Theodore Peters 260919785
Ecse 324, Winter 2022
Lab 3 Report

Part 1: VGA Drivers

The VGA drivers for setting character or text are quite simple; they merely need to set the correct bits of the base address with the input coordinates and set the halfword or byte, respectively, in memory. The character writing procedure additionally checks that the provided input is within the bounds of the character buffer; this is done with an unsigned compare. An unsigned compare is used because the range of valid values is 0 to 79 (for width), so only a single unsigned comparison against 79 is needed to determine if the x input falls in this range, as a negative x value treated as an unsigned int will be greater than 79. A signed comparison, on the other hand, would require checking both ends of the range. Because of the simplicity of these functions, only the scratch registers need to be used so nothing is added to the stack.

For the functions to clear pixel and character buffers also behave very similarly to each other, but iterate over different sized loops. However, for efficiency purposes, these functions do not call the corresponding functions for setting pixels or characters. This is because each of these functions set a single halfword or byte, and because the memory for these buffers are stored contiguously, this would require accessing the same memory 2 or 4 times, respectively. It is faster to set each full word to zero for all of the valid words; because the total number of pixels is even, and the character width is divisible four, there are no parity issues at the edge with a word only encoding the color information in its first half. Again, these methods only utilize the scratch registers, so the stack is not needed.

Testing for these drivers was already provided by the starting code, which implemented a main method heavily reliant on their functionality. The sample code was run multiple times to ensure that the buffer could be repeatedly cleared and set, but further testing was not necessary.

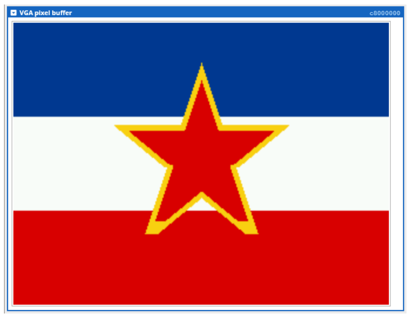
Part 2: PS/2 Driver

The single PS2 driver required for this deliverable first reads the PS/2 data register and checks the RVALID bit. If this is zero, zero will be immediately returned in a1, indicating that no valid keycode could be read. Otherwise, the character information will be stored at the address indicated at a1. Because the character is represented as the final 8 bits of the PS/2 data, no additional computation is required to ensure that only this information is stored, as a `strb` operation will already store only these bits. Finally, 1 will be returned in this case.

Once again, the starting code provided all the needed testing functionality. It was verified the PS/2 driver worked successfully based on the provided table, with both make and break codes for keys. It was also determined that holding down keys would result in the keycode being rendered multiple times, after a delay.

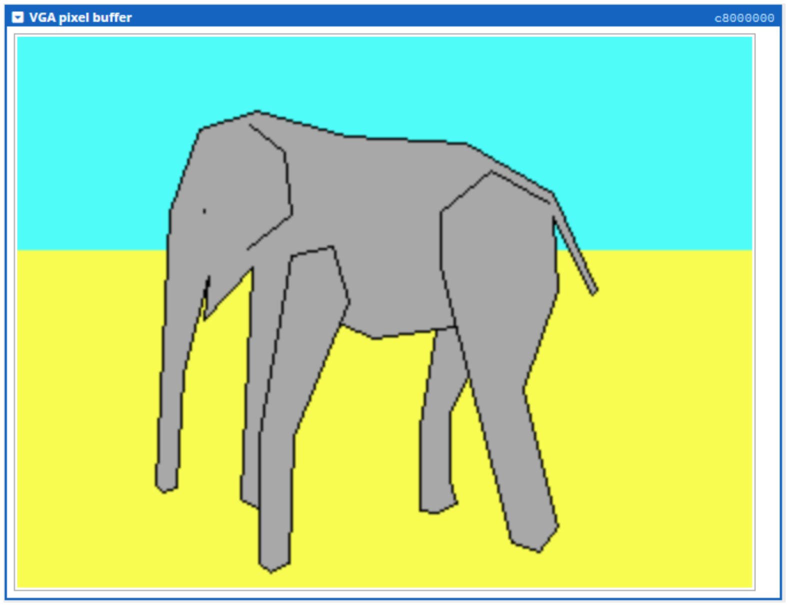
Part 3: Vexillology

For the real life flag, I chose the flag of Yugoslavia:



