Lab 4: High Level I/O - VGA, PS/2 Keyboard

For ECSE 324: Computer Organization

Report Written by

Antoine Wang (260766084)

Alfred Wang (260771591)

Due Nov 17th at 11:59PM

McGill University

1. Introduction

The aim of this lab is to implement new interfaces for more complex I/O such as VGA video signal and PS/2 keyboard. Comparing to last lab (Lab 3) which is producing and receiving information generates on the DEC-SO1 board, such as button presses and slider switches, this lab incorporates more sophisticated structures like buffers and 2-D arrays to store and send data. However, the basic operation of input and output, which is essentially done by the loading and storing commands in assembly, remains unchanged.

2. VGA

For the first section of this lab, an interface for processing VGA signals are required with 5 methods listed below.

2.1 VGA_clear_charbuff_ASM()

The buffers for both character and pixel display are 2-D arrays. Character buffer, with the base address located at 0xC9000000, has 80 horizontal components and 60 vertical components. The strategy used is **nested "for" loops** which iterates through a column of 60 components, then increment the horizontal counter and clear the vertical counter to arrive at the top position of the next column. The two for loops with horizonal iteration (x-loop) is shown below. Note the illustration is simply for looping, not complete code.

```
clear_char_xloop:
    ADD R2, R2, #1
    CMP R2, #79 // see if counter exceeds XMaximum
    BGT clear_char_Done
    MOV R3, #0 // every time move to a new row, clear the Y counter to zero
clear_char_yloop:
    CMP R3, #60 //y<=59
    BEQ clear_char_xloop

    //********clear the char pattern at that address*******************

    ADD R3, R3, #1 // looping
    B clear_char_yloop
```

Then, following the manual of the DEC-SO1 computer, the address pattern for the buffer is that the lower 7 bits is index for X (bit 0-6) and the following 6 bits is index for Y (bit 7-12). Using the pattern, every time the clearing algorithm passing down a unit, the y counter is first shifted 7 bits left to leave the space for X offsets and then combine with the current X counter and the base (0xC9…) to generate the exact address for the current position. The final step is storing a all-zero byte (R7 in the code) into that position to clear. Code segment for this operation is shown below.

```
LSL OFST, R3, #7        //shift Y offset (y counter) into corresponding position (bit 7-12)
ORR OFST, OFST, R2      // combine x offset and Y offset
ADD R4, BASE, OFST  //combine X ,y and the base to get final address
STRB R7, [R4]       //clear the char pattern at that address
```

2.2 VGA_clear_pixelbuff_ASM()

Clearing the pixel buffer follows the exact same algorithm as clearing that of the character one. There are slightly some differences, first for pixel buffer every element is 2 bytes in size since the data stored is a 16-bit RGB value. Thus, when iterating through every element the address is incrementing 2. This is achieved by truncating the last bit of the address unused. Secondly, the address pattern is also slightly changed. The size (resolution) of the pixel buffer is 320*240, meaning at last 9 bits of X offset and 8 bits of Y can represent all the element. Following the exact same logic, the address is bit 0 being the truncated bit, 1-9 for X offset and 10-17 for Y. The rest is the base bits. Code below shows how final address is generated.

```
LSL OFST, R3, #10   //Y address shift to bit 10-17
LSL R8, R2, #1      //X bit from 1-9, leave bit 0
ORR OFST, OFST, R8
ADD R4, BASE, OFST  //generate final address
STRH R9, [R4]       //clear by writing zeros
```

A possible improvement is that the current algorithm increments Y first but the natural address pattern increases X. Thus, if the X loop is the inner one, there would be no need to shift the Y address. Since every time a row is traversed, the next address corresponds to the first element in the next row.

