ECSE 324 Lab 3: Drawing with VGA, Handling the PS/2 Device and Creating a Labyrinth Game

Karim Elgammal

McGill ID: 260920556

Task 1: Drawing Things with the VGA Panel

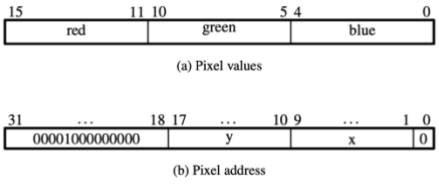
Problem Description:

Within this task we have been asked to implement a total of four drivers that interact with the VGA panel. We are asked to implement two drivers for the character display functionality, and two drivers for the pixel display functionality. More specifically we are asked to implement VGA_clear_pixelbuff_ASM and VGA_clear_charbuff_ASM, whose functionalities are to clear the pixel buffer for all pixels on the screen and to clear the character buffer for all characters on the screen respectively. These drivers are expected to take no arguments and further; they do not return anything.

Further, we are asked to implement VGA_draw_point_ASM and VGA_write_char_ASM, which take the pixel coordinates and character coordinates respectively as arguments. These drivers also take on a third argument in r2; which specifies the color of the pixel for VGA_draw_point_ASM and specifies the ASCII character for the VGA_write_char_ASM. These drivers use the arguments of the coordinates fed to them to locate said coordinate and then either change the color; or change the character that is presented at that location.

Design Decisions:

We have implemented the drivers for writing pixels (Fig. 1c) as follows; we access the buffer of each of these components using a load instruction; where each access to an address relates to an address of a pixel that is calculated using relative addressing to the base address of the respective buffer. At this address, we have access to a 16-bit value that represents the color of the pixel which is constructed of a red, green, and blue values. And so, we simply shift the x and y coordinates to the appropriate locations of the 32-bit address of the pixel buffer (Fig. 1b). Then, we simply store r2, which holds the RGB value of the color, into the location in memory pointed to by the address we had just calculated using the coordinates and base address of the pixel buffer.



(a) Pixel values

(b) Pixel address

```
VGA_draw_point_ASM:
        push { r0-r5, LR }
        LSL r0, #1
        LSL r1, #10
        ADD r0, r0, r1
        LDR r3, =pixel_buffer_address
        ADD r0, r0, r3
        STRH r2, [r0]
        POP { r0-r5 , LR}
        BX LR
```

c) VGA_draw_point_ASM Subroutine

Figure 1- Pixel Buffer Diagram and Driver for Writing Pixels

The driver for writing characters follows the same structure, differing by a couple of lines that check that we are not using invalid coordinates. We also shift the x and y coordinates fed into the driver an appropriate number of times to match the schematic of how the x and y coordinates are fed into the buffer (Fig. 2). And now, instead of storing the RGB value to that address, we simply store the ASCCII code that was fed into the driver by r2.

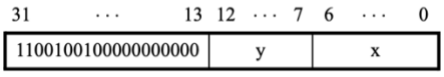


Figure 2 – Character Buffer Diagram

We have implemented the drivers for clearing the VGA screen using the drivers we implemented to write characters and pixels. We simply iterate through the entire coordinates of the screen by first choosing an x coordinate (starting from 0) and then iterating through all the coordinates of y and on each iteration calling VGA_draw_point_ASM or VGA_write_char_ASM with an argument in r2 of 0 for the driver to display a black point, or no characters at that location. We place limits on the coordinates by using the CMP instruction in order to realize the borders of the screen pixel layout and the screen character layout.

Optimization:

Throughout this task, optimizations for the character write and pixel write drivers are limited as they only effectively consist of a load instruction that loads the address of the respective buffer into a register and a store instruction that takes the value of our pixel color or ASCCI character code and stores it in the address we calculated using the x and y coordinates. And so, while there is not much optimization for the structure of these subroutines; there could be optimizations in the section of the code in which we calculate the address from the x and y coordinates. In order to shift, add and concatenate the x and y coordinates we utilize an extra register to store the intermediate values between calculations; we can instead limit this practice and calculate each of these intermediate values into the r0 and r1 registers directly, which will save access to registers and therefore allow for a more efficient process for the ALU on the processor.

