Lab report 1

Combinational Systems Lab

Programming a Combinational System

Our ARM STM32F746G-DISCOVERY board has 11 GPIO ports from A to K. We have a pinout board with leds and switches that uses the GPIO ports A, F, C, G, I. The component names are shown in the following tables:

Switch Names	**SW7**	**SW6**	**SW5**	**SW4**	**SW3**	**SW2**	**SW1**	**SW0**
Switch Pin ID	`PF10`	`PF9`	`PF8`	`PF7`	`PF6`	`PA15`	`PA8`	`PA0`

LED Names	**LED7**	**LED6**	**LED5**	**LED4**	**LED3**	**LED2**	**LED1**	**LED0**
LED Pin ID	`PG7`	`PG6`	`PC7`	`PC6`	`PI3`	`PI2`	`PI1`	`PI0`

We try a simple test program. First, we make a few equations detailing the logical function we want to have.

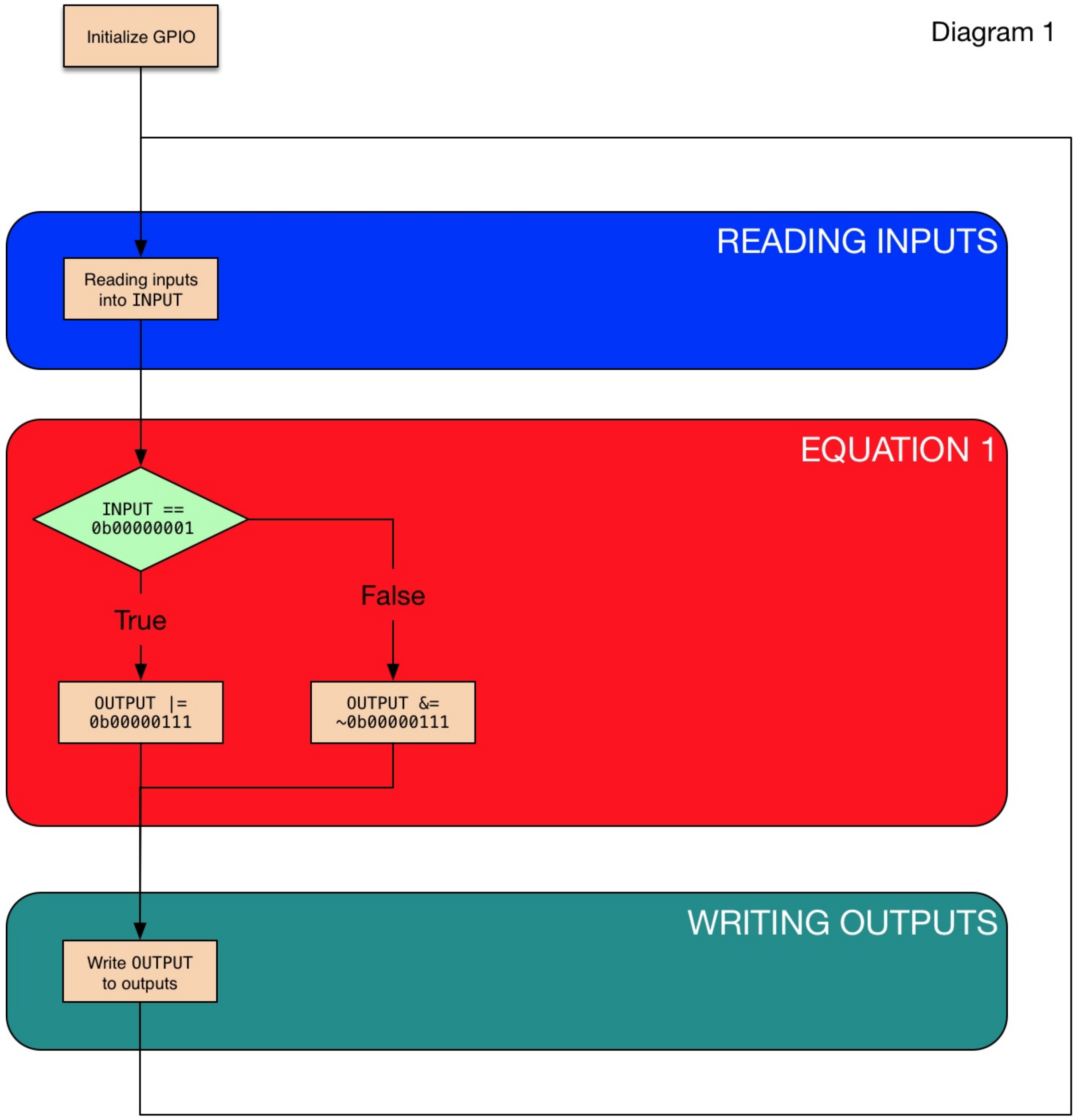
$$LED0 = SW0$$

$$LED1 = SW0$$

$$LED2 = SW0$$

Then we can make a flowchart of our system. The picture will show how our code will interact with the GPIO registers. The Diagram 1 below gives a design where we can flip a single switch to turn on and off 3 leds.

Diagram 1



From the flowchart, we can start coding. We need to now initialize our GPIO ports and configure them to the functions we need.

We enable the GPIO ports with `RCC->AHB1ER`, setting bits to 1 in positions that corrosponds to our GPIO ports.

```
// initialization
// activate peripheral
RCC->AHB1ENR |= 1 << 0 | 1 << 8; // ports A and I
```

Next we set the mode of the pins using `GPIOx->MODER` pointer. Mode 0 is input, 1 is output 2 is alternative function, and 3 is analog mode.

