# Part 1

## Brief Description:

In this part of the lab, the task was creating a set of driver functions that can control the screen. There were four different drivers that had to be implemented, they were namely, **VGA_draw_point_ASM, VGA_clear_pixelbuff_ASM, VGA_write_char_ASM and VGA_clear_charbuff_ASM.**

## Approach Taken:

The approach taken closely resemble the instructions provided in the lab were very clear and easy to understand and implement. For the, VGA_draw_point_ASM, the pixel buffer was 320 pixels wide by 240 pixels high. Moreover, individual pixels can be accessed by doing 0xc8000000 | (y << 10) | (x << 1), where (x,y) correspond to the coordinates on the screen. After loading the base address, the instructions given in the lab were followed which can be seen in the code excerpt below. R0 is x while R1 is y and R2 corresponds to the color.

```
LDR R3,=0xC8000000 //loading base address
LSL R0,R0,#1
LSL R1,R1,#10
ORR R4,R0,R1
ORR R3,R3,R4
STRH R2,[R3]
```

The approach for VGA_write_char_ASM was very similar to VGA_draw_point_ASM, difference being that the character buffer was 80 pixels wide and 60 pixels high. Furthermore, accessing individual pixels was now performed through using 0xc9000000 | (y << 7) | x. The code for this is like the code shown above.

The approach for VGA_clear_pixelbuff_ASM was very similar as it solely consisted of continuously calling the VGA_draw_point_ASM driver explain above through all the available (x,y) coordinates with the colour argument set to 0 (corresponding to the colour black).

The approach for VGA_clear_charbuff _ASM was very similar as it solely consisted of continuously calling the VGA_write_char_ASM driver explained above through all the available (x,y) coordinates with the character value set to 0.

## Challenges faced and solution:

There were issues faced with the STR instructions, because the STR instruction were being used without any suffixes. After carefully reading the instructions, the suffixes were added.

## Possible Improvements:

The current program is already optimized, and no possible improvements can be made.

# Part 2

## Brief Description:

In this section of the lab, the task was to write a single driver called read_PS2_data_ASM. This driver would read data from key stroke events. It would check the data from the PS/2 data register at address 0xFF200100. This register contains the RVALID bit, and if this bit was a 1, the current content of that register was from a new key stroke event.

## Approach Taken:

The approach taken very much resembles the instructions that were provided for the lab. The address for the PS/2 data register was loaded and then its content was loaded as well. Thereafter, the instructions that were provided were followed closely. The RVALID bit can be accessed by shifting the content of the data register by #15 bits to the right, thereby obtaining its least significant bit and then subsequently doing an AND operation with this bit and the number #1. If the result is a 1, meaning it is valid, the content would be stored at the address in R0 and then subsequently return a #1 in the R0 register. If a valid result was not obtained, a #0 would be returned in the R0 register. The approach can be seen in the code below.

```
LDR R1,=0xFF200100
LDR R1,[R1]
LSR R2,R1,#15
AND R2,#1
CMP R2,#1
STREQB R1,[R0]
MOVEQ R0,#1
MOVNE R0,#0
```

## Challenges faced and solution:

At first, the instructions were interpreted wrong, as the content stored in the register R0 were being loaded and not the content of the register at 0xFF200100. This resulted in the code not executing properly and undesirable results. However, after rereading the instructions thoroughly, the mistake was found, and the instructions specified were understood that the R0 register represents the address where the data that was read will be stored.

## Possible Improvements:

The code was written through following the instructions given in the lab. Therefore, the current program is already optimized.

# Part 3

## Brief Description:

In this section of the lab, using the drivers previously written in the first two parts, everything was to be put and a fully functional tic-tac-toe game had to be built. This part of the lab had to consist of writing both the logic of the game as well as the visible user-interface on the screen.

## Approach Taken:

This part of the lab was divided into two different subsections. The first part corresponded to the user-interface and the second was the logic of the game.

### Approach Taken for User Interface:

For the background of the screen, the **VGA_clear_pixelbuff_ASM** that was described in the first section was used to make the background a black colour. Furthermore, a file called line.s that consisted of a subroutine called draw_rectangle was provided. This subroutine had five arguments, which were the (x,y) coordinates, the width and the height and finally the colour of the rectangle. The draw_rectangle subroutine was used to draw 4 different rectangles in a subroutine called draw_grid_ASM. These 4 rectangles corresponded to the look of the user-interface for a 3x3 tic-tac-toe game. The grid is 207 pixels by 207 pixels. Since the whole screen is 320 pixels by 240 pixels, a scaled version of the screen on paper and calculated the coordinates and width, length of the rectangles. A white colour was chosen for the grid so that it fits with the black background.

