ECSE 324: COMPUTER ORGANIZATION

LAB REPORT 3

BY

ZARIF ASHRAF

ID: 260782942

Throughout this lab I was supposed to implement several different drivers using assembly language ARMv7. The first part involved creating 4 different drivers which could respectively draw points on the VGA screen, clear points already on the screen, write characters on the VGA screen, and finally clear characters that were already on the screen. The second part was reading data input into the PS/2 keyboard. The final part was to design 2 flags and see it all come together using the VGA and the PS/2 driver. In order to develop these programs, I worked with the CPUlator ARMv7 System Simulator.

PART 1:

VGA_draw_point_ASM:

After initializing ".equ VPB_Base, 0xC8000000" I started with a callee-save convention since this is a subroutine, and likely to be called by other subroutines who might want to use the values of the registers they passed onto this subroutine later on. The new few lines of code that follow check that the values of the x-coordinate or y-coordinate that might be passed in fall within the range of pixels available on the VGA pixel buffer. In case they are not, they are branched back to where they were called from.

```
MOV R3, #300            // the block checks if the coordinates are within range
ADD R3, R3, #19
CMP R0, #0              // check x coordinate is greater than 0
BXLT LR
CMP R0, R3              // check x coordinate is less than 319
BXGT LR
CMP R1, #0              // check y coordinate is greater than 0
BXLT LR
CMP R1, #239            // check y coordinate is less than 239
BXGT LR
```

For the next step, register R3 is made to point to the VPB_Base, while the x-coordinate is shifted to the left by 1 and the y-coordinate by 10, since that's the bit they start from on the pixel address. Each of them are then, in turn, added to the base address, and once we have the exact location we want to draw a point at, we store the value of our input as a half-word inside this location, to be reflected on the VGA screen.

Finally, the registers are popped as part of the callee-save convention and branched back to where the call came from.

```
LDR R3, =VPB_Base
LSL R0, #1              // shift by 1 since x coordinate starts from the 1st bit
ADD R3, R3, R0          // add x coordinate to the base address
LSL R1, #10             // shift by 10 since y coordinate starts from the 10th bit
ADD R3, R3, R1          // add y coordinate to the base address
STRH R2, [R3]           // store the input value to the address
POP {R0-R3, LR}
BX LR
```

The initial challenge I faced while writing this was to try and compare r0 against 319, since that's the upper limit of the x coordinate, but is too large a value to hold for the simulator. It took me some piazza posts to figure out a way to solve this issue. One way to further improve this might be to incorporate the LSL instructions and the ADD instructions, however I feel they would still use the same number of registers.

VGA_clear_pixelbuff_ASM:

This part begins with the callee-save convention as well and goes onto initialize x and y counters for the pixel values, while also holding the value 319 inside the register R5. Both the counters start with an initial value of 0. R2 points to the base address, and r3 contains the value #0.

The driver is then run through 2 separate loops. The first loop checks if the value of value of the x counter is within range. In case it is not, the driver branches it back to where it was called from, after popping the register values as part of the callee-save convention.

```
POP {R0-R7,LR}
BX LR
```

If the x counter is within range, the value of the y counter is checked to see if it's within range. If y counter is out of range, it means the driver has iterated through all pixels in a single column, so it is time to move onto the next, in which case, the value of the x counter is incremented by 1.

However, if the value of the y value was within range, it was shifted to the left by 10, and the x value was shifted to the left by 1, then both were added to the destination address. Finally, a value of 0 was stored inside this exact location. Copies of x and y coordinates were used in this case, so that the count would remain unaltered during these calculations. A register R4 was also used to reflect the location, since the following values would also be provided in relation to the base address, and altering the register R2, which holds this base address would tamper the calculations for the next coordinates to be supplied.

```
MOV R4, R1          // take y counter
LSL R4, #10         // shift by 10 since y starts at the 10th bit
ADD R4, R4, R2      // add base address
MOV R7, R0          // make a copy of the x counter
LSL R7, #1          // shift by 1 since x starts at the 1st bit
ADD R4, R4, R7      // add the value of x counter

STRH R3, [R4]       // store 0s into the location
```

The y counter is incremented, and the loop continues as long as y is within range.

One of the problems I faced during writing this was the idea of using copy registers for x and y counters and using a different register other than the one which holds the value for the base address to reflect the outcome. It took me some amount of thought to finally get it right. Plus, I wasn't initially sure about how to store a value of 0 for the colors at the required location, but looking at some of the pre-provided code for lab 3 and going through piazza solved this for me.

A possible improvement here would be to use fewer possible registers.

VGA_write_char_ASM:

After initializing ".equ VCB_Base, 0xC9000000" I started with a callee-save convention since this is a subroutine, and likely to be called by other subroutines who might want to use the values of the registers they passed onto this subroutine later on. The new few lines of code that follow check that the values of the x-coordinate or y-coordinate that might be passed in fall within the range of available on the VGA character buffer. In case they are not, they are branched back to where they were called from.

```
CMP R0, #0              // check if x coordinate is greater than 0
BXLT LR
CMP R0, #79             // check if x coordinate is less than 79
BXGT LR
CMP R1, #0              // check if y coordinate is greater than 0
BXLT LR
CMP R1, #59             // check if y coordinate is less than 59
BXGT LR
```

For the next step, register R3 is made to point to the VCB_Base, while the y-coordinate is shifted to the left by 7, since that's the bit it starts from on the character address. The x and y coordinates are then, in turn, added to the base address, and once we have the exact location we want to write a character at, we store the value of our input as a byte inside this location, to be reflected on the VGA screen.