As for the clearing drivers, the structure of our code leads to many loops being executed and we implemented two sets of these drivers; this can be optimized by sharing drivers that commit to the same functionalities and instead having scratch registers which convey which subroutine is calling the driver. This would have allowed for reducing complexity of code.

Task 2: Handling the PS/2 Port

Problem Description:

Within this task we are asked to implement a singular driver that takes as an input a memory address in r0. This driver must then poll the RVALID bit within the PS/2 data register; if the RVALID bit is 1, this signifies that the data register has data within its data bits 0-7 that is ready to be read. This driver must then take this data and store it within the address that was supplied by r0. Further, this driver should return by way of the r0 register a 0 or a 1, which corresponds to the state of the RVALID bit in the PS/2 data register.

Design Decisions:

Throughout designing this task, we were able to implement it using a simple structure (Fig. 3) that involves several load instructions that load the address of the variable we are using to place PS/2's data. Further, we use the STRB instruction to further optimize our program. We extract the data from the PS/2 data register by running an AND command with the value #0xFF, which allows us to examine the last 8 bits of information stored in the data register. We also use the same approach to extract the RVALID bit from the very same register. However, in this rendition of the structure we also shift the register value by 15 bits to the left to access bit number 15, which hold our RVALID bit. Finally, we move the RVALID bit we just extracted into r0, for the subroutine to return.

```
read_PS2_data_ASM:
    PUSH {r1 - r3, LR}
    LDR r1, =ps2_data_register
    LDR r2, [r1]
    MOV r3, r2
    AND r3, r3, #0xff
    LSR r2, #15
    AND r2, r2, #0x1
    TST r2, #0x1
    STRNEB r3, [r0]
    MOV r0, r2
    POP {r1 - r3, LR}
    BX LR
```

Figure 3: read_PS2_data_ASM

Optimization:

There are optimization factors that were implemented when designing this subroutine; firstly, we have tried to avoid the use of unnecessary registers for our calculations; however, this could have been improved further. More specifically, when we extract the RVALID bit, we can just return that bit, instead of testing it using a TST instruction with the value of #0x1. There are some redundancies within the code that allow for a more readable and comprehendible structure to follow. Further, this subroutine is essentially a driver that polls the PS/2 device on whether the data it holds is ready or not; we can instead implement an interrupt-based structure that can avoid having the processor busy while waiting for the data to be ready. This is discussed further in task 3; as it pertains to waiting for an input from a user.

Task 3: Creating the Labyrinth Game

Problem Description:

Task 3 asks us to use the drivers that we have implemented in task 1 and task 2; to create a working maze game program that allows the user to move in a 9 by 12 map of obstacles, where the objective of the game is to reach the exit. An example of a map can be examined in figure 4. The task first consists of using the VGA drivers to draw an empty 9 by 12 grid that fits the VGA screen. Then, by an input from the user (from 1 to 9) we are directed to a specific map in memory which is loaded into the empty grid we drew. The program then displays the exit, and further the player location (represented by an ampersand in our running program). Once the game has been set up completely and the screen depicts that of figure 4, the program must allow the user to move the ampersand (using the up, down, left, and right keys on the keyboard), to the exit, which is blocked by a series of blocks that the ampersand is not allowed to move into. Further, once the user reaches the exit, the VGA screen should display the winning message/status and wait for 10 seconds before returning to the empty grid and asking the user for another map number input.

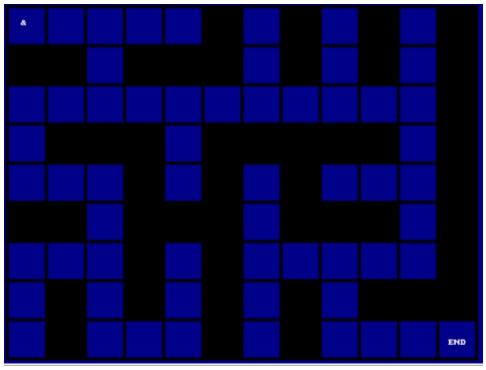


Figure 4 – Example Gameplay Map

To implement this, we must design a few drivers that implement the following functionalities: fill the empty grid with squares, move the ampersand to valid locations within the map, wait for an input from the user and finally, a driver that allows for the winning status to be reached. There are also sub-drivers that are used to reach these functionalities which utilize the previously implemented drivers.

