Homework 3 Report

李京叡, 0616329

***Abstract*—This electronic document is the homework 3 report for the course—Microprocessor Systems: Principles and Implementation at NCTU in 2020 fall.**

I. INTRODUCTION

An I/O controller is a circuit that helps the processor interact with peripheral devices. I/O controllers have functionality specific to handling the demands of talking to and controlling peripheral devices. The controller has the ability to drive the devices with the correct voltage level and with the correct interface standard. I/O controllers may be as simple as a circuit processing logic level inputs or outputs as in the case of driving the status of a LED or detecting a button is pressed. In many cases, the I/O controller maps peripheral devices onto system address ranges and handles data transfer via direct memory access to the processors system memory. They may also be required to measure the time between logic events or perform signal condition. In this assignment, the peripheral device is 1602 LCD device. The I/O controller help Aquila SoC to talk to the LCD device. There are two ways to design the controller, using GPIO device and text-screen buffer device.

A. GPIO device

A GPIO is a general digital signal pin on an integrated circuit which may be used as an input or output, or both, and can be controlled by the users at runtime. A GPIO can be controlled by software. For the hardware part, it can be implemented by using a register as GPIO register. For example, there are 3 control ports and 4 data ports for LCD device, which are RS, RW, E, DB4, DB5, DB6, and DB7. We can wire each bit of the GPIO register to each of 7 control ports. In this way, the processor is able to control the LCD device by assigning value to the GPIO register. The software transfers the request into pin assignment. The processor decode the software code to pull up or down the voltage of a pin and have the processor assign the proper value to the corresponded registers.

B. Text screen buffer device

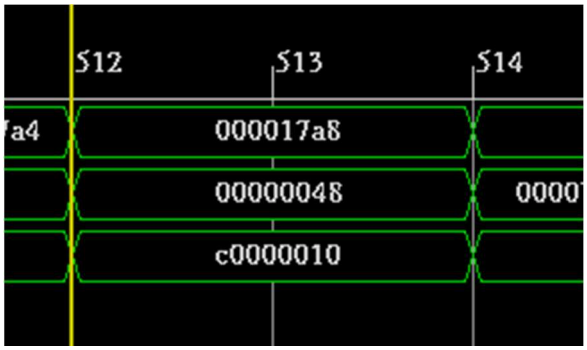
For this method, we maintain a text buffer that stores the text we want to show on the LCD screen. The text buffer can be accessed by the processor to update the content. And the hardware controller scans the buffer continuously to update the LCD screen. The things software should do is to transfer the text into ASCII code and indicate which target address should the code be stored. This method uses memory mapped scheme. It means I/O and memory share the same address space. The advantage of the scheme is that we can see accessing I/O as accessing memory. And the disadvantage is that the addresses that mapped to the I/O devices cannot be used as the real memory. This is the approach I use in this assignment, and the implementation details are shown in the following.

II. SOFTWARE

For the software part, I create a new function, **print_to_lcd**, with a string as the input. And I designate the memory address 0xC0000010 as the device address. There is a while loop in the function, and it scans the input string character by character until it meets the null character, or 32nd character. The function stores the corresponded ASCII code of each character into the address. Since LCD screen can only display 16 characters on each of the two rows, 32 characters in a time, the input string can only be displayed properly if its length is less than or equal to 32 characters. The function keeps a counter to count the number of characters that had been stored into the target address, and those characters over 32 will be overlooked by the function. As a result, the LCD screen can only display the first 32 characters of the strings no matter how long the string is.

III. HARDWARE

In the beginning, my thought of implementing the hardware part was to read the target address, 0xC0000010, from the memory. In my opinion, to have this task done, I had to revise the memory module, adding a new address request signal and a corresponded output data signal. This is quite complicated and also not efficient, since memory is a relatively slow device. As a result, I come up with a simpler idea, when I observing the signals going into and out from the memory module. I use the ILA tool to see the signal of data signal, p_d_core2mem and the address signal p_d_addr and found that each memory request will trigger both signals at the same clock time. To be more specific, take my print_to_lcd function as an example. If I send a string “Hello World” into the function, first, the function stores the ASCII code of the first character, H, which is 0x48 into the target address 0xC0000010. In some time, the p_d_core2mem and p_d_addr would be set as 0x00000048 and 0xC0000010 respectively. The signal is shown below:



, where the top row represents program counter, the second row represents data going into memory, the bottom row represents the address signal sent into memory.Viewing this phenomenon, I can easily write some simple logic circuit to capture the p_d_core2mem when the p_d_addr is 0xC0000010 and update the LCD screen text buffer. This approach is more straightforward and simpler than the one I though in the beginning since it doesn't need to change anything in the memory module. In this way, I catch the data in the time when it is going to be written into the memory. What's more, because it doesn't need to make a request to the memory, it is more efficient. All the changes can be done in the aquila_top and soc_top.

For the LCD module control, there are two registers store the string that should be displayed on the screen. I look it as a row of string being split into 2 rows. I keeps a counter to count the character in the hardware code. The string will be displayed starting from the upper row, and if the string is longer than 16 characters, those characters over 16 will be displayed on the bottom row.

IV. PROBLEM

A problem I met when I was trying to test the hardware code by software code was that the compiler would optimize my software code and eliminate some lines, which made the code logic different from my design. For example, I wrote the following code:

```
int print_to_lcd(char *string) {

   *(int *)0xc0000010 = 65;
   *(int *)0xc0000010 = 66;
   *(int *)0xc0000010 = 67;
   *(int *)0xc0000010 = 68;
   *(int *)0xc0000010 = 69;

        return 0;
}
```

, where I intended to store ASCII code of "A" to "E" in order into the address 0xC0000010. I expected that the LCD screen should display "ABCDE", however, there was only a character "E" on the LCD screen. I turned to check the assembly code and found that it was optimized by the compiler:

```
00001784 <print_to_lcd>:
   1784:    c00007b7        lui     a5,0xc0000
   1788:    04500713        li      a4,69
   178c:    00e7a823        sw      a4,16(a5)
   1790:    00008067        ret
```

The assembly code indicated that the processor only store "E" to the target address. However, my original thought was to store character "A" to "E" one by one, but the compiler seen it as consecutively assign value to the target address and storing character "A" to "D" to the target addresses are redundant operations. As a result, the final value stored in the address should be the final value being assigned. Even I set the compiler optimization parameter to -O0 to force the compiler not to optimize my code, the result was the same. To solve the problem, I found the solution is to using the keyword, volatile. This kind of problem was rare to me before, since I program software only. I hardly concerned how the software interact with the hardware. However, in this situation, I have to take both software and hardware into consideration, and the compiler plays an important role in connecting the hardware and software. The keyword volatile prevents the compiler from applying any optimization on objects that can change in ways that cannot be determined by the compiler. In C, when the system need to read the values of the objects declared as volatile, it always reads the current value the object from the memory location rather than keeping its value in temporary register at the point it is requested. Anyway, the property of a volatile is that it would be omitted from optimization of compiler. As a result, none of the lines of assignment would be eliminated and the code can be compiled as I design.