**Drawing things with VGA**

Approach

The subroutine VGA\_draw\_point\_ASM takes A1 and A2 as parameters, respectively, the x and y coordinates of the point drawn on the screen, and A3, the pixel colour (takes 2 bytes). It verifies if the values are within the allowed boundaries (between 0 and 319 for x and 0 and 239 for y). If false, it returns. Otherwise, it shifts x by 1 bit to the left and y by 10 bits to the left to get the proper address offset of the pixel buffer. Then, the address of the pixel buffer, 0xc8000000, is loaded, and A1 and A2 are added to it; that way, we have the pixel address of coordinates x and y. A3 is stored at this address as a half-word, then the function returns.

The subroutine VGA\_clear\_pixelbuff\_ASM iterates through the pixel buffer with two nested for loops and calls VGA\_draw\_point\_ASM with A3=0 for every pixel.

The subroutine VGA\_write\_char\_ASM takes A1 and A2 as parameters, respectively, the x and y coordinates of the character drawn on the screen, and A3, the ASCII code (takes 1 byte). It verifies if the values are within the allowed boundaries (between 0 and 79 for x and 0 and 59 for y). If false, it returns. Otherwise, it shifts y by 7 bits to the left to get the proper address offset of the screen. Then, the address of the character buffer, 0xc9000000, is loaded, and A1 and A2 are added to it; that way, we have the address of the character buffer of coordinates x and y. A3 is stored at this address as a single byte, then the function returns.

The subroutine VGA\_clear\_charbuff\_ASM iterates through the character buffer with two nested for loops and calls VGA\_write\_char\_ASM with A3=0 for every character.

Testing strategy

I ensured first that the boundaries were properly applied by placing a pixel at the four corners of the screen and checking if their positions were correct. Then some attempts to place pixels just outside the screen were made to be sure the subroutine was returning without storing the pixel at an invalid address.

Then, VGA\_clear\_pixelbuff\_ASM was called to ensure that all pixels were properly cleared.

The same testing methodology was made for the character buffer.

Then, the test code was run, and the actual result was compared to the expected result found in the lab document.

Challenges

The only challenge was to ensure that the correct number of bits was stored at the buffer address since some necessary bits for the buffer are lost (like colour levels, for example) if it is too small. If it is too big, the program may write over neighbouring cells with the wrong data. Thus, colours had to be stored as a half-word, and characters had to be stored as a byte.

Shortcomings and improvements

To clear the buffers, the subroutine to write on the buffer is called for each cell for convenience and clarity, but to save performance (calling new subroutines is heavy on performance) and registers, instead, one for loop can be used to write 0 on all the buffer’s length directly.

**Reading keyboard input**

Approach

The subroutine read\_PS2\_data\_ASM takes A1, the address where the data will be stored, as a parameter and checks the RVALID bit in the PS/2 data address. To do so, it loads the data of the PS/2 at 0xff200100, does a logical shift to the right of 15 bits and execute a TST instruction with 0x1. If the result is 1 (RVALID is 1), the data collected is stored to A1, then the function returns 1. If it is 0, the function only returns 0.

Testing strategy

I first ran the test code provided. Then, different buttons were pressed, and the codes shown on the screen were compared to the ones on the table to ensure they corresponded to the keys pressed. I ensured that the break code also appeared when I released the key. Then, I tested if one short keystroke registered only one make and break code when released and if one long-pressed key repeatedly sent the make code.

Challenges

The only challenge was to ensure that the data of PS/2 was stored to A1 only when the data was fresh and that it returned 1 if it was the case.

Shortcomings and improvements

The subroutine doesn’t discern between a make code and a break code, so if it were used to read the keyboard input and to count a specific character by its make code, the character would be counted twice per keystroke, one when it is pressed and one when it is released (following an F0 code). An improvement would be that when it reads F0 (break code), the subroutine would wait until it receives the following code (the make code of the key released), stores it to A1 and returns 2. That way, we can know which key was released and ignore the data if we only want to read the keys entered.