Finally, the registers are popped as part of the callee-save convention and branched back to where the call came from.

```
LDR R3 ,=VCB_Base
ADD R3, R3, R0          // add x coordinate to the base address
LSL R1, #7              // shift by 7 as y coordinate starts from 7th bit
ADD R3, R3, R1          // add y coordinate to the base address
STRB R2, [R3]           // store the input value to the address
POP {R0-R6,LR}
BX LR
```

Once I had done the VGA_draw_point_ASM, there weren't a lot of problems I faced with this one.

VGA_clear_charbuff_ASM:

This part begins with the callee-save convention as well and goes onto initialize x and y counters. R2 points to the base address, and r3 contains the value #0.

The driver is then run through 2 separate loops. The first loop checks if the value of value of the x counter is within range. In case it is not, the driver branches it back to where it was called from, after popping the register values as part of the callee-save convention.

If the x counter is within range, the value of the y counter is checked to see if it's within range. If y counter is out of range, it means the driver has iterated through all y values in a single column, so it is time to move onto the next, in which case, the value of the x counter is incremented by 1.

However, if the value of the y value was within range, it was shifted to the left by 7, and the x value remains unshifted, and then both were added to the destination address. Finally, a value of 0 was stored inside this exact location. Copies of x and y coordinates were used in this case, so that the count would remain unaltered during these calculations. A register R4 was also used to reflect the location, since the following values would also be provided in relation to the base address, and altering the register R2, which holds this base address would tamper the calculations for the next coordinates to be supplied.

```
MOV R4, R1              // take y counter
LSL R4, #7              // shift by 7 since y coordinate starts from the 7th bit
ADD R4, R4, R2          // add the base address
ADD R4, R4, R0          // add the x counter

STRB R3, [R4]           // store value into the location
```

The y counter is incremented, and the loop continues as long as y is within range.

Once I had done the VGA_clear_pixelbuff_ASM, there weren't a lot of problems I faced with this one.

read_PS2_data_ASM:

After the usual callee-save convention, R1 points to the base address of the PS2 keyboard and the value of this address is then loaded into R1. This is so we can perform calculations on the value of this address. At first, all bits except RVALID are cleared and the RVALID bit is checked. If it is equal to 0, it is branched to an end where R0 is set to be 0, and the registers are popped back before finally branching back to where the call initially came from.

```
MOVLT R0, #0            // if RVALID = 0, r0 =1
POP {R1-R2, LR}
BX LR
```

If RVALID is not equal to 0, all bits except the data bits are cleared, and these data bits are then stored inside R0, as requested by the lab outline. The value of R0 is also set to 1, before finally popping the registers and branching to where the call initially came from.

```
AND R1, #0xFF           // all bits except the data bits are cleared
STRB R1, [R0]           // store data inside r0
MOV R0, #1              // r0 = 1
POP {R1-R2, LR}
BX  LR
```

An initial problem I faced regarding this was to figure out how I could perform calculations on the memory address. I cannot think of a possible improvement that could be made in this case yet.

draw_real_life_flag:

The draw_texan_flag subroutine was used as a guidance for this subroutine. I wanted to draw the flag of Poland, so I needed two rectangles, red and white, divided in the middle.

```
ldr     r4, .flags_L32+4
str     r4, [sp]
mov     r3, #120    // height
mov     r2, #320    // width
mov     r1, #0      // y-coordinates
mov     r0, #0      // x-coordinates
bl      draw_rectangle
```

The above code was used to draw the white rectangle, by loading the color into r4, and then declaring the parameters x, y, width, and height accordingly. The same was repeated for the red rectangle, after changing the color to red, and readjusting the parameters.

It took me a little bit of time to figure out how to exactly adjust the parameter values, but everything other than that was quite straightforward.

draw_imaginary_flag:

This subroutine is quite self-explanatory as well. I added the following code to add the vertical orange rectangle with the white star in the middle.

```
ldr     r3, .flags_L420
str     r3, [sp]
mov     r3, #240
mov     r2, #106
mov     r1, #0
mov     r0, r1
bl      draw_rectangle
ldr     r4, .flags_L32+4
mov     r3, r4
mov     r2, #43
mov     r1, #120
mov     r0, #53
bl      draw_star
```

The next bit of code that follows draws the top white rectangle.

```
str     r4, [sp]
mov     r3, #120
mov     r2, #214
mov     r1, #0
mov     r0, #106
bl      draw_rectangle
```

The rest of the code draws the bottom orange rectangle before popping the registers.

There weren't a lot of difficulties I faced with this subroutine, once I had figured out how to do draw_real_life_flag.