Lab3 report

Wen Cui 260824815

Within this lab, the online DE1-SoC simulator was used to display pixels and characters using VGA controller and accept keyboard input via PS/2 port.

Part1 drawing things with VGA

Brief description: In this part, we are required to implement 4 subroutines in order to display color and characters on the VGA board. The followings include the details of implementation.

- VGA_draw_point_ASM: this subroutine receives 3 input: r0 stores the x coordinate; r1 stores the y coordinate; r2 stores the color to be drawn on the screen. Following callee-save convention, we first push all the registers and link register used later to the stack. Then, base address of the pixel buffer is loaded into r5. After that, we compare the x and y coordinate received as arguments to make sure they fit in the range. This can be achieved by using comparing instructions with r0, 319 and r1, 239 (since the pixel buffer provided an image resolution of 320 * 240 pixels) if the position provided is out of range, the subroutine will be terminated by popping the saved registers and link register and “bx lr” instruction. Else, if the coordinate fits in the range, according to the instruction, the pixel color can be accessed at 0xc8000000 | (y << 10) | (x << 1). "<<” can be achieved by LSL and “|” can be achieved by ORR instruction. And the base address has already been stored in r5. The result of this sequence of operations will be stored in r4. In the end, the pixel value of color passed in r2 can be stored in r4 (address) by STRH instruction which stores half-word at the address since the color value only takes 16 bits. After that, the subroutine will be terminated by popping saved registers and link register and “bx lr” instruction.
- VGA_clear_pixelbuff_ASM: the main idea is similar to the previous subroutine. In order to clear all the memory location in the pixel buffer, we store 0 in all valid locations on the screen. Reaching all memory locations can be achieved by a nested loop. More specifically, I used outside loop to loop over x coordinates (0 – 319) (stored in r0) and inside each outside loop, an inside loop to go over all y coordinates (0 – 239) (stored in r1) can be used to get all positions, r8 is also used to reset y coordinates during each iteration of outer loop. In each iteration, branch VGA_draw_point_ASM is called, inside which r0 stores x coordinate, r1 stores y coordinates, r2 (color value) is moved to 0 in order to clear. Additionally, callee-save convention is achieved by pushing and popping registers upon entering and leaving subroutine.

- VGA_write_char_ASM: this subroutine writes the ASCII code passed in the third argument (r2) to the screen at the (x, y) coordinates given in the first two arguments (r0 and r1). First, following the callee-save convention, we push all the registers and link registers on the stack. Then, the base address of character buffer memory can be stored in r5. The next step is to determine whether the coordinates fit in the range by comparing instructions between r0, 79 and r1, 59 since the resolution of character buffer is 79 * 59. If the coordinates are out of the range, the subroutine will be terminated by popping saved registers and link register and "bx lr" instruction. Else, according to the instruction, the individual character can be accessed at 0xc9000000 | (y << 7) | x. "|" s can be achieved using 2 ORR instructions and LSL instruction on y can perform "<<". After the sequence of operations, the address will be stored in r4. Then, the ASCII value stored in r2 can be displayed by performing STRH instruction to store half word (16-bit character value) at memory address r4. After that, the subroutine will be terminated by popping saved registers and link register and "bx lr" instruction.
- VGA_clear_charbuff_ASM: this subroutine clears all valid memory locations in character buffer. In order to do that, we need to access all valid memory location (x : 0-79) (y: 0-59) using nested loops. Similar to VGA_clear_pixelbuff_ASM, outer loops loop over all x values (stored in r0) and makes use of r8 to reset y coordinate (r1) and inner loop loops over all y values (stored in r1). After obtaining each (x, y) coordinate, branch VGA_write_char_ASM is called, inside which r0 corresponding to x, r1 corresponds to y and r2 (character value) is moved to 0 in order to clear. In this way, all char buffers can be cleared. Then the subroutine can be terminated. Additionally, callee-save convention is achieved by pushing and popping registers upon entering and leaving subroutine.

After implementing 4 subroutines, using the provided template, the hello world graph can be successfully displayed on the VGA display.

Main approach The first main approach taken is the use of LSL and ORR instruction. The key point for the subroutine is memory address calculation. Using the formula provided in the instruction, we need to first left shift x/y coordinates, then 2 orr instructions can be used on shifted x y coordinates, base address to obtain the actual memory location. The other main approach is the use of nested loop to access each valid memory location. Since we need to go over all valid x and y coordinates, both outer loop and inner loop need to be used to access each point. The next main approach is the use of STRH to store half a word in memory location. We us STRH instead of STR since only 16 bits are used to store color/ ASCII character.

Main challenges One challenge I faced was to compare x coordinate with 319. At first, I used mov instruction to load the bounding of VGA display resolution into register. However, since 319 is misaligned, this cannot be achieved. After multiple tries, I identified the bug and used the strategy of mov 300 and then add 19 to load the bounding into register. Additionally, I forgot to reset the y coordinates each time in the outer loop at first.

Although it did not make any difference when printing out “Hello world”, this bug was detected in the later part of this lab. After multiple tries and using the debug mode, this problem was successfully identified and fixed in the end. Other than these, the implementation was relative straightforward by following the instruction, no other major challenge was faced during implementation.

Possible improvement One possible improvement for the program is to combine multiple instructions that perform ORR/ LSL to less instructions or reuse some registers during operation. In this way, less registers can be used to perform calculation, the number of positions on the stack that are used to store callee-save registers can be reduced. Additionally, the program will be simplified, and the readability and understandability of the program can be improved significantly at the same time.