Design Decisions:

To realize the functionalities of our program; we have chosen to implement a polling-based structure to how we deal with the PS/2 device. That is, upon entering a 'cycle' of the program (the rendition of 1 of the mazes to completion), we continuously poll the PS/2 data register's RVALID bit to let us know if a key has been registered. We then compare the data received on two rounds of 'detecting' RVALID bits; in order to extract the break code of the key that has been registered. We chose to base our indicator of a key on the break code, as it is composed of unique numbers after the initial identifier. And so, we discard any irrelevant key identifiers that do not pertain to the keys we are expecting on the 'first' read of the PS/2 data register. And then, we match the end of the break code with a number pertaining to the map number (Fig. 5). We implemented the same structure to poll the user for directional moves when the game is being played, but instead of matching the break code with numbers we matched them with relative identifiers that allow us to call the move_ASM driver that then completes the move if valid.

```
our_second_read2:
    CMP r3, #0x16
    ADDEQ r5, #0x1
    BEQ full_click_registered2

    CMP r3, #0x1e
    ADDEQ r5, #0x2
    BEQ full_click_registered2

    CMP r3, #0x26
    ADDEQ r5, #0x3
    BEQ full_click_registered2

    CMP r3, #0x25
    ADDEQ r5, #0x4
    BEQ full_click_registered2

    CMP r3, #0x2e
    ADDEQ r5, #0x5
    BEQ full_click_registered2
```

```
our_second_read:
    CMP r3, #0x75
    ADDEQ r5, #0x1
    BEQ full_click_registered

    CMP r3, #0x6b
    ADDEQ r5, #0x10
    BEQ full_click_registered

    CMP r3, #0x72
    ADDEQ r5, #0x100
    BEQ full_click_registered

    CMP r3, #0x74
    ADDEQ r5, #0x1000
    BEQ full_click_registered
```

Figure 5 – Comparing PS/2 Data for Map Input and Arrow Input

When coming to design the polling-based input from user for the directional moves, we considered going through all three bytes that the break code for the arrow keys consists of; this is to ensure that no other key is interpreted as a move. However, the PS/2 device available in the simulation at hand does not accept the 0xF3 command, which is the command that would have allowed us to set the typematic rate and delay; allowing us to control the timings of the make and break codes being sent. And so, to enhance the functionality and responsiveness of the program to our keys, we only wait for the last two bytes to be registered as a valid input to the PS/2 data register, and take a move based on that process reaching completion. After testing our program on both implementations, we have opted to use the latter as it allows us a more responsive program.

The program utilizes an array of coordinates that is accessed by our draw_ampersand_ASM driver to calculate the central locations of the ampersand at any given time in conjunction with the 9 by 12 grid. This design choice allows us to fix the ampersand locations based on the coordinates of the grid (which do not scale exactly to the character coordinates of the screen. And so, to bypass this, we have gone through all the locations that the ampersand could be displayed in and relayed these locations to our draw_ampersand_ASM using the array_of_coordinates. We use the same grid-to-character coordinates for the x and y points to allow a central location in the square. This array can be examined in figure 6.

```
array_of_coordinates: .word 3, 10, 16, 23, 30, 36, 43, 49, 56, 62, 69, 75
```

Figure 6 – Array Used to Calculate Grid to Character Coordinates

As per the drivers which fill the screen and draw the grid, we utilized the previously implemented drivers or even their sub-drivers. An example is shown in figure 7, where we use the sub-drivers of the VGA_clear_pixelbuff_ASM within the VGA_fill_ASM driver. This design choice allows for a reduced complexity in terms of the code structure and in terms of the runtime while compiling the program as it is one less subroutine to account for in the instructions mapping.

```
VGA_fill_ASM:
        PUSH {r0, r1, r2, r3, r4, r5, LR}
        MOV r0, #0
        MOV r1, #0
        LDR r2, =background_color
        MOVW r3, #319
        BL  choose_x_pixel_clearing
        POP {r0, r1, r2, r3, r4, r5, LR}
        BX LR
```