Furthermore, a plus (+) shape for player 1 was used as well as a square shape for player 2. For the plus shape, it consisted of another two rectangles. One that would have a higher width and low length, and the other one vice-versa. The square shape was two rectangles. Firstly, draw_rectangle was used to draw a square with the fill color set to white. Thereafter, a slightly smaller rectangle with the fill colour set to black was drawn. This would resemble the look of a square as the draw_rectangle subroutine would draw a black square on top of the white making it look like a square as the white rectangles become the borders for the black rectangle. Moreover, the results of the game and whose turn it was on were to be shown at the top center of the screen. The **VGA_write_char_ASM** was used to write those sentences. The ascii codes for the several letters that were to be used in showing those sentences were researched. To implement these functions, subroutines called **WRITE_PLAYER_TURN**, **WRITE_PLAYER_WIN and WRITE_DRAW_RESULT** were made.

### Approach Taken for Logic:

To start the game, the user had to press the “0” key on the keyboard. To check if that key had been pressed, the read_PS2_data driver described in section 2 was used. After checking the RVALID bit, the break code for the “0” key was checked. If a “0” had been pressed, the game

will with player 1's turn. If not, the code will continuously keep polling for "0" pressed key event. When player 1's turn starts, the message corresponding to their turn would be written at the top center of the screen. Thereafter, player 1's placement will be checked by branching and linking to a subroutine called **CHECK_PLAYER_1_PLACEMENT**. The first check in that subroutine would be to see if the player has pressed 0 to restart the game. If not, several checks would be made to see which of the 1-9 keys had been pressed. Based on the key pressed, the code would branch to one of 9 different subroutines written to draw the plus shape in one of the 9 grids. In all those subroutines, the first check would be to see if a block had already been placed in that grid by checking the contents of an address in memory corresponding to the label, "check_block_placed". If a block was already present, player 1's placement would be checked again. If not, then the block would be drawn in the grid.

Afterwards, when player 1's turn had finished, the 8 patterns corresponding to a win would be compared with the contents of the grid. A matching pattern on the board would correspond to a player 1 win. If it was a win, the result would be written on the screen and the game will be stopped. A continuous check is made again until the user presses a '0' to start another game. If player 1 does not win, a draw result will be checked. If the result is not a draw either, player 2 will now get their turn.

The logic for player 2's turn is identical to player 1's turn, only with square shapes instead of a plus. After his block had been placed, the pattern for a player 2 win would be compared with the contents of the board. If they were equal, the result would be displayed on the screen and the game would end. If it was not a win for player 2, the same process of player 1 and player 2 placing their blocks will be performed until a win pattern is seen.

There were 8 different win patterns. To check for the win, a comparison between the content of the memory for "check_block_placed" would be made with the 8 different win patterns. The content of the "check_block_placed" were 18 bits, where every 2 bits corresponded to one of the 9 grids. If player 1 places their plus in one of the 9 grids, the corresponding two bits would be filled with the bits "01" and if it was player 2, they would be filled with "10" instead. Therefore, a calculation was made as to what the 18 bits would be for the 8 different patterns. If the AND operation between the pattern and "check_block_placed" corresponded to the pattern, that player would be regarded as the winner. As for draw, the contents of the "check_block_placed" were checked again, and if it is seen that every grid had already been occupied, the result was a draw. This is because a draw is only check for after player 1's turn as a draw is only possible after player 1's turn. This is seen in the code in the following page.

```
CHECK_DRAW:
PUSH {R1,LR}
LDR R0,=check_block_placed
LDR R0,[R0]
AND R1,R0,#0b11
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b1100
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b110000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b11000000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b1100000000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b110000000000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b11000000000000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b1100000000000000
CMP R1,#0
BEQ NOT_DRAWN
AND R1,R0,#0b110000000000000000
CMP R1,#0
BEQ NOT_DRAWN
BL WRITE_DRAW_RESULT
MOV R0,#1
POP {R1,LR}
BX LR
NOT_DRAWN:
MOV R0,#0
POP {R1,LR}
BX LR
```

## Challenges faced and solution:

The biggest challenged faced was trying to figure out how to check for a drawn game. The code to check for the case of any of the players winning had already been written. However, a realization was made that a draw can only take place after player 1's turn and that every grid had to be filled with blocks for a draw to occur. After making that realization, the **CHECK_DRAW** subroutine was written, which has been explained above.

## Possible Improvements:

Currently, the code consists of 18 different subroutines for drawing either a plus or a square in one of the nine grids. However, that is a very inefficient method to place the blocks. A possible improvement would be to place the different coordinates of the nine grids in an array at the beginning of the program. The coordinates would then be used in only two different subroutines for drawing a plus and drawing a square.