ECSE 324: Computer Organization
Lab 3: Keyboard and VGA

Abstract

In this lab, you will explore the more complex input/output capabilities of the DE1-SoC: the PS/2 keyboard port, and VGA display. You will begin by writing a driver for each of these devices, and then put them together into an application.

Summary of Deliverables

- Source code for:
  - A library implementing a complete VGA driver
  - A library implementing a complete PS/2 driver
  - An application that draws maps on the VGA screen in response to PS/2 input
- Demo, no longer than five (5) minutes (week of April 4th)
- Report, no longer than four (4) pages (10 pt font, 1” margins) (due April 12th at 11:59 pm)

Please submit the above in a single .zip archive, using the following file name conventions:

- Code: `part1.s`, `part2.s`, `part3.s`
- Report: `StudentID_FullName_Lab1_report.pdf`

Grading Summary

- 50% Demo
- 50% Report

Changelog

- 21-Mar-2022 Initial revision
- 24-Mar-2022 Fixed a typo in Part 3 specifying what keys should be pressed to cycle through flags.

Overview

In this lab we will use the high level I/O capabilities of the DE1-SoC simulator to

1. display pixels and characters using the VGA controller, and
2. accept keyboard input via the PS/2 port.

For each of these topics, we will create a driver. We will test the drivers both individually and in tandem by means of test applications.

Part 1: Drawing things with VGA

The DE1-SoC computer has a built-in VGA controller that can render pixels, characters or a combination of both. The authoritative resource on these matters is Sections 4.2.1 and 4.2.4 of the DE1-SoC Computer Manual. This section of the lab provides a quick overview that should suffice for the purpose of completing this lab.

To render pixels, the VGA controller continuously reads the pixel buffer, a region in memory starting at `0xc8000000` that contains the color value of every pixel on the screen. Colors are encoded as 16-bit integers that reserve 5 bits for the red channel, 6 bits for the green channel and 5 bits for the blue channel. That is, every 16-bit color is encoded like so:

15 … 11	10 … 5	4 … 0
Red	Green	Blue

The pixel buffer is 320 pixels wide and 240 pixels high. Individual pixel colors can be accessed at `0xc8000000 | (y << 10) | (x << 1)`, where `x` and `y` are valid x and y coordinates.

As previously hinted, we can also render characters. To do so, we will use the character buffer, which is analogous to the pixel buffer, but for characters. The device's VGA controller continuously reads the character buffer and renders its contents as characters in a built-in font. The character buffer itself is a buffer of byte-sized ASCII characters at `0xc9000000`. The buffer has a width of 80 characters and a height of 60 characters. An individual character can be accessed at `0xc9000000 | (y << 7) | x`.

Task: Create a VGA driver

To provide a slightly higher-level layer over the primitive functionality offered by the pixel and character buffers, we will create a driver. That is, a set of functions that can be used to control the screen.

To help get you started, we created an application that uses such functions to draw a testing screen. Your job is to create a set of driver functions to support the application. Download vga.s and augment it with the following four functions:

- `VGA_draw_point_ASM` draws a point on the screen with the color as indicated in the third argument, by accessing only the pixel buffer memory. Hint: This subroutine should only access the pixel buffer memory.

- `VGA_clear_pixelbuff_ASM` clears (sets to 0) all the valid memory locations in the pixel buffer. It takes no arguments and returns nothing. Hint: You can implement this function by calling `VGA_draw_point_ASM` with a color value of zero for every valid location on the screen.

- `VGA_write_char_ASM` 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). Essentially, the subroutine will store the value of the third argument at the address calculated with the first two arguments. The subroutine should check that the coordinates supplied are valid, i.e., x in [0, 79] and y in [0, 59]. Hint: This subroutine should only access the character buffer memory.

- `VGA_clear_charbuff_ASM` clears (sets to 0) all the valid memory locations in the character buffer. It takes no arguments and returns nothing. Hint: You can implement this function by calling `VGA_write_char_ASM` with a character value of zero for every valid location on the screen.