```
GPIOI->MODER |= 1 << 2 * 2 | 1 << 1 * 2 | 1 << 0 * 2; // output mode for PI2, PI1, PI0
```

We only need to set output mode because our input pin is already on input mode by default. Then, our main loop will read the input, process the equation, and then write to the outputs. Reading the input is easy, we use the *and* operator to select our SW0 pin from `GPIOA->IDR` input data register with a mask.

```
INPUT = GPIOA->IDR & 0b00000001;
```

After that, we check if it is the same bits that we are expecting, then we set our output bits using the *or* operation to add to our `OUTPUT` variable.

```
        if (INPUT == 0b00000001)
            OUTPUT |= 0b00000111;
```

If it is not the condition we are expecting, we will clear those bits by *and*ing them.

```
        else
            OUTPUT &= ~0b00000111;
```

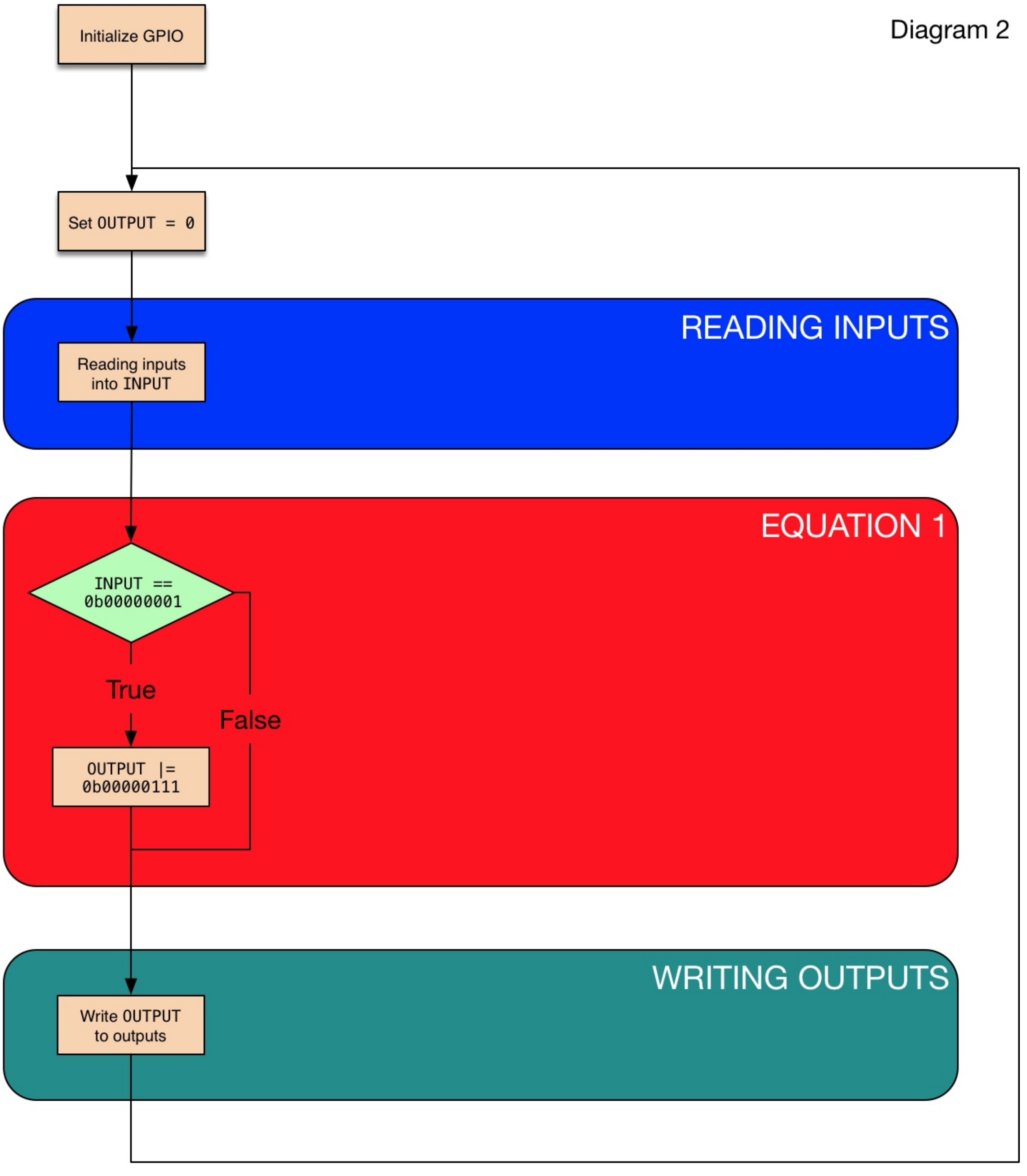
Finally we write to the output at `GPIOI->ODR`, the theoutput data register. Note, if our input didn't change from the last time the loop iterated, we have to leave the bits as is without changing them. To do that, we check if the OUTPUT variable contains our values, if it does, we set those bits into the ODR. If not, we clear those bits.

```
        if ((OUTPUT & 0b00000111) == 0b00000111)
            GPIOI->ODR |= OUTPUT & 0b00000111;
        else
            GPIOI->ODR &= OUTPUT & 0b00000111;
```