**Game of Life**

Approach

The board is stored in an array of the length of 192 (to represent the grid of 16x12). The cursor position is stored at V2 and holds the index of the cursor in the array. V1 holds the potential next cursor index if the position changes and starts at 0. V5 (starts at 0) holds a flag that will signal if the next PS/2’s read data will be ignored, and V6 holds the initial SP address (before the game starts and uses the stack).

The program initializes by calling multiple subroutines. First, VGA\_clear\_charbuff\_ASM is called to remove any character on the screen in case there was some, then VGA\_clearwhite\_pixelbuff\_ASM is called. This subroutine is the same as VGA\_clear\_pixelbuff\_ASM, except all the pixels are set to white (0xFFFF) instead of black (0x0). Then, GoL\_draw\_grid\_ASM with A1=0x0 is called.

This subroutine takes A1 as a parameter, which will be the colour of the grid. Then, two nested for loops are used to iterate between all the grid squares. The pixel coordinates for the first square point at the top left of the square. X and Y are incremented by 20 in their proper loops so that the coordinates point to the top left corner of the next square. For each square, an inner outline of 1 pixel wide is drawn. To do so, a one-pixel-wide line is drawn at the four sides of the square with their endpoints at the corners of the square. The endpoints of the line and the colour are parameters, respectively, A1 (x1), A2 (y1), A3 (x2), A4 (y2) and A5 (stored in the stack) for the subroutine VGA\_draw\_line\_ASM. Each loop ends when the pixel coordinates of the variable loops are out of bounds.

The subroutine VGA\_draw\_line\_ASM takes A1 (x1), A2 (y1), A3 (x2), A4 (y2) and A5 (stored in the stack) as parameters, the endpoints coordinate and the colour of the line, respectively. At the start, it makes sure that V1 (x1) and V2 (y1) hold the largest value and V3 (x2) and V4 (y2) are the smallest. Otherwise x1/y1 and x2/y2 are switched so that V1>V3 and V2>V4. After that, if y1=y2, a horizontal line is drawn by calling VGA\_draw\_point\_ASM with the current V1 and V2 and A5 (colour) as the parameters and decrementing x (V1) by 1 until it is lower than V3 (x2). If x1=x2, a vertical line is drawn by calling VGA\_draw\_point\_ASM with the current V1 and V2 and A5 (colour) as the parameters and decrementing y (V2) by 1 until it is lower than V4 (y2).

Then, GoL\_draw\_board\_ASM is called. This subroutine draws the board by colouring each square with its correct colour based on the current board array. Two nested for loops are used to iterate between all the grid squares, and the loops variables represent the square coordinates (0 ≤ x < 16, 0 ≤ y < 12). For each square, its state is loaded from the board state array. To get the offset, the y coordinate is multiplied by 16 (the array index moves by a multiple of 16 for the y-axis), and the value of x is added to it. The result is multiplied by 4 (every array element takes 4 bytes) and then added to the base address of the board state to get the effective address of the cell status. If the status is 1, a black square is drawn by calling GoL\_fill\_gridxy\_ASM with the square coordinates and the black colour as parameters. Otherwise, a white square is drawn.

The subroutine GoL\_fill\_gridxy\_ASM takes A1 (x), A2 (y), and A3 as parameters, the square coordinates and the square’s colour, respectively. It converts the square coordinates to pixel coordinates by multiplying them by 20. Then, to prevent drawing the square on the outline (and redrawing the grid each time the board is updated), the square will be inside the outline (one pixel smaller on all sides compared to the outline). To do so, the pixel coordinates are incremented by 1, so they point inside the top-left corner of the square’s outline. The bottom-right corner of the square to be drawn is determined by adding 17 to the pixel coordinates so they point inside the bottom-right corner of the square’s outline. The subroutine VGA\_draw\_rect\_ASM is called with the two corner points and the colour as parameters.

This subroutine takes A1 (x1), A2 (y1), A3 (x2), A4 (y2) and A5 (stored in the stack) as parameters, the top-left point, the bottom-right point, and the colour of the rectangle, respectively. It uses two nested for loops, starting at the top-left corner, to iterate through all the pixels containing the rectangle and call VGA\_draw\_point\_ASM at each pixel with the colour given by A5. The loops end after the last pixel at the bottom-right corner of the rectangle is drawn, and the subroutine returns.