Their C prototypes are as follows:

```
void VGA_draw_point_ASM(int x, int y, short c);
void VGA_clear_pixelbuff_ASM();
void VGA_write_char_ASM(int x, int y, char c);
void VGA_clear_charbuff_ASM();
```

Notes:

- Use suffixes `B` and `H` with the assembly memory access instructions in order to read/modify bytes/half-words in memory.

- You must follow the conventions taught in class. If you do not, then the testing code in the next section will be unlikely to work.

Testing the VGA driver

To test your VGA driver, run your finished assembly file. You can inspect the VGA output visually using the VGA pixel buffer tab under the Devices panel of the simulator.

If you implemented your driver correctly, compiling and running the program will draw the following image.

Part 2. Reading keyboard input

For the purpose of this lab, here's a high level description of the PS/2 keyboard protocol. For a more comprehensive resource, see Section 4.5 (pp. 45-46) of the DE1-SoC Computer Manual.

The PS/2 bus provides data about keystroke events by sending hexadecimal numbers called scan codes, which for this lab will vary from 1-3 bytes in length. When a key on the PS/2 keyboard is pressed, a unique scan code called the make code is sent, and when the key is released, another scan code called the break code is sent. The scan code set used in this lab is summarized by the table below. (Originally taken from Baruch Zoltan Francisc's page on PS/2 scan codes.)

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1C	F0,1C	9	46	F0,46	[	54	FO,54
B	32	F0,32	\`	0E	F0,0E	INSERT	E0,70	E0,F0,70
C	21	F0,21	-	4E	F0,4E	HOME	E0,6C	E0,F0,6C
D	23	F0,23	=	55	FO,55	PG UP	E0,7D	E0,F0,7D
E	24	F0,24	\	5D	F0,5D	DELETE	E0,71	E0,F0,71
F	2B	F0,2B	BKSP	66	F0,66	END	E0,69	E0,F0,69
G	34	F0,34	SPACE	29	F0,29	PG DN	E0,7A	E0,F0,7A
H	33	F0,33	TAB	0D	F0,0D	U ARROW	E0,75	E0,F0,75
I	43	F0,43	CAPS	58	F0,58	L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B	L SHFT	12	FO,12	D ARROW	E0,72	E0,F0,72
K	42	F0,42	L CTRL	14	FO,14	R ARROW	E0,74	E0,F0,74
L	4B	F0,4B	L GUI	E0,1F	E0,F0,1F	NUM	77	F0,77
M	3A	F0,3A	L ALT	11	F0,11	KP /	E0,4A	E0,F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7C	F0,7C

O	44	F0,44	R CTRL	E0,14	E0,F0,14	KP -	7B	F0,7B
P	4D	F0,4D	R GUI	E0,27	E0,F0,27	KP +	79	F0,79
Q	15	F0,15	R ALT	E0,11	E0,F0,11	KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D	APPS	E0,2F	E0,F0,2F	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	76	F0,76	KP 1	69	F0,69
U	3C	F0,3C	F1	05	F0,05	KP 2	72	F0,72
V	2A	F0,2A	F2	06	F0,06	KP 3	7A	F0,7A
W	1D	F0,1D	F3	04	F0,04	KP 4	6B	F0,6B
X	22	F0,22	F4	0C	F0,0C	KP 5	73	F0,73
Y	35	F0,35	F5	03	F0,03	KP 6	74	F0,74
Z	1A	F0,1A	F6	0B	F0,0B	KP 7	6C	F0,6C
0	45	F0,45	F7	83	F0,83	KP 8	75	F0,75
1	16	F0,16	F8	0A	F0,0A	KP 9	7D	F0,7D
2	1E	F0,1E	F9	01	F0,01	]	5B	F0,5B
3	26	F0,26	F10	09	F0,09	;	4C	F0,4C
4	25	F0,25	F11	78	F0,78	'	52	F0,52
5	2E	F0,2E	F12	07	F0,07	,	41	F0,41
6	36	F0,36	PRNT SCRN	E0,12, E0,7C	E0,F0, 7C,E0, F0,12	.	49	F0,49
7	3D	F0,3D	SCROLL	7E	F0,7E	/	4A	F0,4A
8	3E	F0,3E	PAUSE	E1,14,77, E1,F0,14, F0,77				

