Assignment 2 Report

EE615 : Embedded Systems Lab

Atharv Gade (210020004)
Yash Meshram (210020053)
29 August 2024

Aim:

a. Part 0

1. Check whether the delay from Lab 1 is equal to the expected value based on the clock speed and no of cycles per instruction.

2. Observe the change in the assembly when you change the declaration of the loop variable to be `register int x`.

3. To find which types of memory are the registers SP and PC addressing

b. Part 1

1. Use the switch connected to the GPIO pins to toggle the LED **ON** whenever the switch is pressed and **OFF** whenever the switch is not pressed.

c. Part 2

1. Use the switch connected to the GPIO pins to toggle the LED **ON** whenever the switch is pressed and **OFF** whenever the switch is not pressed.

Theory:

Part 0:

- To check the delay from Lab 1 is equal to the expected value for 1 second, we calculate the number of iterations of the delay loop that will cause a delay of 1 second. The number of iterations can be calculated by dividing the total number of clock cycles in a second by the number of clock cycles it takes per iteration.

We can calculate the number of iterations by the following formula:

Iterations = (Total # of clock cycles)/(# of clock cycles per iteration*2).

We divide the value by 2 as we are only accounting for the ON or OFF time here. If we were to calculate the delay for (T_on + T_off), there won't be a need to divide by 2.

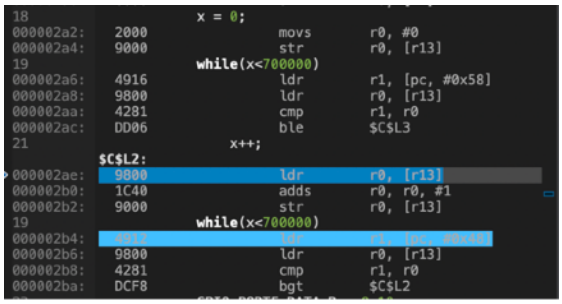
As we know that the total number of clock cycles in 1 machine cycle of the Tiva processor is equal to 16 million, and by observing from the disassembler, that the number of cycles that each iteration of our loop takes

is 11(3* 2 (loads) + 5(rest take single cycle)) , we can calculate the expected value from the above formula to be:

Iterations = (16,000,000)/(11*2) = 727,272(comparable to our expected value of 700,000 from Lab 1).

- The change in assembly which happens by defining the variable as a register is because now the processor doesn't have to fetch the value from the memory to the register and rewrite the updated value back to memory for each increment in the value of the variable. Now, the processor has to only fetch the value once for the first iteration and only has to keep updating the assigned register(R0 in this case) for each iteration. The drawback now however is that the processor now has one register less to work with. This isn't much of an issue for this program as it is lightweight in nature.

with register defn.



without register defn.

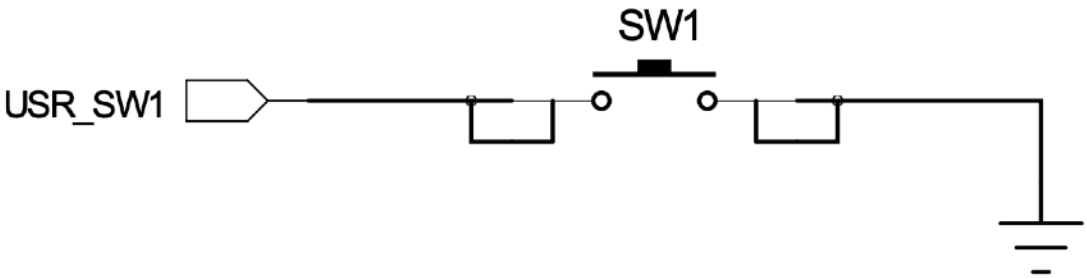
- The Program Counter(PC) addresses the On-Chip Flash memory while the Stack Pointer(SP) addresses the Bit-Banded On-Chip SRAM. We can see this by their values and comparing it with the memory map provided in the datasheet.