Part 2: Reading keyboard input

Description: This program reads keystrokes from the keyboard and writes the PS/2 codes to the VGA screen using the character buffer. Provided the main assembly file, there are only 2 tasks to be done. First, the VGA driver implemented in the previous part can be reused to display character. Then, a subroutine “read_PS2_data_ASM” is implemented as following.

This subroutine receives r0 (a memory address in which the data that is read from the PS/2 keyboard will be stored) as input. At first, according to the callee-save convention, the registers to be used along with the link register is pushed on the stack. Then we load the content at base address of PS/2 data register to r1 to check the RVALID bit. In order to access the valid bit, first, we copy the content into r2, then, according to the instruction, “RVALID bit can be accessed by shifting the data register 15 bits to the right and extracting the lowest bit”, LSR instruction (lsr r2, #15) is used to perform shift operation and AND instruction (and r2, r2, #1) is used to extract the lowest bit. In this way, by “cmp r2, #1”, the result is greater or equal to 0 indicates that it is valid. If it is valid, the data from the same register should be stored at the address in the pointer argument. This can be achieved by first using “AND r1, #0xff “, where r1 stores the content at the base address of the data register, to clear all bits except data bits since data bits are 8-bit long. Then “STRB R1, [R0]” can be used to store data at the location given as input (r0) since data is 1-byte long. After that, we can move r0 to 1 in order to return 1. Else if valid bit is not set, the subroutine will return 0 by moving r0 to 0. Finally, the subroutine will be terminated by popping saved registers along with link register and the “bx lr” instruction.

After implementing the subroutine above, the desired functionality to display keyboard input can be achieved by simply loading the provided main program.

Main approach: The main approach used in this section is use of AND and LSR instruction to access valid bit. By performing “lsr r2, #15” followed by “and r2, r2, #1”, the result can only be 1 if the valid bit is 1. Through this approach, the valid bit can be successfully accessed. Another use of AND instruction is to load the data

from data register. Since we only need the data bits from data register, which is the least significant 8 bits, by performing bitwise and with 0xff , we can clear all the other bits. The other approach used in this part is the return from subroutine. By convention, r0 is always used for input and return value. Therefore, returning from subroutine can be achieved by using moving instruction on r0.

Main challenge: The main challenge I faced in this part is how to check valid bit by ignoring all other bits and how to store only data bits from data register into desired memory location. At first, I used STR instead of STRB, which gave me wrong output. However, after multiple tries and the use of debug mode on the simulator to check register values, the approach described above was developed. Other than this, the rest of this part can be implemented easily by copying the VGA code from previous part. There was no other main challenge faced in this part.

Further improvement: In this part, one of the possible improvements is to use less register in the subroutines by combining multiple instructions. This can also be achieved by reusing some registers inside subroutine. In this way, less callee-save registers are used, and space required on the stack is reduced at the same time. Additionally, the readability and understandability of this program can be significantly improved.

Part 3: Putting everything together: Vexillology

Brief description: This part creates an application that paints a gallery of flags. The user can use keyboard keys to navigate through flags. Pressing the D key will prompt the application to show the next flag. Similarly, pressing the A key will prompt the application to show the previous flag. Pressing A when at the first flag or D when at the last flag cycles to the last or first flag, respectively. There are 2 tasks in this program

1. Include VGA and PS/2 driver functions.
2. Implement flag painting logic for two other flags by rewriting draw_real_life_flag and draw_imaginary_flag . One flag must be a real-life flag and another flag is a flag designed by myself.

This first task can be achieved easily by simply copying the code from previous part into the given template. Then the second part can be achieved by using the draw_rectangle and draw_star subroutines provided. For the draw_real_life flag, I chose flag of China, which is composed of a red rectangle of size (240 * 320) and 5 yellow stars. To draw the red rectangle, following the instruction, x coordinate of left top corner (0), y coordinate of left top corner (0), width (240), height (320), color code for red (45248) are passed respectively to r0-r4 as input arguments to subroutine “draw_rectangle”. In order to draw 5 yellow stars with different sizes and at different positions, following the instruction, x center, y center, radius (33 for the big star, 12 for small stars) and color encoding (452489 for yellow) are passed to r0-r3 as input arguments respectively to draw_star subroutine. In this way, the flag of China can be displayed on the VGA pixal buffer. For the imaginary flag, I drew 3 parallel rectangles each of size 106* 240 with different colors and each rectangle has a yellow star in the

middle. This can be achieved in a similar way as the China's flag with different size, coordinate parameters and color encoding. After implementing the 2 subroutines, the program can be combined together with the PS/2 keyboard subroutine, so that the user can switch the display of the 3 flags as described above.

Main approach: The main approach used in this lab is the understanding and use of draw_rectangle and draw_star subroutines. By reading through the given example for texans flag and understanding the logic of these 2 subroutines, we can create own flags by storing different parameter values into corresponding registers. More specifically, coordinates are passing by immediate values to registers while color encodings are stored as memory words, like the given example, and are loaded into registers inside draw_flag subroutines. In this way, different rectangle and stars can be created and multiple flags can be drawn by connecting these rectangles and stars.

Main challenge: Since the previous parts have already been implemented and tested, this application can be achieved relative easily by combining those parts. The only challenge I faced was the coloring scheme for different color. However, by searching online and reading the manual carefully, after multiple tries, the desired color can be displayed correctly. After that, the remaining part of determining coordinates and sizes can be implemented easily and no major challenge was encountered.

Further improvement: One possible improvement for this part is to determine color encoding and rectangle/ star coordinates more carefully, so that the flag created may match more perfectly with the real flag. Other than that, the subroutines are provided and defined specifically, so I did not come up with other possible improvements.