After, the SP value is stored to V6 for later use, and the cursor is drawn on the screen (starts at 0). To do it, the status of the cell where the cursor is loaded from the board array (to get the effective address, the cursor’s index is multiplied by 4 (size of an element) and added as the offset to the board base address). If the cell is inactive (0), a blue square is drawn at this cell by calling GoL\_fill\_gridxy\_ASM. Otherwise, a red square is drawn. Since the subroutine takes square coordinates, the index of the cursor is converted into these coordinates. To retrieve the x value, the operation AND is executed on the index value with 0xF (essentially index mod 16). To retrieve the y value, a logical shift right of 4 bits is done to the index value (essentially doing an integer division of 16). Then, the main loop is entered.

At the start of the loop, the potential next cursor position is saved to the current cursor position. Then, the function read\_PS2\_data\_ASM is called to poll the keyboard with the data stored at KEYPRESSED. If the function returns with 1 (fresh data retrieved), the key code is loaded from KEYPRESSED and is pushed on the stack. Then, if V6 is 1 (ignore the next code flag), the stack is popped by one to discard the code in the stack, and V6 is set to 0. After that, SP is compared to V6 (initial stack address). If it is the same (no key code stored on the stack), the iteration ends and the loop restart. Otherwise, the code is extracted from the stack and loaded. If it is 0xF0 (break code), the ignore next key flag is set to 1.

If it is the make code for “w”, V1 is subtracted by 16 (go up by 1) and skips to KEYENTERED branch. If it is the make code for “s”, V1 is added by 16 (go down by 1) and skips to KEYENTERED branch. If it is the make code for “a”, the x value of V1 is extracted to a temporary variable by doing an AND operation with 0xF to the index stored at V1 (mod 16). Then x is decremented by 1. If it is within bounds (equal or bigger than 0), V1 is decremented by 1, and the program skips to KEYENTERED branch. If it is the make code for “d”, the x value of V1 is extracted to a temporary variable. Then x is incremented by 1. If it is within bounds (equal or smaller than 15), V1 is incremented by 1, and the program skips to KEYENTERED branch.

If it is the make code for the space bar, the cell’s status is loaded at the cursor location (board status base address + cursor index\*4). An exclusive OR operation with 1 is done to the status to toggle the status of the cell and then stored at the cell’s status address. The cursor is then redrawn using the same way as previously mentioned.

If it is the make code for “n”, the game state is updated. Two nested for loop is used to iterate through all the cells with (0 ≤ x < 16, 0 ≤ y < 12). For each cell, the values of the status of its neighbouring cells are summed up, giving us the number of active neighbour cells. To get the values, the array index of the current cell is obtained by adding to the x value the y value multiplied by 16. To get the cell on the top/bottom, 16 is subtracted/added to the current cell’s index. If the resulting cell address is out of bounds (index<0 or index>192), the resulting cell’s status is ignored (not added to the total), and the next neighbour cell is checked. To get the cell on the left/right, 1 is subtracted/added to the current cell’s index. If the resulting cell address wraps to the opposite side (x=15 when checking left or x=0 when checking right, x is obtained by doing mod 16 to the index), the resulting cell’s status is ignored, and the next neighbour cell is checked. The neighbouring cells at the four corners of the current cell are accessed by combining the additions/subtractions to the index and the bounds checks. For example, to access the top-left cell, 16 is subtracted from the index. If the index is smaller than 0, the neighbour is ignored. Otherwise, 1 is subtracted from the index. If the x value is 15, the neighbour is ignored. Otherwise, the neighbour cell’s status is added. After all neighbouring cell’s statuses are checked, the current cell’s status updates depending on if the cell is active or not. If it is active, with 0 or 1 active neighbour, the cell’s status becomes 0. With 2 or 3 active neighbours, the cell’s status becomes 1. With more than 4 active neighbours, the cell’s status becomes 0. If it is inactive, with 3 active neighbours, the cell’s status becomes 1. The new status is stored in a temporary array representing the next board state. When all the cells are updated, the temporary array is copied back to the current board status array. Then, GoL\_draw\_board\_ASM is called to update the board on the VGA screen.