Table 2-4. Memory Map

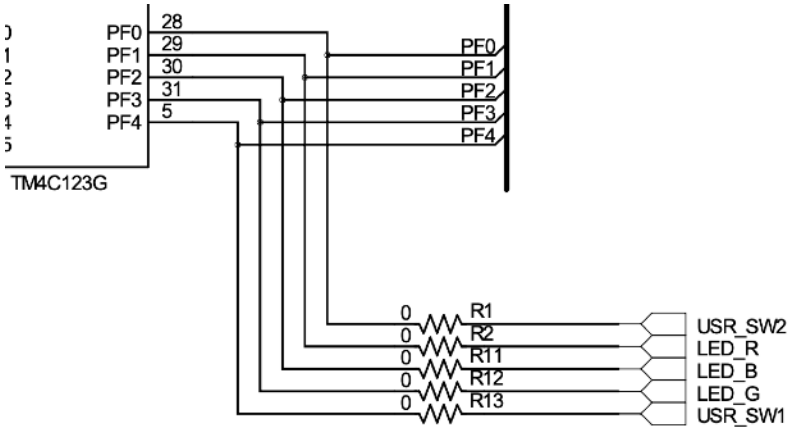
Start	End	Description
Memory		
0x0000.0000	0x0003.FFFF	On-chip Flash
0x0004.0000	0x1FFF.FFFF	Reserved
0x2000.0000	0x2000.7FFF	Bit-banded on-chip SRAM
0x2000.8000	0x21FF.FFFF	Reserved

Part 1:

From the board schematic diagram we can notice that **SW1** has an active low logic. That is, the switch is **ON** whenever the logic is 0(pressed) and **OFF** whenever it's not(logic 1, unpressed).



We can also observe that for **PORTF data register**, the bits are arranged in the following order - [SW1, LED_G, LED_B, LED_R, SW2] - where SW2 is the LSB.



To execute the experiment, we use **SW1** to toggle the LED.

As **SW1** is an active low switch, whenever it'll remain unpressed, it will have a bit value of 1 and we can exploit this to keep checking the switch status.

Whenever the value of the **GPIO_PORTF_DATA_R** register will be less than 0x10 (i.e. 0001 0000 in binary), we can definitely say that it (**SW1**) is pressed as it is the lowest value for which the switch will be unpressed.

We use this condition to load the **GPIO_PORTF_DATA_R** register with the value 0x02 (LED_R) to turn on the red LED whenever **SW1** is pressed.

Part 2:

Similar to Part 1, we check the value of the **GPIO_PORTF_DATA_R** register to be less than 0x10. However, instead of loading it with the value 0x02 whenever the switch was pressed, we now use a counter to toggle between the different colored LEDs.

Initially, the value of the counter variable 'i' is set to zero. When **SW1** is first pressed, and since the value of the counter variable 'i' is 0, we load it with the value 0x02(LED_R). The LED_R will remain ON until **SW1** is unpressed. When it is indeed unpressed, while exiting the loop, we increase the value of the counter variable 'i'.

The same logic is followed to toggle to Blue(LED_B : 0x04 && i==1) and Green(LED_G : 0x08 && i==2) LEDs.

While exiting the loop for which the Green LED is kept ON while pressed, we reset the value of the counter register 'i' back to 0 so that the next LED which will glow after Green will be the Red LED.

Code:

Part 0:

```
int main(void)
{
    SYSCTL_RCGC2_R |= 0x00000020;       /* enable clock to GPIOF */
    GPIO_PORTF_LOCK_R = 0x4C4F434B;     /* unlock commit register */
    GPIO_PORTF_CR_R = 0x1F;             /* make PORTF0 configurable */
    GPIO_PORTF_DEN_R = 0x1E;            /* set PORTF pins 4 pin */
    GPIO_PORTF_DIR_R = 0x0E;            /* set PORTF4 pin as input user
switch pin */
    GPIO_PORTF_PUR_R = 0x10;            /* PORTF4 is pulled up */

    long int x = 0;
        while(1)
        {
            GPIO_PORTF_DATA_R = 0x12;
            x = 0;
            while(x<700000)
            {
                x++;
            }
            GPIO_PORTF_DATA_R = 0x10;
```

```
            x = 0;
            while(x<700000)
            {
                x++;
            }
        }

}
```