It should be noted that the bottom of the star is cut off, this is an issue with the provided methods for drawing stars. The colors used by the flag are stored as words, and loaded into a register and then placed on the stack, for the rectangle drawing methods, as each different color is needed. Because drawing this flag requires calling methods for drawing stars and rectangles, it was necessary to push the link register to the stack at the beginning and end of the method. To ensure that writing the flag colors to the stack will not overwrite the link register, the stack pointer is decremented by a word at the beginning of the method to allow writing the color information there. It is then incremented again at the function end so the link register can be popped to return.

For my imaginary flag, I drew this pleasant elephant:



The method for drawing the elephant is much the same in terms of using the stack, except it makes calls to implemented methods for drawing lines and floodfill (for the three discontinuous regions: the body, a front and back leg) in addition to the rectangle drawing calls made for the background. Because the signatures for these are similar to the provided drawing methods, calling them was similar.

The drawing of lines was done with Bresenham's line algorithm, which conveniently does not require any floating point calculations. The implementation is fairly standard, it divides the input coordinates based on octet to and reduces them to two cases, one where the change in x coordinate is greater than the change in y, and one where it is less, and then calls the VGA driver accordingly to draw pixels either adjacently or diagonally based on the slope. This method takes the x and y coordinates for the start and end of the line in the first four registers (the order is not important as they will be conditionally reordered to reduce the problem to 2 octets), and the color on the stack. After pushing several of the saved registers to the stack to avoid clobbering, the color needs to be retrieved from the stack; however, as the stack now points to the last register pushed to it, load instruction is given an index of 20 in addition to the base stack pointer, as there are 5 registers pushed, 4 bytes each. The line drawing method makes repeated calls to part 1's `VGA_draw_point_ASM`; because this method overwrites the value of the scratch register `a1`, this register needs to be pushed to the stack before calling and popped afterward to keep its value, as per the register conventions.

For coloring in the elephant, a simple, naive implementation of floodfill was used using the program stack as a stack to store coordinates. As arguments, the function takes x, y, fill color, and boundary color in `a1` - `a4`. Here, boundary color indicates the color of the border where floodfill should halt; in my case, this was black for the outline of the elephant. The top coordinate on the stack is taken, and adjacent coordinates that have not yet been filled are filled then added to the stack, until the stack is empty. Whether the stack is empty is checked by comparing the stack pointer to a frame pointer taken at the beginning of the function, when a coordinate has yet to be added to the stack.

This implementation is not reusable for any arbitrary region of the buffer for two reasons. Firstly, for certain large regions, the program stack will overflow and start writing to unrecognized device memory. To mitigate this problem, the stack usage was halved by storing the x and y coordinates as half words inside a single word when writing them to the stack, rather than writing both words. However, it is still possible that certain regions will cause the stack to overflow. The second issue is that no checks were implemented along the boundaries of the screen, so attempting to fill a region that is not internally enclosed inside the pixel buffer will cause it to overflow. This is an improvement that should be added for a more general algorithm, but in my case was unnecessary as the region I am filling is totally inside the buffer, and regions bordering the edge can still be filled by just drawing a line of the fill color around their edge before calling the floodfill function. These edge case comparisons were excluded for the sake of simplicity. Because the floodfill loop will terminate when the stack pointer is in the same position it was at the start of the loop, we ensure the saved registers can then be popped from the stack, so no registers will be clobbered despite the heavy stack modifications.

Again, the starter code provided much of what was required for testing; to ensure that the line drawing algorithm was functioning correctly, the lines drawn were compared against a reference image for the imaginary flag, from which the x and y values for line endpoints were gathered. Testing of the floodfill was performed by filling the image; though the floodfill implementation has issues preventing it from being used in more general cases, it was sufficient for this elephant, which was all that was necessary.