Jerry Xia, 260917329 | Lab 3 Report

Part 1

A brief description of each part completed (do not include the entire code in the body of the report).

- This program involved creating a VGA driver that was able to perform four functionalities - draw to a point, clear the pixel buffer data, write to the character buffer, and clear the character buffer. These are all predetermined places within the memory that the programmer can access.

The approach taken (e.g., using subroutines, stack, etc.).

- The stack was used to push values and preserve them as the program proceeds. The first method was simple; the only memory accessed was that of the display. By using r0 as x, r1 as y and r2 as the value, and storing it using 'strh', the value was able to be written into the correct data register. This also involved the bit manipulation shown in the lab's manual.
- `VGA_clear_pixelbuff_ASM:` An upper limit of 360 and 240, for x and y respectively, was set as well. This allowed for a loop, called i_loop_pixbuff and j_loop_pixbuff, to iterate through the indices of the display and set the pixel to 0, black. This method also used branch and link instructions as taught in class before.
- `VGA_clear_charbuff_ASM:` The above loop's logic was applied to this method as well, just with upper bounds for x and y of 80 and 60. As for the character writing method, strb was used as opposed to strh, as taught in class. This also used branch and link instructions, as it was very similar to the method described above.

The challenges faced, if any, and your solutions.

- At first, by using the instruction STR, the method VGA_clear_pixelbuff_ASM did not remove the right values. By considering the word approach, through understanding the bitwise use of the manual, it was understood to write to the first two bytes of memory only. This allowed me to get the correct answer.
- A similar approach was used with STR in VGA_clear_charbuff_ASM, by using STRB. Through understanding of the manual it was understood that clearing just a byte was necessary to get the desired result.

Possible improvement to the programs.

- There are a lot of stack operations that can be mitigated/reduced. The pushing of general use registers r0-r3 was potentially unnecessary and could be grouped together (e.g., push {r0-r4} instead of two separate operations) which could lead to less lines of code and overall better understandability.

Part 2

A brief description of the part completed.

- This program involved using the FIFO keyboard to add values to the screen. These weren't characters but rather the make/break codes that corresponded to the keys pressed by the user.
- These codes would be displayed from left to right, from the top corner to the bottom corner of the screen.

The approach taken (e.g., using subroutines, stack, etc.).

- This program used one of the same methods from the previous part, `VGA_clear_pixelbuff_ASM`, where the screen was set to black by passing a parameter, r2, with value #0x0 that cleared the 16 bits in that memory space, for all sections of the display.
- Data would be read from read_PS2_data_ASM as part of the PS/2 driver. It would load the address of the driver, shift by 15, and check if the bit resultant from that is 1. If so, then it would branch to data exists, where it would store the bite into a register, R2, and then exit.
- The other subroutine that was written was VGA_write_char_ASM, which uses this value from R2 to store into the character buffer. However, first it checks if the number is out of bounds or not. If so, it leaves. Otherwise, it shifts the coordinates as described in the manual to the right positions (R1 -> "y", R0 -> "x").

The challenges faced, if any, and your solutions.

- There were no challenges faced in this part of the lab. However, the make/break codes proved to be an issue when filtering out the right data to be processed in Part 3. Please see the challenges faced in Part 3 for further explanation.
- Otherwise, since this part of the lab only involved reading and writing data, and we already have seen that strb and strh are important in preserving this execution, this thus terminates the challenges faced in this component.

Possible improvement to the program.

- A potential failsafe for the program to terminate when the last space is occupied would be more beneficial as it would allow for the user to input to a capacity. However, this was not accounted for.
- Since the setup was rather straightforward, no further considerations should be made to improve the program - it does it's job simply and in a straightforward manner.

Part 3

A brief description of the part completed.

- This part involved creating a Tic-Tac-Toe game. It involved the use of the previous components as well as new logic that would allow for the game's flow:
    - When 0 is pressed, the game shall start; when 0 is pressed in-game, the game shall restart.
    - Numbers 1 - 9 represent positions on the grid.
    - The grid must be drawn out in a 207 x 207 pixel format.
    - The game must look overall presentable.

The approach taken (e.g., using subroutines, stack, etc.).

```
class Solution(object):
    def coordinates(self, coords):
        out = []
        for i in range(len(coords)):
            [x, y] = coords[i]
            out.append(int(x) | int(y) << 7 | int(3372220416))
        return out

s = Solution()
print(s.coordinates([
                    [23, 13], [39, 13], [55, 13],
                     [23, 29], [39, 29], [55, 29],
                     [23, 45], [39, 45], [55, 45],
                    [74, 4]])
)
```

- This program involved using the following script (shown above). It is able to calculate the coordinates of where the characters are placed in the character buffer. These were evaluated to what you see here:

```
.equ oneposition, 3372222103
.equ twoposition, 3372222119
.equ threeposition, 3372222135
.equ fourposition, 3372224151
.equ fiveposition, 3372224167
.equ sixposition, 3372224183
.equ sevenposition, 3372226199
.equ eightposition, 3372226215
.equ nineposition, 3372226231
.equ playertracker, 3372221002
.equ diagonal, 273
```

- After this, the keyboard was polled until a character 0 was inputted, in order to start the game.
- When the game is started, the entire screen is cleared to a more suitable colour (green) that is light on the eyes. The player is set to 1, which is the X player. The character buffer is also cleared.
- After this, all the interrupts are set up. This involved checking which interrupt occurred (at the end of SERVICE_IRQ) and branching to the appropriate position following this. The pushbutton template was replaced with the keyboard FIFO, which was hooked up to CPU1.
- There is a player tracker variable that is able to store information on the current player. Every time the player makes a move, this variable is updated from memory and used again to swap players for the next round.
- Now, there is also an ISR method for the PS2 driver. It checks digits 0-9 if they have been pressed, 0 allowing for the game to restart. Otherwise, it draws shapes in the form of ASCII characters onto the screen by:
    - Loading the correct position into a register.
    - Using that memory and, depending on what player is currently on, writing an X or an O based on that IF condition.
    - After this, it updates the one hot encoded string that contains the position of all the X's or O's.
    - It then checks that string if there is in fact an exact pattern. If there is, then a winner is found and then the program branches to the end, where we simply clear the screen and display "Player {x} wins!" where {x : 1, 2}.
    - If there is a draw, then the screen isn't cleared, but rather there is text displaying that there is a draw: "Draw!"
- There are several subroutines that help achieve this, notably clear_corner, write_player_1_turn, write_player_2_turn, draw_an_X, draw_an_O.
- This program also includes repertoire from the previous programs, such as VGA_clear_pixelbuff_ASM, and VGA_clear_charbuff_ASM.

The challenges faced, if any, and your solutions.

- The keyboard contains make/break codes, which had to be handled when considering what needed to be successfully read. When the user types into the keyboard, the first two codes need to be ignored (the make code, F0, thus leaving the break code that can be read afterwards). This was an initial problem because I would simply send the make codes without considering that the make and break codes would need to be accounted for. This was fixed by reading the

input twice and using the third entry as the code to determine what key was displayed on the screen.

Possible improvement to the programs.

- I was not able to implement a successful test to check if the tile was already filled. So, when a player would write an O to an X, then there would just be an overlap, the one-hot encoded string would update, and this would ultimately lead to a game that was corrupted. Having this check would make the player feel at ease - as they would not have to worry about mistyping a character in the wrong spot.