William Zhang
ID: 260975150

Friday, December 2, 2022

ECSE 324 – Lab 3 Report

In lab 3, the I/O capabilities of the DE1-SOC Simulator are used to display pixels and characters using the VGA controller and to accept keyboard input via the PS/2 port. Drivers are created for each topic and tested using provided test code. The pixel buffer is 320 pixels wide and 240 pixels high, whereas the character buffer has a width of 80 characters and a height of 60 characters.

Task 1: Drawing things with VGA

Problem

The first task of lab 3 is to implement a set of driver functions to control the screen (the pixel and character buffers). The first function, 'VGA_draw_point_ASM', draws a point on the screen with the colour as indicated in the third argument by only accessing the pixel buffer memory. It should have the x and y positions as well as the 16-bit RGB color code as inputs. Colours are encoded as 16-bit integers that reserve 5 bits for the red channel, 6 bits for the green channel and 5 bits for the blue channel. The second function, 'VGA_clear_pixelbuff_ASM', clears all the valid memory locations in the pixel buffer. It takes no argument and returns nothing. The third function, 'VGA_write_char_ASM', writes the ASCII code passed in the third argument (r2) to the screen at the (x,y) coordinates given in the first two arguments. The last function, 'VGA_clear_charbuff_ASM', clears all the valid memory locations in the character buffer. It takes no argument and returns nothing.

Solution

The pixel buffer pixels are addressed by using the combination of a base address and an x,y offset. The base address is 0xC8000000 and the (x,y) offset of size 18 bits consists of 8 y-coordinate bits, 9 x-coordinate bits and 1 unused bit. The first function, 'VGA_draw_point_ASM', is implemented by writing to the memory address #0xC8000000. The X-coordinate is shifted by 1 to the left and is added to the memory address value register whereas the Y-coordinate is shifted by 10 to the left and then added to that register. The color code for the point is then stored in the memory address associated with the memory address value register. The second subroutine, 'VGA_clearpixelbuff_ASM', sets the initial x and y coordinates to 0 and sets the color value to white. It then branches to loop, which calls 'VGA_draw_point_ASM' and goes through every (x,y) coordinate in the pixel buffer.

The character buffer character slots are addressed using by using the combination of a base address, 0xC9000000, and an (x,y) offset. The (x,y) offset has a size of 13 bits, which consists of 6 y-coordinate bits and 7 x-coordinate bits. The function 'VGA_write_char_ASM' first checks if the input values for x and y are within the bounds. It then adds the coordinate values to the register which contains the base address. It stores the ASCII code of the character in the memory address associated with the memory address value register. The 'VGA_clear_charbuff_ASM' sets the x and y coordinates as well as the character code to 0 and iterates through a loop which sets each character buffer address to 0.

Improvements

A possible improvement includes iterating through the loop recursively instead of iteratively when clearing the pixel and character buffers, which could reduce the size of the code and decrease the time needed by the function to achieve its purpose since there would be fewer calls in general.

Task 2: Create a PS/2 driver

Problem

The second task of lab 3 is to implement a function which checks the PS/2 data register and stores the value at the address in a pointer argument. If the data is valid, the subroutine should return 1 and 0 otherwise. The DE1-SOC receives keyboard input from a memory-mapped PS/2 data register at address 0xFF200100. The register has an RVALID bit that states if the current contents of the register are a new value from the keyboard.

Solution

The function is implemented by setting the value in register r4 to ff200100, which is the address of the data register. The value at that address is then loaded into r4. The subroutine then checks the RVALID bit, which can be accessed by shifting the PS/2 data register value by 15 bits to the right. If the RVALID bit is 1, the function calls the subroutine 'storedata'. If it is 0, the function moves the value 0 into r0 and returns. The 'storedata' subroutine begins by isolating the 8 first bits of the register r4, which contains the data from the PS/2 register. The data from r4 is then stored at the address indicated by the pointer argument in r0. The value 1 is then moved into r0 and the function returns.

Improvements

Possible improvements that could be made to the program include the removal of the 'storedata' branch and finding an efficient way to move the value 0xFF200100 into register r5. The 'storedata' branch can be avoided by returning from the function read_PS2_data_ASM if the RVALID bit is set to 0 or continuing with the rest of the calls that are in 'storedata' if the RVALID bit is set to 1. Also, the value 0xFF200100 must be moved into register r5 using 3 mov calls and 2 add calls. Instead, the function could do it by moving 0x100 into r4 and adding 0xFF2 to r4 with a logical shift left of 20 bits.

Task 3: Labyrinth

Problem

The third task of the lab is to implement a labyrinth game. The layout of the obstacle course is saved in memory in a 3D structure with dimensions The user should be able to select a course by clicking on one of the numbers 1-9 on the keyboard. The goal of the game is to make the ampersand '&' character reach the exit. The ampersand can be moved using the keyboard arrows through the empty tiles. In addition, the ampersand cannot move through a black obstacle.

To implement the labyrinth, 8 subroutines need to be created: 'VGA_fill_ASM', 'draw_grid_ASM', 'draw_ampersand_ASM', 'draw_exit_ASM', 'draw_objects', 'fill_grid_ASM', 'move_ASM', and 'result_ASM'. The ampersand always starts at (0,0) and the exit is always located at (11,8). The ampersand should also not move beyond the boundaries, and it cannot move diagonally.