Two other parameters involved are the **typematic delay** and the **typematic rate**. When a key is pressed, the corresponding make code is sent, and if the key is held down, the same make code is repeatedly sent at a constant rate after an initial delay. The initial delay ensures that briefly pressing a key will not register as more than one keystroke. The make code will stop being sent only if the key is released or another key is pressed. The initial delay between the first and second make code is called the typematic delay, and the rate at which the make code is sent

after this is called the typematic rate. The typematic delay can range from 0.25 seconds to 1.00 second and the typematic rate can range from 2.0 cps (characters per second) to 30.0 cps, with default values of 500 ms and 10.9 cps respectively.

Task: Create a PS/2 driver

The DE1-SoC receives keyboard input from a memory-mapped PS/2 data register at address `0xff200100`. Said register has an RVALID bit that states whether or not the current contents of the register represent a new value from the keyboard. The `RVALID` bit can be accessed by shifting the data register 15 bits to the right and extracting the lowest bit, i.e., `RVALID = ((*(volatile int *)0xff200100) >> 15) & 0x1`. When `RVALID` is true, the low eight bits of the PS/2 data register correspond to a byte of keyboard data.

The hardware knows when you read a value from the memory-mapped PS/2 data register and will automatically present the next code when you read the data register again.

For more details, see Section 4.5 (pp. 45-46) of the DE1-SoC Computer Manual.

Download ps2.s. This assembly file implements a program that reads keystrokes from the keyboard and writes the PS/2 codes to the VGA screen using the character buffer. Copy your VGA driver into ps2.s. Then implement a function that adheres to the following specifications:

- **Name**: `read_PS2_data_ASM`
- **Input argument (r0)**: A memory address in which the data that is read from the PS/2 keyboard will be stored (pointer argument).
- **Output argument (r0)**: Integer that denotes whether the data read is valid or not.
- **Description**: The subroutine will 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 pointer argument, and the subroutine should return 1 to denote valid data. If the `RVALID` bit is not set, then the subroutine should simply return 0.

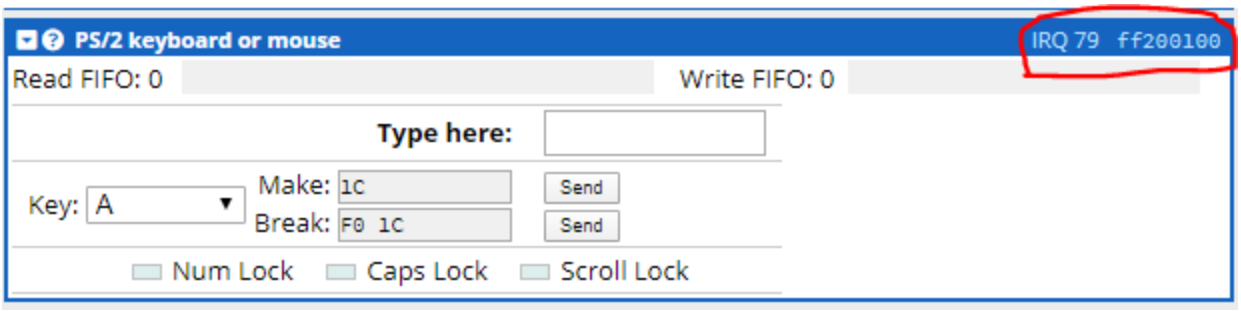
`read_PS2_data_ASM`'s C declaration is as follows:

```
int read_PS2_data_ASM(char *data);
```

Testing the PS/2 driver

To verify that the PS/2 driver is working correctly, you can type into the simulator's PS/2 keyboard device and verify that the bytes showing up on the screen correspond to the codes you might expect from the table in this section's introduction.

