G21_Lab4_Report

This Lab focuses on the high-level I/O capabilities of the DE1-SOC FPGA board, including the application of VGA controller, PS/2 port controller and audio controller on different applications.

Part 1. VGA

VGA clear charbuff ASM

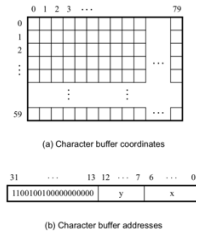

Figure A character buffer coordinates

For char buffer, the x and y counters are set as 80*60 grid. We take two counters and create a nested loop to iterates all grids and store a byte of zero in that grid(representing the address). As shown above, we set x and y to be the last grid and count back. When x reaches zero, y decrease by one and reset x counter. Then for each grid, store zero in it.

VGA clear pixelbuff ASM: For pixel buffer, the x and y counters are set as 320*240. When x counters in both assembly program reached the boundary value, they were reset and both y counters will be incremented by 1. After running the "clear" function, the corresponding memory buffer would be cleared.

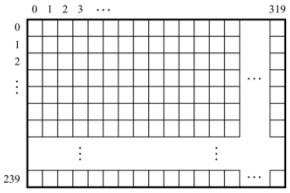

Figure 37. Pixel buffer coordinates.

Figure B pixel buffer coordinates

VGA write char ASM: For VGA_write_char_ASM, the boundaries are first checked (must be within the range 0 <= x <= 79, 0 <= y <= 59), Than we take the y counter and left shift it by 7 which makes space for x counter. The most important and difficult part is to add x and y counter. We use ORR operation to add y and x counter to one and than store byte the ACSII value in the address.

VGA write byte ASM: For VGA_write_byte_ASM, the third input must be transformed into hexadecimal first. Since it is a char (1 byte), equivalent to 8 bits, the first 4 bits are the first hexadecimal character, the last 4 bits are the second hexadecimal character. If the hex value represented by the first 4 bits is within 0-9, the data to be stored will be the input plus #48, if the hex value is ranged as A-F, the input will fist subtract #10 (the input itself) and plus #65 (according the ASCII table), then store to the address calculated by ORRing the input x and y parameter plus the base address. Same for the second four bits. The most challenging part is to convert hex digit to ASCII. We were stuck on the offset. Then we figure out for letter, we need to minus 10 from the offset.

VGA draw point ASM: Lastly, the VGA_draw_point_ASM first checks the inputs x and y are in the range 320*240 (used 319*239 since staring from 0), then store the third input (store halfword used here since the input type is short) to the address calculated by the base address plus offset passed by the first two inputs.

Main C: Three sample tester functions are provided to be implemented in main.c in association with 4 pushbuttons on board. When Pushbutton0 is pressed with any slider switches is on, test_byte() is called the monitor will show hexadecimal values from 00 to FF and loop over the entire screen, if there isn't any slider switches open, test_char() is called and the monitor will be displaying the whole ASCII table (repeated). When Pushbutton1 is pressed, test_pixel() is called and a looped color spectrum will be displayed. When Pushbutton3 is pressed, VGA_clear_charbuff_ASM() is called and colors will be cleared. When Pushbutton4 is pressed, VGA_clear_pixeluff_ASM() is called and all characters (ASCIIs and HEXs) will be cleared. This was implemented by using several if statements in an infinity while loop in the C program.

Part 2. Keyboard

ps2_keyboard.s; In the s file, the subroutine "read PS2 data ASM" takes a char pointer variable data as input, in which the data that is read will be stored and return an integer that denotes whether the data read is valid or not.

The basic function of this subroutine is to check the RVALID bit in the PS/2 Data register. If it is valid, then the data from the same register should be stored at the address in the char pointer argument, and return 1. Otherwise, the subroutine should return 0.

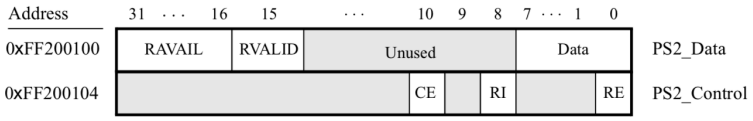

Figure 43. PS/2 port registers.

Figure C PS/2 port register

We take the PS2_Data address and the value of it into a register. And use test operation between the result and 32768(1000000000000000 in binary). If the result turns out to be equal. Than it means Rvalid bit is 0. Otherwise, RVALID is 1. Than we store the data in the char pointer and return 1 to indicate the valibility.

The challenge we face the operation we use to store data in the char pointer. We used to use STR but it doesn't work. Then we realize that data is need to use the store byte operation.

main.c: In this part, an application is created and it should read raw data from the keyboard and display it to the screen if it is valid. We set two counter x, y to keep track of the address and a char c. In the while loop, we use 'read_PS2_data_ASM(char *data)' to check whether the input is valid. If it is valid, use VGA_write_byte_ASM(x, y, c); and increase x counter by 3 to update the position on the grid. we also need to set the boundary for the x, y counter.

The most challenging part in this part is to understand how the PS/2 bus works.

Part 3. Audio

Audio.s: This subroutine take one integer argument and write it to both left and right FIFO only if have space. And return 1 if data was written and 0 otherwise.

First, we load the address of base address of Fifospace address of the audio port register. And store the input into that address. Since we want to check eight bits of WSLC, we use and operation between #0xFF000000 and the input to shift right by 24 to make the eight bits WSLC to be easier to compare. Then compare the result with 0 if WSLC value is 0, end the subroutine and if the value is not zero, return 1. Next, we check the WSRC 8-bit. Similar as before, except this time, we shift the result from and operation by 16, not 24. If the both WSLC and WSRC is not zero, this indicates that data can be written in leftdata and rightdata. And at the end, return 1.

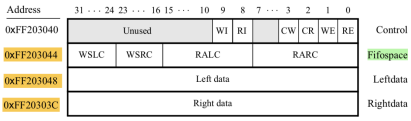

Figure 35. Audio port registers.

Figure D audio port register

Main.c: For this part, we set the sampling rate to be 48000sample/sec and frequency to be 100Hz. This means that for each period there is 480 samples and in every 240 sample, a ‘1’ should be written to the FIFOs and other 240 samples, a’0’ should be written. So we write a while loop and take a counter starting from 0 and increase the counter(write 1 to FIFOs) until it reaches 240(write 0 to FIFOs).