2.3 VGA_write_char_ASM(x, y, char c)

This method takes in X-Y coordinates and the ASCII value of input character, then finds the corresponding position and store the ASCII value in. The generating of the address is same as previous part. The loop is not needed since only the selected spot is going to be overwritten. The only code added is that a check clause which ensures the X-Y coordinates are legitimate.

```
// check if the input coor are valid
// if not valid, exit the subroutine without writing anything
CMP X, #80
BGE write_char_failed
CMP X, #0
BLT write_char_failed
```

2.4 VGA_write_byte_ASM(x, y, char byte)

This method is similar to the previous one with input X and Y representing the same coordinate. The input byte is actually an integer which is going to be displayed as two hexadecimal number (00 to FF). For simplicity, the write char method in 2.3 is reused and this is achieved by portioning the input number into two 4-bit number (0-F). Then the value is turned into its ASCII value by indexing through an ASCII array. Finally, the subroutine of 2.3 is called. Below is the storing of the higher 4 bits.

```
LDR R7, =Ascii_Array
MOV R3, R2          // duplicated input char
LSR R2, R3, #4      // discard lower 4 bit in one of the char duplication
AND R2, R2, #15     // extract bit 4-7
LDRB R2, [R7, R2]   // Using the input number (essentially a number from 0-F)
BL VGA_write_char_ASM // use write subroutine to store the
```

A possible solution to even further simplify the code is to use STRH (store half-word) command. In this way the two-byte ASCII value for both two hexadecimal numbers are stored directly. The method will also be independent from the write char one.

2.5 VGA_draw_point_ASM(x, y, short colour)

For writing the screen with pixels, STRH command is actually used since the input "colour" is a 16-bit RGB value. Since the address of the pixel buffer is truncated (described in 2.2), storing halfwords directly follows the natural address pattern. All positions are now multiples of 2 in address value.

```
LSL OFST, Y, #10    //Y address shift to bit 10-17
LSL R5, X, #1       //X bit from 0-9, leave bit 0
ADD OFST, OFST, R5  //get final offset (X+Y)
ADD R4, BASE, OFST  //get final address (base +y+x)
STRH R10, [R4]      //store colour at the address (color is 16 bits, half word)
```

3. PS/2 Keyboard

The goal of this section of the lab is to get inputs using a PS/2 keyboard connected to the board. There is only one subroutine related to this section, which checks the RVALID bit first, and uses the value of RVALID to decide whether to get the input data. There is an input for this subroutine, which is the desired location to store the keyboard data if there is a valid input.

Both the data and RVALID bit are stored at 0xFF200100, which in this case, we named it PS2_Data. The RVALID bit is stored at the 16th bit, and the input data is stored at the lowest 8 bits. As the code shown below, we firstly check the value of RVALID bit. If RVALID is 0, there will not be any data received from the keyboard, and the return value will be 0. If RVALID is 1, we get the data from the lowest 8 bits from PS2_Data, store it at the desired location, and finally return 1.

```
read_PS2_data_ASM:
  //r0 is character pointer that the keyboard data will be stored at
  LDR R3, =PS2_Data   //R3 holds the address of keyboard data
  LDR R4, [R3]        //R4 holds the keyboard data
  MOV R1, #0x8000     //R1 holds a value that only the 16th bit is 1
  MOV R5, #0xFF       //lowest 8 bits are 1
  AND R2, R4, R1      //R2 will be 1 if rvalid is set to 1
  CMP R2, #0          //to see if rvalid is 1 or not
  BEQ INVALID         //if not 1, jump to invalid (end)

  AND R6, R4, R5      //get the lowest 8 bits from the keyboard data
  STRB R6, [R0]       //store data to the desired location
  MOV R0, #1          //return 1 if successfully get the data
  BX LR               //exit subroutine

INVALID:
  MOV R0, #0          //return 0 if no data received
  BX LR               //exit subroutine

  .end
```

4. Main

In order to test whether our program works, a C file is created to do tests for all the subroutines, and the result of the tests can be seen on a screen connected to the board using the VGA port.

For the VGA test, we have created three methods, each for testing VGA_write_char_ASM (), VGA_write_byte_ASM (), and VGA_draw_point_ASM (). Finally, we integrated these three test methods onto one large method, that can implement each test using pushbuttons and slide switches on the board.

