ECSE 324 - Lab 4 Report

ARCHIT GUPTA (260720757)
KAI LUO (260479742)

Section 1 – VGA

VGA.s

Clear_charbuff:

This subroutine clears 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.

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

Clear_pixelbuff:

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.

We had trouble because we were trying to access the entire word in memory. This subroutine could be improved by accessing entire words instead of half-words. Since all the subroutine should 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.

Write_char:

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.

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.

Write_byte:

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.

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.

Draw_point:

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 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 colors.

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

Main C:

The main.c file 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. Several 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 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.

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.

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 could have been made more accurate.