Our final program all together looks like the following:

```
// Test program
#include "stm32f7xx.h"                      // Device header

int main() {
    // initialization
    // activate peripheral
    RCC->AHB1ENR |= 1 << 0 | 1 << 8; // ports A and I

    // set pin mode
    // set switches to input
    // pin PA0 is already on input by default
    // set leds to output
    GPIOI->MODER |= 1 << 2 * 2 | 1 << 1 * 2 | 1 << 0 * 2; // output mode for PI2, PI1, PI0

    unsigned int INPUT, OUTPUT;

    while (1) {
        // get current state
        INPUT = GPIOA->IDR & 0b00000001;

        // process input
        if (INPUT == 0b00000001)
            OUTPUT |= 0b00000111;
        else
            OUTPUT &= ~0b00000111;

        // output
        if ((OUTPUT & 0b00000111) == 0b00000111)
            GPIOI->ODR |= OUTPUT & 0b00000111;
        else
            GPIOI->ODR &= OUTPUT & 0b00000111;
    }
}
```

But this code above is inefficient to write. Everytime when we need to write to the output registers, we have to check if their bits are set or not set, and decide how to write into the pins based on that. Also whenever we want to add or remove a bit in the `OUTPUT` variable, we have to counciously decide to write a 1 or a 0 with `if` and `else` statements. A better way to code is to by default set `output` to zero, and only change the bits that we need to flip. The better way to design the program is shown in diagram 2:



As we can see, the system is simpler even in flowchart form, and the code is even simpler when revised:

```
// Test program
#include "stm32f7xx.h"                    // Device header
#include "my_utils.h"

int main() {
    // initialization
```

```
    // activate peripheral
    activate_peripheral(A);
    activate_peripheral(I);

    // set pin mode
    // set switches to input
    set_register_mode(A, 0, 0);
    // set leds to output
    set_register_mode(I, 0, 1);
    set_register_mode(I, 1, 1);
    set_register_mode(I, 2, 1);

    uint INPUT, OUTPUT;

    activate_led(I, 3);

    while (1) {
        OUTPUT = 0;
        // get current state
        INPUT = GPIOA->IDR & 0b00000001;

        // process input
        if (INPUT == 0b00000001)
            OUTPUT |= 0b00000111;

        // output
        GPIOI->ODR = (GPIOI->ODR & ~0b00000111) | (OUTPUT & 0b00000111);
    }
}
```