4.1 Test for VGA_write_char_ASM ()

For the test method of VGA_write_char_ASM (), as the code shown below, we used a double nested for-loop to go through the entire screen and print consecutive ASCII characters column by column.

```
void test_char() {
  int x,y;                // x is row, y is column
  char c = 0;             // start with first ASCII as the input
  for (y=0; y<=59; y++) {             //start with column, then jump to next column
    for (x=0; x<=79; x++) {           //each row of current column
      VGA_write_char_ASM(x, y, c++);  //write character to each position
    }
  }
}
```

4.2 Test for VGA_write_byte_ASM ()

As the code shown below, the test method for VGA_write_byte_ASM () is very similar to VGA_write_char_ASM (). There are only two differences. The first one is, instead of printing ASCII characters, we print 8-bit hexadecimal expressions on the screen. The second difference is, instead of adding 1 to the x-coordinate, we add 3 to the x-coordinate after each print. The reason for adding 3 is that the hexadecimal expression occupies 2 unit-spaces, and we add another empty space at the back to

separate hexadecimal expressions between each other; therefore, a total of 3 is added to the x-coordinate.

```
18 void test_byte() {
19   int x,y;                  // x is row, y is column
20   char c = 0;               // start with 0 as the input, which will be translated to hexi on the screen
21   for (y=0; y<=59; y++) {          //start with column, then jump to next column
22     for (x=0; x<=79; x+=3) {       //each row of current column, x increases by 3, 2 for hexi, 1 for space
23       VGA_write_byte_ASM(x, y, c++);    //write character
24     }
25   }
26 }
```

4.3 Test for VGA_draw_point_ASM ()

The test method for VGA_draw_point_ASM () is also very similar to VGA_write_char_ASM (), as the code shown below. The only difference is, instead of going through each unit-space, we go through all the pixels and print them with different colors.

```
28 void test_pixel() {
29   int x,y;                      // x is row pixel, y is column pixel
30   unsigned short colour = 0;    //color to be drawn on the screen
31   for (y=0; y<=239; y++) {      //start with column, then jump to next column
32     for (x=0; x<=319; x++) {    //each row of current column
33       VGA_draw_point_ASM(x,y,colour++);       //print the color for the pixel
34     }
35   }
36 }
```

4.4 Integration

Finally, as the code shown below, we put all the method written above together into one integrated method, using pushbuttons and slide switches to implement each test. When the first button is pushed as well as the slide switch is off, VGA_write_char_ASM () will be tested; however, if the slide switch is on, VGA_write_byte_ASM () will be tested. If the second button is pushed, VGA_draw_point_ASM () will be tested. Button 3 and 4 are responsible for clearing characters and colored pixels on the screen.

```
38 void vga() {
39   while (1) {
40     if (read_PB_data_ASM() == 1){             //if the first button is pushed
41       if(read_slider_switches_ASM() == 0) {   //if no slide switch is on
42         test_char();                          //call test char
43       }
44       else {
45         test_byte();                          //otherwise, call test_byte
46       }
47     }
48     else if (read_PB_data_ASM() == 2) {       //if the second button is pushed
49       test_pixel();                           // call test pixel
50     }
51     else if (read_PB_data_ASM() == 4) {       //if the third button is pushed
52       VGA_clear_charbuff_ASM();               //call VGA_clear_charbuff_ASM()
53     }
54     else if (read_PB_data_ASM() == 8) {       //if the fourth button is pushed
55       VGA_clear_pixelbuff_ASM();              //call VGA_clear_pixelbuff_ASM()
56     }
57   }
58 }
```

4.6 Test for read_PS2_data_ASM ()

The test for read_PS2_data_ASM () is straightforward. As the code shown below, we check whether the keyboard input is valid or not. If it is valid, we use VGA_write_byte_ASM () to print the hexadecimal value of the keyboard input on the screen and update the x and y coordinate for the next input.

```
67    while(1) {
68      if (read_PS2_data_ASM(&value)) {          //if data can be read successfully
69        VGA_write_byte_ASM(x, y, value);        //write the data to the designated position
70        x += 3;                                 //space automatically (2 for data, 1 for space)
71        if (x > max_x) {                        //if the row is full
72          x = 0;                                //reset x to 0
73          y += 1;                               //jump to next row
74          if (y > max_y) {                      //if all rows are full
75            y = 0;                              //reset to first row
76            VGA_clear_charbuff_ASM();
77          }
78        }
79      }
80    }
81  }
```