ECSE 324 - Lab #4: High Level I/O: VGA, PS/2 Keyboard, and Audio

Alex Hale (260672475) and Tre Mansdoerfer (260680162)

November 24, 2017

Section 1 - VGA

VGA.s

VGA_clear_charbuff_ASM

This subroutine clears all the spaces in the character buffer by setting their values to zero. The subroutine iterates through the 80 x 60 grid, prepares the memory address of the target byte using boolean operations and shifts, then stores a full byte of zeros to the address.

When debugging this subroutine, it took a long time to realize that we needed to use a STRB operation in order to access only the desired byte. Before we realized this, we experienced difficulty because we were trying to access the entire word in memory. We also had to take a closer look at the DE1-SOC manual after realizing that we were not properly preparing the memory address as described in the manual - the addressing scheme for the character buffer is different than the addressing scheme for the pixel buffer.

This subroutine could be improved by accessing entire words instead of individual bytes. Since all the subroutine has to do is store zeros, we could access every fourth location in the grid and store a word's worth of zeros, then increment the counter by four.

VGA_clear_pixelbuff_ASM

This subroutine clears all the spaces in the pixel buffer by setting their values to zero. The subroutine iterates through the 320 x 240 grid, prepares the memory address of the target half-word using boolean operations and shifts, then stores a full half-word of zeros to the address.

When debugging this subroutine, it took a long time to realize that we needed to use a STRH instruction in order to access only the desired half-word. Before we realized this, we experienced difficulty because we were trying to access the entire word in memory. We also had to take a closer look at the DE1-SOC manual after realizing that we were not properly preparing the memory address as described in the manual - the addressing scheme for the pixel buffer is different than the addressing scheme for the character buffer. Finally, we could not load the number 319 as an immediate value, so we had to come up with an alternative strategy to move this value into a register.

This subroutine could be improved by accessing entire words instead of half-words. Since all the subroutine has to do is store zeros, we could access every fourth location in the grid and store a word's worth of zeros, then increment the counter by four.

VGA_write_char_ASM

This subroutine writes the integer value of the specified character to the specified address in the character buffer. The subroutine verifies that the specified address is within the boundaries of the buffer, constructs the memory address of the target byte using boolean operations and shifts, then stores the byte to the memory address.

When debugging the subroutine, we realized that we needed to use the STRB instruction to store individual byte instead of the STR instruction, which stores an entire word. We also had to double-check that we were using the appropriate addressing scheme for the character buffer, as defined in the DE1-SOC manual.

This subroutine is relatively straightforward, so there are not many opportunities for improvement available.

VGA_write_byte_ASM

This subroutine writes the hexadecimal representation of the input integer to the specified address in the character buffer. The subroutine verifies that the specified address is within the boundaries of the buffer, separates the input byte into two sections of 4 bytes, converts each 4-bit segment into a hexadecimal digit using the ASCII convention, constructs the two required memory addresses, then stores the two characters to memory.

When debugging the subroutine, we experienced similar issues as the first three subroutines concerning the addressing scheme and the STR instruction. In addition, it took us a few iterations of the code to realize that we had to separate the input string into two sets of four bits. Finally, we had to find an efficient way to convert the four-bit input segments into the appropriate hexadecimal characters, since the digits 0 - 9 and A - F are not next to each other in the ASCII scheme.

This subroutine could be condensed by reusing code. The subroutine uses the same technique for each of the two input segments, and a loop could be use to reuse this code and condense the file. In addition, there is likely a way to determine the two hexadecimal digits without having to separate the input string, which would greatly increase code efficiency.

VGA_draw_point_ASM

This subroutine writes the input half-word to the specified address in memory. The subroutine checks that the supplied location is within the allowed grid, prepares the memory address according to the scheme in the manual, then stores the input half-word to the memory address.

This subroutine presented the same challenges as the *VGA_write_char_ASM* subroutine, with the exception of the different addressing scheme and the storing of a half-word instead of a byte.

This subroutine does not have much opportunity for improvement, other than the combining of some instructions.

main.c

The Part 1 section of the main.c file is very simple, since most of the code was provided. After including our push-button and slider-switch drivers from Lab 3, we used the same approach as before to detect button presses and switch movements. We assigned the appropriate key combinations to the subroutines given in the lab manual. The *test_char()* function displays a screenful of all the ASCII characters, the *test_byte()* function displays a screenful of hexadecimal digits, and the *test_pixel()* function displays a grid of colours.

There were no noticeable challenges, and there are no known available improvements.

Section 2 - Keyboard

ps2_keyboard.s

This subroutine checks the PS/2 data register to see if the input is valid. If valid, it stores the data to the memory address provided as input, then returns 1. If not valid, it returns 0 and does not store anything to the input memory address.

When debugging this subroutine, we noticed that we could not use the PS/2 data register to supply the data for the return. This is because once this memory address is read, the values within it changed. Instead, we had to store the data in the memory address provided as input so it can be accessed by main.c.

No major improvements are available.

main.c

The logic in main.c is substantial for this subroutine. After declaring variables and allowing the user to clear the screen with the pushbuttons, the *read_PS2_data_ASM(char *data)* subroutine is used to check if the input is valid and, if so, move the input data to the address of the data pointer variable. A number of cases are then checked to determine what to output on the screen, after which the counters tracking the position on the grid are updated.

- If the input data is the same as the current data
    - If the previous data is a break code, the user is tapping and releasing one key: output the key's data for each press.
    - Otherwise, a key is being held: use a typematic scheme to determine output
- If the input data is blank: nothing is pressed, so do nothing
- If the input data is the same as the previous data, the input data is the end of the break code
    - Update the list of values, but do not print anything
- Otherwise, the input is a new key press. Display the key's data once

This subroutine presented the most difficulty of the lab, due to two issues. The first problem was determining the structure of the condition block outlined above to output the correct data in each keypress scenario. The second problem was realizing that the assembly subroutine must be fed a pointer to a *different* memory address than the PS/2 data register in order to function properly.

The timing of the typematic procedure is quite slow and variable, and could be improved by using a proper timer instead of counting the number of iterations. The structure of the conditional block could be made much simpler and easier to understand.

Section 3 - Audio

AUDIO.s

This subroutine checks whether the audio FIFO is full. If there is space, it stores the provided argument to the FIFO and returns 1. If there is not space in both sides, it does not store anything and returns 0.

Other than an error in the manual that led us astray momentarily, the writing of this subroutine did not require any significant debugging. There are not any significant improvements available.

main.c

This code sends to the *audio_port_ASM(int data)* subroutine 240 high signals, followed by 240 low signals. Since the sampling frequency is 48K, this ideally creates a 100 Hz square wave output.

This code did not present any noteworthy issues. Due to the time taken to run the code, this code does not produce a wave of exactly 100 Hz. This code could benefit from a proper clock instead of a counter, or at least some calibration to determine an appropriate "fudge factor".