There is also a custom library header used in the above program. The custom library provides easy ways of GPIO initialization with defines and helper functions. The helper functions also handles special cases with certain GPIOs where their reset states are not zero.

```
// my_utils.h
#ifndef MY_UTILS_H
# define MY_UTILS_H

#define A 0
#define B 1
#define C 2
#define D 3
#define E 4
#define F 5
#define G 6
#define H 7
#define I 8
#define J 9
#define K 0xa

typedef unsigned int uint;

void activate_peripheral(uint gpio);

/**
```

```
 * gpio name
 * pin  pin of gpio
 * mode 0 for read (input), 1 for write (ouput), 2 for alt function, 3 for analog
 */
void set_register_mode(uint gpio, uint pin, uint mode);

void activate_led(uint gpio, uint pin);

#endif
```

```
// my_utils.c
#include "stm32f7xx.h"                  // Device header
#include "my_utils.h"

void activate_peripheral(uint gpio) {
    RCC->AHB1ENR |= 1 << gpio;
}

void set_register_mode(uint gpio, uint pin, uint mode) {
    GPIO_TypeDef  *GPIOx;
    GPIOx = (GPIO_TypeDef *) (GPIOA_BASE + 0x400 * gpio);
    if ((gpio == A && (pin == 15 || pin == 14 || pin == 13)) ||
          (gpio == B && (pin == 4 || pin == 3))) {
        GPIOx->MODER &= ~(0x3 << pin * 2);
    }

    GPIOx->MODER |= mode << pin * 2;
}

void activate_led(uint gpio, uint pin) {
    GPIO_TypeDef  *GPIOx;
    GPIOx = (GPIO_TypeDef *) (GPIOA_BASE + 0x400 * gpio);
    GPIOx->MODER |= 1 << pin * 2;
    GPIOx->ODR   |= 1 << pin;
}
```

These header files and libraries are an attempt to simplify code when our combinational systems gets more complex further on.

Implimentation of Equations

We will try to impliment the equation in our exercize 3 handout:

$$LED0 = SW0 \cdot \overline{SW1} + SW3 \cdot SW4 \cdot \overline{SW5}$$

$$LED1 = SW7$$

$$LED2 = \overline{SW6}$$

$$LED3 = SW2 \cdot SW3$$

And the code will look like this:

```
// Combinational System 1
#include "stm32f7xx.h"                  // Device header
#include "my_utils.h"

int main() {
    // initialization
    // activate peripheral
    RCC->AHB1ENR |= 1 << A | 1 << C | 1 << F | 1 << G | 1 << I;

    // set pin mode
    // set switches to input
    for (int i = 10; i > 5; i--)
        set_register_mode(F, i, 0);
    set_register_mode(A, 15, 0);
    set_register_mode(A, 8, 0);
    set_register_mode(A, 0, 0);
    // set leds to output
    for (int i = 3; i > -1; i--)
        set_register_mode(I, i, 1);

    uint portA, portF, INPUT, OUTPUT;

    while (1) {
        OUTPUT = 0;
        // get current state
        portA = GPIOA->IDR;
        portF = GPIOF->IDR;
        // create and combine inputs
        INPUT = (portA&(1<<0)) | (portA&(1<<8)) >> 7 | (portA&(1<<15)) >> 13 | (portF&0x7c0) >> 3;

        // process input
        if ((INPUT & 0b00000011) == 0b01 || (INPUT & 0b00111000) == 0b011 << 3)
            OUTPUT |= 0b00000001;
        if ((INPUT & 0b10000000) == 0b1 << 7)
            OUTPUT |= 0b00000010;
        if ((INPUT & 0b01000000) == 0b0 << 6)
            OUTPUT |= 0b00000100;
        if ((INPUT & 0b00001100) == 0b11 << 2)
            OUTPUT |= 0b00001000;

        // output
//      GPIOG->ODR = (GPIOG->ODR & ~0b11000000) | (OUTPUT & 0b11000000);
        GPIOC->ODR = (GPIOC->ODR & ~0b11000000) | (OUTPUT << 2 & 0b11000000);
        GPIOI->ODR = (GPIOI->ODR & ~0b00001111) | (OUTPUT & 0b00001111);

    }
}
```

Our code is a bit harder to read when there are multiple lines of conditionals. Writing our combinational system out in bit form is easy to read and understand when there isn't a lot of lines, but when there are more equations like in system 1, it is cumbersome to read even when bitshifts are used as an attempt to reduce the amount of ones and zeros.

