ECSE 324 - Lab 4 Report - Group 46

Jessica Li (260665795)

Claire Liu (260654285)

Discussion on Approaches to Each Section:

VGA.s:

The goal of this section is to display/clear the character buffer and pixel buffer based on the pushbutton pressed. To achieve this goal, five subroutines were created: VGA_clear_charbuff_ASM, VGA_clear_pixelbuff_ASM, VGA_write_char_ASM, VGA_write_byte_ASM, and VGA_draw_point_ASM. The code that tests these subroutines is shown at the end of this section.

The logics for clear_charbuff and clear_pixelbuff are similar. We constructed a nested loop consists of two loops. The outer loop loops through all x-coordinates of the pixel/character buffer and the inner loop loops through all y-coordinates of each buffer. During each iteration, the buffer value at every x and y-coordinate is set to 0 in order to clear the buffer.

The logics for draw_point and write_char are similar. First we check whether the input x, y of the subroutine are valid by comparing the input arguments with the minimum and maximum of the buffer dimensions. For draw_point, we compared x with 0 and 319 and y with 0 and 239. For write_char, we compared x with 0 and 79 and y with 0 and 59. Then, the corresponding address are calculated by adding the input x and y arguments to the buffer base address. Finally, the input char argument is stored to the calculated appropriate address.

For write_byte, we started by comparing the input x and y arguments with the char buffer dimensions (0 and 78 for x, 0 and 58 for y since the method needs to print out two HEX numbers). For displaying each HEX number of the input char's ASCII, we calculated the appropriate buffer address is calculated by adding the input x and y to the base address, then used LSR to extract the individual HEX number, added this number to the address of the HEX_ASCII array, extracted the appropriate number in the array and stored that number to the buffer address.

```
// VGA
        if (read_slider_switches_ASM()> 0 && PB_data_is_pressed_ASM(PB0)){
            test_byte();
        }
        //if slide switch not on and PB0 is pressed, call test_char()
        if (read_slider_switches_ASM() == 0 && PB_data_is_pressed_ASM(PB0)){
            test_char();
        }
        //if PB1 is pressed, call test_pixel()
        if (PB_data_is_pressed_ASM(PB1)){
            test_pixel();
        }
        //if PB2 is pressed, clear char buffer
        if (PB_data_is_pressed_ASM(PB2)){
            VGA_clear_charbuff_ASM();
        }
        //if PB3 is pressed, clear pixel buffer
        if (PB_data_is_pressed_ASM(PB3)){
            VGA_clear_pixelbuff_ASM();
        }
```

ps2_keyboard.s:

The goal of this section is to display the scan code of the keys when the key is pressed. The subroutine read_PS2_data_ASM checks whether RVALID in the data register is 1. If it is 1, the subroutines returns 1, otherwise it returns 0. We extracted the RVALID bit (16th bit) by using an ADD statement and checked its value.

```
read_PS2_data_ASM:
    LDR R1, =PS2_DATA
    LDR R2, [R1]
    AND R3, R2, #0x8000
    CMP R3, #0x8000
    BEQ VALID
    MOV R0, #0
    BX  LR

VALID:
    AND R2, R2, #0x000000FF
    STRB R2, [R0]
    MOV R0, #1
    BX  LR


    .end
```

```
int main() {
    int x = 0;
    int y = 0;
    char data = 0;
    int counter = 0;
    int sample1 = 0x00FFFFFF;
    int sample0 = 0x00000000;

    while(1){
        /*if (counter<=240) {
            if (audio_ASM(sample1))
                counter++;
        }
        else {
            if (audio_ASM(sample0))
                counter++;
        }

        if (counter > 480) {
            counter = 0;
        }
        */


//KEYBOARD
            if (read_PS2_data_ASM(&data)){
                if (x>=79) {
                    x = 0;
                    y++;
                }
                VGA_write_byte_ASM(x,y,data);
                x+=3;
            }
```

In main.c, we first check if the address we use is valid by calling the read_PS2_data subroutine. If it is valid, we call VGA_write_byte to write the scan code of the key onto the screen.

Audio.s:

```
audio_ASM:
    PUSH {R4-R5, LR}
    LDR R1, =FIFO_SPACE
    LDR R2, =LEFT_DATA
    LDR R3, =RIGHT_DATA
    LDR R1, [R1]                    //load content of FIFO_SPACE into R1
    AND R4, R1, #0x00FF0000         //R4 now has WSRC
    AND R5, R1, #0xFF000000         //R5 now has WSLC
    CMP R4, #0                      //Compare WSRC with 0
    BEQ INVALID                     //if WSRC = 0, go to INVALID
    CMP R5, #0                      //Compare WSLC with 0
    BEQ INVALID                     //if WSRC = 0, go to INVALID
    STR R0, [R2]                    //store input int to LEFT_DATA
    STR R0, [R3]                    //store input int to RIGHT_DATA
    MOV R0, #1                      //return value = 1
    POP {R4-R5, LR}
    BX  LR                          //return


INVALID:
    MOV R0, #0                      //return value = 0
    BX  LR                          //return


    .end
```

The goal of this section is to play a 100Hz wave. For the audio_ASM subroutine, we extracted the WSRC and WSLC bit from the audio data register and checked whether they are equal to 0 to make sure that there is still space available to store data. If they are 0, then the subroutine returns 0. If there is still space, the subroutine stores the input integer to left and right data and returns 1.

```c
int main() {
    int x = 0;
    int y = 0;
    char data = 0;
    int counter = 0;
    int sample1 = 0x00FFFFFF;
    int sample0 = 0x00000000;

    while(1){
        if (counter<=240) {
            if (audio_ASM(sample1))
                counter++;
        }
        else {
            if (audio_ASM(sample0))
                counter++;
        }

        if (counter > 480) {
            counter = 0;
        }
```

Since the sampling rate of the De1-Soc computer is 48k samples per second and we want 100Hz wave. We need to input 240 ones and zeros for every cycle. In main.c, we constructed several conditional statements to write alternating ones and zeros. Once the counter exceeds 480, we clear the counter and start the sampling process again.

Potential Improvements:

Part 2 can be synchronized by a timer, which is more sufficient and it use less memory.

Challenges Encountered:

1. We encountered some challenges while dealing with the character buffer. Initially we simply used STR and LDR for accessing the content of the character buffer, but we realized later that we had to use STRB and LDRB for the character buffer. This problem made us spent a lot of time debugging our code and re-examining our logic.