If you implemented your PS/2 and VGA drivers correctly, then the program will print make and break codes whenever you type in the simulator's keyboard input device. Make sure to use the keyboard device that says `ff200100`.



Note: If you did not manage to implement a working VGA driver, then you can still get credit for the PS/2 driver by replacing `write_byte` with the implementation below. It will write PS/2 codes to memory address `0xfff0`. Delete all calls to VGA driver functions and delete the `write_hex_digit` function to ensure that your code still compiles.

```
write_byte:
        push    {r3, r4, lr}
        ldr     r4, =0xfff0
        and     r3, r3, #0xff
        str     r3, [r4]
        pop     {r3, r4, pc}
```

Part 3. Putting everything together: Vexillology

We will now create 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.

Download flags.s. This file implements an input loop that reads keystrokes from the keyboard, tests if they are **A** or **D** key presses, and cycles flags accordingly. It also implements a function that draws the flag of Texas.

Your task is twofold:

1. Include your 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 can be a flag you designed yourself.

Hint: The starter code includes the `draw_rectangle` and `draw_star` functions. You can use these functions to draw any flag that consists of rectangles and stars. This encompasses many flags, from the flags of the US to France and even China.

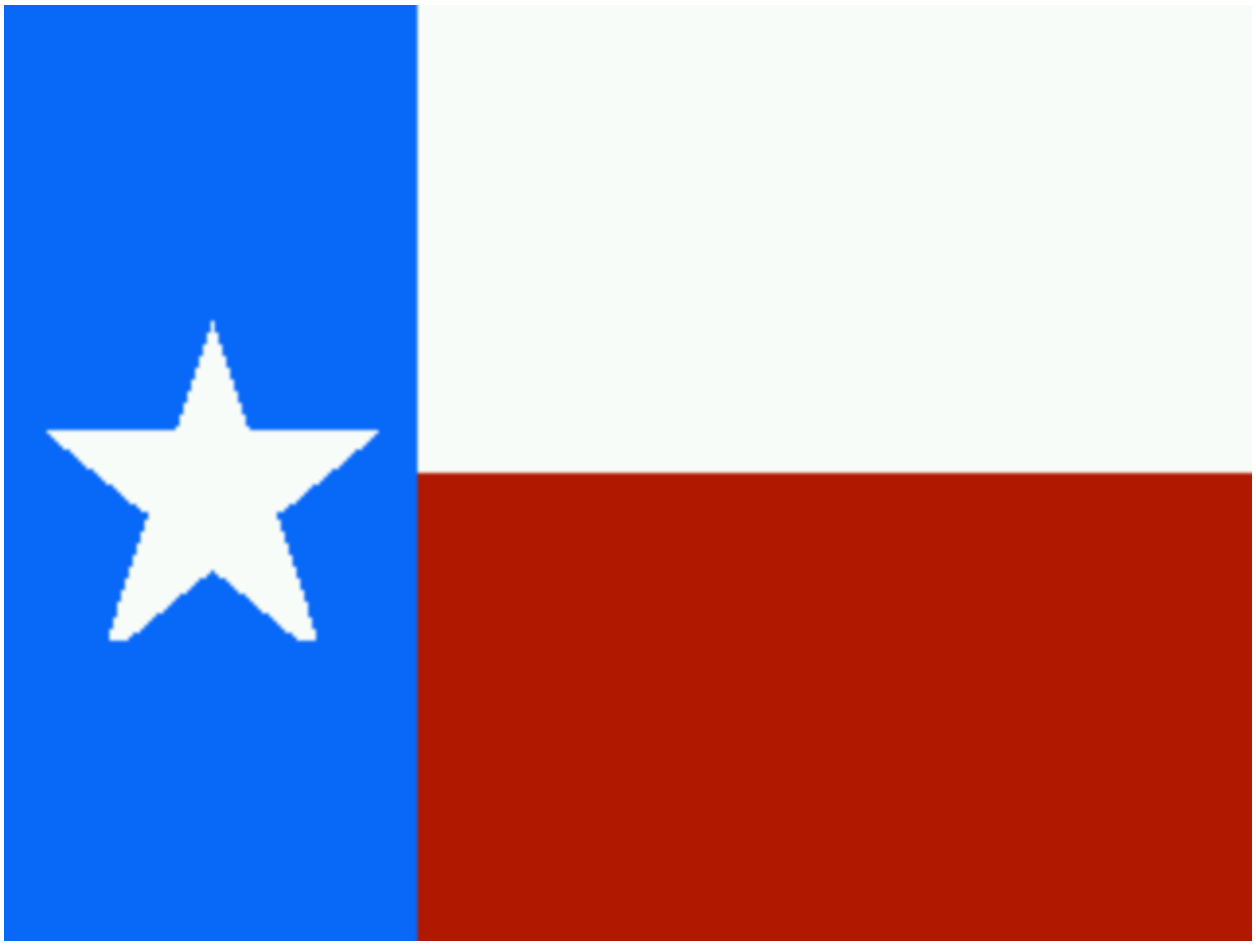
`draw_rectangle` draws a rectangle. It takes five arguments. The first four arguments are stored in registers r0 through r3. The fifth argument is stored on the stack at address `[sp]`. `draw_rectangle`'s signature is:

```
/**
 * Draws a rectangle.
 * @param x The x coordinate of the top left corner of the rectangle.
 * @param y The y coordinate of the top left corner of the rectangle.
 * @param width The width of the rectangle.
 * @param height The height of the rectangle.
 * @param c The color with which to fill the rectangle.
 */
void draw_rectangle(int x, int y, int width, int height, int c);
```