Part 1:

```
#include <stdint.h>
#include <stdbool.h>
#include "tm4c123gh6pm.h"

int main(void)
{
    SYSCTL_RCGC2_R |= 0x00000020;       /* enable clock to GPIOF */
    GPIO_PORTF_LOCK_R = 0x4C4F434B;     /* unlock commit register */
    GPIO_PORTF_CR_R = 0x1F;             /* make PORTF0 configurable */
    GPIO_PORTF_DEN_R = 0x1E;            /* set PORTF pins 4 pin */
    GPIO_PORTF_DIR_R = 0x0E;            /* set PORTF4 pin as input user
switch pin */
    GPIO_PORTF_PUR_R = 0x10;            /* PORTF4 is pulled up */


      while(1)
      {
          GPIO_PORTF_DATA_R = 0x10;
          if(GPIO_PORTF_DATA_R < 0x10)
          {
              GPIO_PORTF_DATA_R = 0x02;
          }
      }
}
```

Part 2 :

```
#include <stdint.h>
#include <stdbool.h>
#include "tm4c123gh6pm.h"

int main(void)
{
    SYSCTL_RCGC2_R |= 0x00000020;       /* enable clock to GPIOF */
    GPIO_PORTF_LOCK_R = 0x4C4F434B;     /* unlock commit register */
    GPIO_PORTF_CR_R = 0x1F;             /* make PORTF0 configurable */
    GPIO_PORTF_DEN_R = 0x1E;            /* set PORTF pins 4 pin */
    GPIO_PORTF_DIR_R = 0x0E;            /* set PORTF4 pin as input user
switch pin */
    GPIO_PORTF_PUR_R = 0x10;            /* PORTF4 is pulled up */


    register int i = 0;
      while(1)
      {

          GPIO_PORTF_DATA_R = 0x10;
                    if(GPIO_PORTF_DATA_R < 0x10)
                    {
                            if(GPIO_PORTF_DATA_R == 0x00 && i==0)
                            {
                                while(GPIO_PORTF_DATA_R < 0x10 && i==0)
                                {
                                    GPIO_PORTF_DATA_R = 0x02;
```

```
                                    }
                                        i = 1;
                                }
                                else if(GPIO_PORTF_DATA_R == 0x00 && i==1)
                                {
                                    while(GPIO_PORTF_DATA_R < 0x10 && i==1)
                                    {
                                        GPIO_PORTF_DATA_R = 0x04;
                                    }
                                        i = 2;
                                }
                                else if(GPIO_PORTF_DATA_R == 0x00 && i==2)
                                {
                                    while(GPIO_PORTF_DATA_R < 0x10 && i==2)
                                    {
                                        GPIO_PORTF_DATA_R = 0x08;
                                    }
                                        i = 0;
                                }
                        }
        }
}
```

Result:

The project helped us calculate the delay by observing the number of cycles taken by each iteration of the delay loop. The project also helped understand the working of the disassembler to observe that when we change the definition of the variable x to be a register, we don't have to constantly load the value from the memory to the register and re-write the value back to memory. Now, the value will be stored in one of the registers which the compiler allocates(in this case, R0); so the cycles taken will be less as compared to when it wasn't defined as a register int before.

Also, we were able to find out about how the values in the registers change by simulating the program in discrete steps. We also were able to verify the memory map of the SP and PC registers by comparing it with the datasheet.

In parts 1 and 2, the use of the GPIO ports to take digital user inputs was understood. In particular, we understood how to manipulate the **GPIO_PORTF_DATA_R** register to toggle the LEDs ON or OFF.

Conclusion:

We were successful in demonstrating that the expected value for the delay from the first lab project was approximately equal to the one calculated using the number of cycles per iteration of the loop.

The changes in the assembly were also observed when the type declaration of the counter variable ‘x’ was changed to be a register. Now, the additional cycles taken to load and store the updated value of x were saved. As a result, the delay value for which we previously increased ‘x’ to, also decreased.

For parts 1 and 2, the **GPIO_PORTF_DATA_R** register was manipulated such that whenever the value of the port was less than 0x10(0001 0000), we could say with assurance that **SW1** was pressed; hence invoking the LEDs to turn **ON**.