After that, the program reaches the KEYENTERED branch. A boundary check ensures that the potential cursor index (V1) is between 0 and 191, inclusive, and if the cursor changes position (V1!=V2). If it is, the cursor is updated on the screen. Otherwise, the cursor is not drawn; the current index (V2) is copied to the potential cursor index (V1) (revert to the current index) and the loop restarts. If the cursor is drawn, the cell’s status at the cursor is loaded. If inactive, a white square is drawn at its location by retrieving the square coordinates from the index (see previously mentioned method) and calling GoL\_fill\_gridxy\_ASM for the colour white (0xFFFF). If it is active, the square drawn is black instead. Then, the potential cursor index is copied to the current cursor index to register the new cursor position, and the cursor is drawn using the previously mentioned method. Finally, the loop restarts.

Testing strategy

I first ensured that VGA\_draw\_line\_ASM was working correctly. I tested if it successfully drew vertical lines, horizontal lines, points, lines on the edges and drawing lines while the x and y values of the start point were smaller than the x and y values of the endpoint. Then, I ensured VGA\_draw\_rect\_ASM worked adequately. I tested drawing rectangles at the edge with multiple colours and verified if the rectangles were properly filled and in the correct coordinates. When implementing the cursor, I ensured that the cursor couldn’t go out of bounds or wrap back to the opposite side when moving into the edges and was moving only once when a movement key was pressed once. I also ensured the cursor would move continually when a movement key was long-pressed. I ensured that the space bar correctly toggled the cell it was on and that a cell’s status was properly drawn after the cursor passed over it. To test the update status, I replicated on the board each possible condition and ensured that the cells updated correctly. I also ensured that cells on edge updated correctly and didn’t affect cells on the opposite side. Finally, the board status on the lab document was replicated to verify if the updated board was the same as the one shown in the document.

Challenges

One challenge was to figure out how to draw the board. Drawing a pixel-wide line between each square would create a different offset on each square and would make drawing squares more difficult as coordinates are harder to calculate. I decided it would be simpler to draw an inner outline on each square, thus effectively making a 2 pixels separation between each square. Another challenge was converting between x and y coordinates and the index of the array. Coordinates are easier to work conceptually, but the only way to access a cell’s content was in an array, so we needed the index of the cell. That means conversion was inevitable in some parts. Converting from coordinates to the index is easier than the other way. It only requires adding x to y\*16 since y moves in multiple of 16 and x is between 0 and 15, inclusive.

Conversely, y can be retrieved from the index by doing an integer division of 16 (aka logical shift right of 4 bits), and x can be retrieved by an AND instruction with 0xF (mod 16). Since hardware division is impossible, if x were not a power of 2, this operation would have been much trickier (no shift possible to divide). A third challenge was ensuring the cursor moved only once per short keystroke. Because of the latency between the first break code and the make code, just making the program ignore the data fetched by the next read\_PS2\_data\_ASM is insufficient, and a key was registered two times, one for when it is pressed and one for when it is released. The solution is to store every code in a stable memory location (in this case, the stack) and to read the stack when it is not empty, removing the next pushed code when the first break code is read to prevent the double registration of a keystroke.

Shortcomings and improvements

A shortcoming of the program is that the grid size and boundaries are hard-coded, so the program can’t scale up or down, and the grid’s dimensions can’t change.

While coordinates make coding much easier conceptually, only using indexes for the logic would improve the program’s performance since all the conversions would not be needed. Moreover, only one register would be needed to hold the position instead of 2. Additionally, it would take only one for loop instead of two nested to iterate through all the cells. Another performance improvement can be when drawing the board with GoL\_draw\_board\_ASM; instead of redrawing every cell, for each cell, the square can be filled only if the current colour is not the same as the colour the cell would be filled.