Figure 7 – VGA_fill_ASM Calling choose_x_pixel_clearing, a Driver for the VGA_clear_pixelbuff_ASM Subroutine

Our move_ASM driver is at the heart of the operation of our program, it allows us to move from one block in the grid to another, without letting the user land on a block that is filled with an obstacle. This subroutine takes as an input r0, r1, and r2, where r0 and r1 are our proposed new coordinates for the ampersand to move into, and r2, the index of the current map we are playing. The function first checks if this move would be legal in terms of maximum and minimum coordinates. Then using relative addressing checks if there is a 'block' in the location we want to move to. This is done by realizing that each map address can be calculated as map_address = base_address_of_input_map + 432i, where i is the desired map number we want to access. This is due to there being 432 bytes between the start of one map to the start of the next one. We follow a similar logic to extract the status of each coordinate on the grid, that is we know that the rows start every 48 bytes from the start of the previous row, and so using this scheme we find if there is a '1' recorded at that memory location, and then we take moves if there is not a 1 stored in that memory location. Finally, this subroutine returns a value in r3 that allows us to flag when a move has been made and when a move has failed due to the restrictions described above. A portion of the subroutine is supplied in figure 8 to showcase the calculation being performed.

Through these main drivers, and subroutines that guide the flow of the program from one driver to the other, we can implement the full running program until the user reaches the coordinate (11,8) which is our exit. Once it is reached, we used the A9 private timer in order to stall for 10 seconds before restarting the entire program to wait for another map input. The timer is configured using a start value of # 0x77359400 to account for the 200Mhz clock frequency; this allows us a stalling of 10 real-time seconds. We have chosen to divide all the functionalities into multiple subroutines to reduce the complexity of the code and allow for streamline debugging. Further, the accesses to memory to validate moves allows us to reduce the registers being used to perform ALU operations.

```
SUB r2, #1
LDR r3, =input_mazes
MOVW r4, #432
MUL r2, r4
ADD r3, r3, r2
@now our address of map is in r3 (our base)
PUSH {r0, r1}
LSL r0, #2
MOV r5, #48
MUL r1, r5
ADD r3, r3, r0
ADD r3, r3, r1
LDR r6, [r3]
CMP r6, #0x1
POPEQ {r0, r1}
MOV r3, #0
BEQ end_of_move
BNE make_a_move
```

Figure 8 – Portion of our move_ASM Driver

Optimizations:

Throughout this implementation, we can recognize certain instructions or techniques which can be modified to optimize the performance, time complexity or efficiency of the program in terms of the processor's tasks. The first optimization/improvement that is suggested, is implementing the program using an interrupt technique rather than the polling technique. Owing to the fact that when the processor is constantly polling the device, it is in a 'busy' state and further uses more registers than needed to perform the task at hand. An interrupt technique allows us to instead wait for the device to request an interrupt, which proves to be more efficient. This can be changed within the polling for the map number and further the polling for the moves of the ampersand.

Further optimizations include setting the typematic rate and delay of the PS/2 device in a way that allows us to register user clicks in a more efficient way. Perhaps increasing these times, which will allow us to register full clicks using one 'access cycle' rather than the two 'access cycles' we currently utilize. The program accesses memory in order to validate a move; an optimization to this could be instead making it so that the map is extracted into a number of registers, that could then be used to validate each move. There are also instances within our code, where we have multiple drivers that are very similar in functionality. For example, draw_grid_ASM utilizes two sets of drivers that essentially do the same processing steps, however one is by horizontal orientation and the other for a vertical orientation; these sets of drivers could have been combined into one to allow for a shorter code length and in turn shorter compile time.

Lastly, there are some instances within our code, where rather than using relative addressing syntaxes, we directly use counter registers to hold the counter values. This increases the number of registers which are being used and in turn, decreases the overall performance of our program.