draw_star draws a star. It takes four arguments, stored in registers r0 through r3. Its signature is:

```
/**
 * Draws a star.
 * @param x_center The x coordinate at which the star is centered.
 * @param y_center The y coordinate at which the star is centered.
 * @param radius The star's radius.
 * @param c The star's color.
 */
void draw_star(int x_center, int y_center, int radius, int c);
```

Hint: The flag of Texas as drawn by the starter code should look like the image below. If it doesn't, then your VGA driver might be faulty.



Deliverables

Your demo is limited to 5 minutes. It is useful to highlight that your software computes correct partial and final answers; draw our attention to the registers and memory contents at appropriate points to demonstrate that your software operates as expected.

Your demo will be graded by assessing, for each software deliverable, the correctness of the observed behavior, and the correctness of your description of that behavior.

In your report, for each software deliverable, describe:

- Describe your approach (e.g., how you used subroutines, the stack, etc)
- Challenges you faced, if any, and your solutions
- Shortcomings, possible improvements, etc

Note that you need not describe the operation of the given libraries, only how you made use of the provided functions.

Your report is limited to four pages, total (no smaller than 10 pt font, no narrower than 1" margins). It will be graded by assessing, for each software deliverable, your report's clarity, organization, and technical content.

Grading

Your demo and report are equally weighted. The breakdown for the demo and report are as follows:

Demo

- 30% Part 1: VGA driver
- 30% Part 2: PS/2 driver
- 40% Part 3: Vexillology application

Each section will be graded for (a) clarity, (b) technical content, and (c) correct execution:

- 1pt *clarity*: the demo is clear and easy to follow
- 1pt *technical content*: correct terms are used to describe your software
- 3pt *correctness*: given an input, the correct output is clearly demonstrated

Report

- 30% Part 1: VGA driver
- 30% Part 2: PS/2 driver
- 40% Part 3: Vexillology application

Each section will be graded for: (a) clarity, (b) organization, and (c) technical content:

- 1pt *clarity*: grammar, syntax, word choice
- 1pt *organization*: clear narrative flow from problem description, approach, testing, challenges, etc.
- 3pt *technical content*: appropriate use of terms, description of proposed approach, description of testing and results, etc.

Submission

Please submit, on MyCourses, your source code and report in a single .zip archive, using the following file name conventions:

- Code: `part1.s`, `part2.s`, `part3.s`
- Report: `StudentID_FullName_Lab1_report.pdf`