffer has a width of 80 characters and a height of 60 characters. Address = 0xC9000000 + R1 << 7 + R0. This method is very similar to **VGA_draw_point_ASM** but we should use STRB because ASCII code is 8-bit (a byte). Otherwise, we would have "memory address misaligned" again.

VGA_clear_charbuff_ASM: this method clears the character buffer by writing 0 to all the corresponding memory addresses. Similar to **VGA_clear_pixelbuff_ASM,** a screen is two-dimensional, and we need two for-loops to write 0 into all memory addresses for characters. Within the inner loop, subroutine **VGA_write_char_ASM** is called to perform "writing 0".

Part2: ps2.s
In this part, we would learn about how to interact with keyboard input (keystroke events) by PS/2 bus. Only one function needs to be created: **read_PS2_data_ASM.**



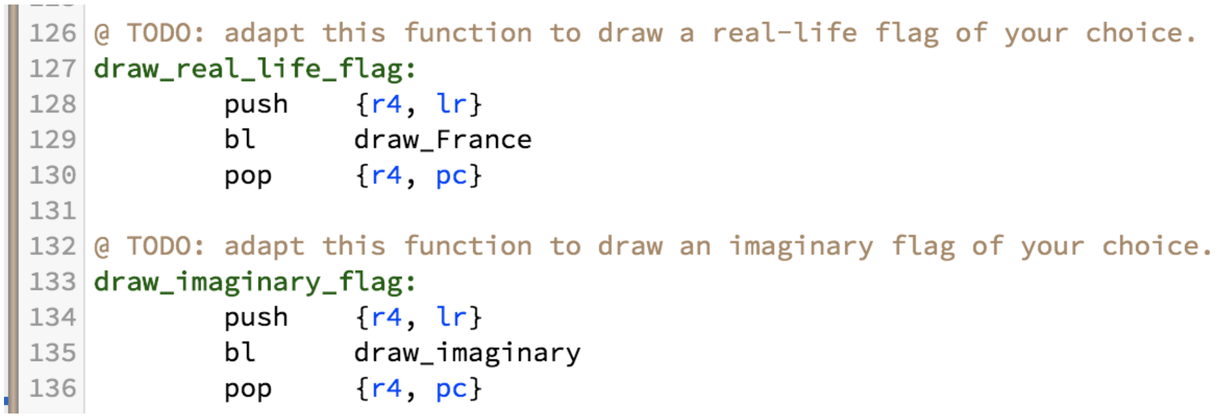
```
@ TODO: insert PS/2 driver here.
.equ PS2_data, 0xff200100
//input R0 the address
//output r0 =1 or 0
read_PS2_data_ASM:
    push {r1-r4}
    ldr r4, =PS2_data
    ldr r1, [r4]
    mov r2, #1
    lsr r1, r1, #15
    AND R3, R1, R2  //Get last bit
    CMP R3, R2      //
    BEQ TRUE
    MOV R0, #0  //ps2 driver loads
    B RETURN
TRUE:               //samll problem here:
    ldr r1, [r4]    //load twice. every time it loads, it wil change the content
    STRB R1, [R0]   //load next value code
    MOV R0, #1
RETURN:
    POP {R1-R4}
    BX LR
```

PS/2 data register is at address 0xFF200100. Its 15-th bit is the VALID bit that states whether the current contents of the register represent a new value from the keyboard. Therefore, after loading the content of [0xFF200100] into R1, we need to check whether the 15-th bit is 1 or 0. To achieve this goal, we need shift right R1 so that the 15-th bit becomes 0-th bit. Then we use AND and CMP to test whether this 0-th bit is 1 or 0. If it is 0, this means there are no new contents. The subroutine can return R0 with value of 0. If it is 1, this means there are new contents. Meanwhile, the subroutine should store the new contents into [R0] and return R0 with value of 1.

This code works fine for part2 task. However, it will bring about unexpected behaviour in part3. The problem is this: when I press A or D, the screen will quickly display for example, my imaginary flag but then refresh the screen to display the real flag, instead of directly displaying real flag. In other words, another flag will be shortly displayed before the correct flag comes up. This is an unexpected behavior, but my PS/2 driver passed the task in part2. So, I just took that PS/2 driver for granted. Then I came up with an analysis. One keystroke event would send Make Code and Break Code. For example, pressing A will send 1C and releasing A will send F0 1C. Since there are two 1C here. I thought this was the reason why the screen would behave in this way. Then I tried to only send 1C and now the screen would only be drawn once.

However, the instructor pointed out the culprit of this problem during my Demo. It turned out to be my PS/2 driver. The reason why the screen would be drawn twice is that I load the content of PS/2 data register twice in the driver and every time it is loaded, the content will be changed. The next value code (Break) would be loaded. Unfortunately, I was still somehow confused about this problem and I would try to improve my code following the instructor's correction.

Part3: flag.s
In this part, we combined VGA and PS/2 driver functions to create an application that could paint a gallery of flags. The first step was to copy paste the VGA and PS/2 driver functions into our program. Then we should visualize our real flag and imaginary flag by calling subroutines as shown below.



```
@ TODO: adapt this function to draw a real-life flag of your choice.
draw_real_life_flag:
        push    {r4, lr}
        bl      draw_France
        pop     {r4, pc}

@ TODO: adapt this function to draw an imaginary flag of your choice.
draw_imaginary_flag:
        push    {r4, lr}
        bl      draw_imaginary
        pop     {r4, pc}
```

With the subroutines **draw_France** and **draw_imaginary**, I constructed flags by calling **draw_rectangle** and **draw_star.** The difficult here was to understand the input parameters. Due to calling conventions, only R0, R1, R2, and R3 can be used as input parameters but **draw_rectangle** and **draw_star** need 5 input parameters. Lab instructor has provided one solution to this problem: the fifth argument is stored on the stack at address [SP]. Therefore, I should either push the fifth argument onto the top of the stack by PUSH instruction or directly overwrite [SP] by STR instruction. Due to subroutine calling convention, stack register must be the same as it was when entering the subroutine. Therefore, I chose to directly overwrite [SP] in order to avoid "clobbered stack register" error.