The first subroutine, 'VGA_fill_ASM', does not take any arguments and fills the VGA panel with a solid colour. The second subroutine, 'draw_grid_ASM', does not take any arguments and draws the empty 9-by-12 grid. The third subroutine, 'draw_ampersand_ASM', takes as arguments the (x,y) coordinates of the mark and draws the '&' mark. The 'draw_exit_ASM' subroutine also takes as arguments the (x,y) coordinates of the mark and draws the 'X' mark. The fifth subroutine, 'draw_objects', takes the (x,y) coordinates of the mark as arguments and draws the obstacles marks. The next subroutine, 'fill_grid_ASM', takes as an argument the number of the obstacle course and draws the obstacles' square marks according to the obstacle course chosen. The seventh subroutine, 'move_ASM', takes as an argument the stored input from the keyboard and moves the '&' according to the input if there is no obstacle in that position. If there is an obstacle, the ampersand does not move to that position. The last subroutine, 'result_ASM', has as input the result of the game and writes the result of the game.

The game is won when the ‘&’ symbol reaches the exit. A text appears on the screen for 10 seconds before the screen reverts to the empty grid layout to start another game.

Solution

The solution is implemented using three designated memory spaces and three variables. The three spaces are ‘keyboarddata’ (size 4), ‘ampersandvalue’ (size 4), and ‘playedmaze’ (size 432). The three variables are ‘LOAD_VALUE’ (0xFFFEC600), ‘CONTROL_REGISTER’ (0xFFFEC608) and ‘INTERRUPT_STATUS_REGISTER’ (0xFFFEC60C). The program also needs the input_mazes data for the nine obstacle courses. The game has a size of 196 by 148 pixels.

For the lab’s purposes, the ‘read_PS2_data_ASM’ subroutine’s ‘storedata’ branch must be changed so that the branch is able to filter through the unnecessary input codes and only store the relevant ones. If the program recognizes a code, it responds accordingly. If the code is E0 or F0, the routine calls read_PS2_data_ASM again to read the next code since those two codes are not useful for the maze game. If the code is 0x75, the program stores the value 11. A value is stored for every arrow key (11 to 14) and every number from 1 to 9 (0 to 8). The values for the mazes are 0 to 8 because those values can directly be used as indexes when copying the maze to be played.

‘VGA_fill_ASM’ is the same as ‘VGA_clearpixelbuff_ASM’ but it stores blue instead of white in each pixel address. The subroutine ‘draw_grid_ASM’ moves the values 0,0,0,196 and 4 into registers r0 through r4. It then calls the branch ‘columnloop’, which draws columns of black pixels. Each of the resulting columns has a total size of 4 and the empty spaces have a size of 12. The program also calls the branch ‘rowloop’, which draws rows of black pixels. Like the columns, each of the rows has a total size of 4 and the empty spaces have a size of 12. That means every slot is of size 12x12.

The ‘draw_ampersand_ASM’, ‘draw_exit_ASM’ and ‘draw_obstacles_ASM’ subroutines move the ASCII code value of the characters to be displayed into register r2 and call ‘VGA_write_char_ASM’. The ‘fill_grid_ASM’ subroutine loads the address of the played maze into r3 and moves the initial coordinates (2,2) into registers r0 and r1. It then calls ‘draw_objects’. This subroutine loads the value of each labyrinth component and checks if it is an obstacle, an ampersand or an exit. It calls the appropriate subroutine when checking each labyrinth slot.

The ‘move_ASM’ subroutine loads the values of ‘ampersandvalue’ and ‘keyboarddata’ into registers r9 and r5, respectively, and moves the needed initial values into registers r6 through r8. It then compares the keyboarddata to each movement and moves the ampersand by switching the value of the ampersand with that of the object at the other position. The program also checks if the movement would result in the ampersand moving out of the bounds of the labyrinth and prevents it from moving out if it is the case. If the ampersand moves to the exit, ‘result_ASM’ is called. The function ‘result_ASM’ writes the message “VICTORY!” under the labyrinth and configures the timer to have a load value of 1 second and a prescaler value of 9 so that the timer counts 10 seconds. The timer is then started and the program branches to ‘checkloop’.
The ’checkloop’ branch checks if the timer has counted down and calls ‘_start’ if it is the case (go back to the initial state).

The program starts by clearing all the values in the PS/2 Data register by calling read_PS2_data_ASM. If then initializes all the register values to 0. It then clears the screen by calling ‘VGA_clear_pixelbuff_ASM’ and ‘VGA_clear_charbuff_ASM’. If fills the screen with the background and the grid by calling ‘VGA_fill_ASM’ and ‘draw_grid_ASM’. The program calls ‘wait’, which is a loop that calls ‘read_PS2_data_ASM’ and returns only if data was stored by the subroutine. The function then loads the value of the maze to be played in r0 and multiplies it by 0x1b0 to get the address of the appropriate maze. The program then copies every value of the labyrinth into the address ‘playedmaze’ using the ‘copy’ subroutine. The function ‘fill_grid_ASM’ is then called to display the labyrinth. After calling ‘wait’ once to clear the PS/2 input register, the function calls ‘play’, a subroutine which calls ‘wait’ and calls ‘move_ASM’ if there is an input from the keyboard. The ‘play’ subroutine loops back to itself if there is a move.

Improvements

Multiple improvements could be made to the program. To clear the data from the PS/2 data register, the subroutines ‘read_PS2_data_ASM’ and ‘wait’ are called but they could be unnecessary in many cases. The code could be shortened by calling those two subroutines effectively. Also, many loops could be combined so that space is saved since the loops essentially do the same thing but with different constraints. By changing those constraints outside the loop, it is possible to avoid having multiple different structures with the same general function. Additionally, the management of the registers could be more efficient. Instead of reinitializing them every time ‘_start’ is called, the effective use of pushes and pops can remove the need for ‘safety reinitialization’.

In conclusion, this lab consisted of implementing a labyrinth game using keyboard input devices as well as the VGA screen. Both the pixel buffer and the character buffer were needed. It also required the use of a timer, which was implemented in lab 2.