We will write our next system with hexidecimal digits, and see if it is more readable. The equations for system 2 is:

$$LED0 = SW0 \cdot SW1 \cdot SW2 \cdot SW3 + SW5 \cdot SW6 + \overline{SW7}$$

$$LED1 = SW0 \cdot SW6 \cdot \overline{SW7} + SW4 \cdot \overline{SW5} \cdot SW6$$

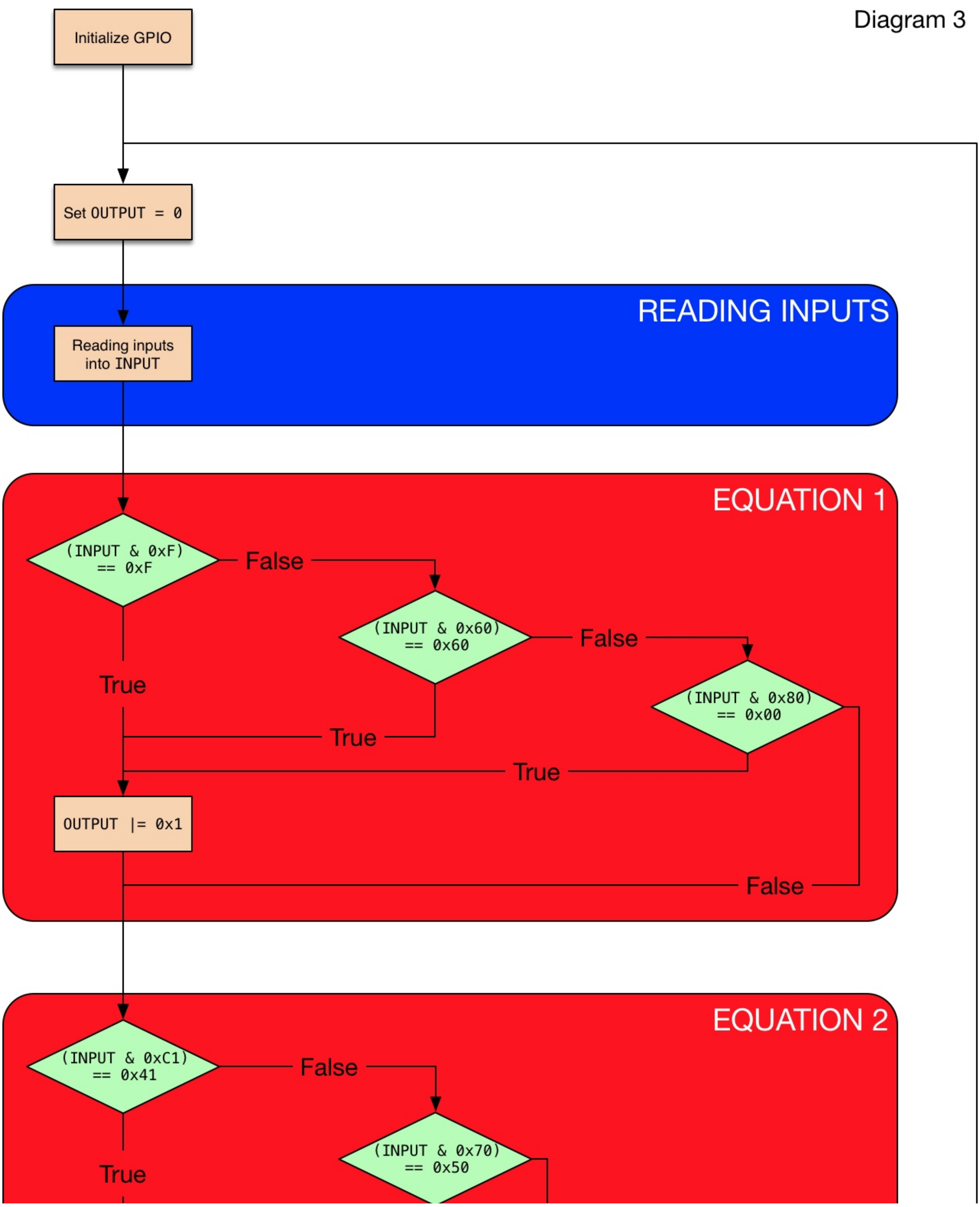
$$LED2 = SW1 \cdot SW2 \cdot SW3 \cdot SW4 + \overline{SW6} \cdot \overline{SW7}$$

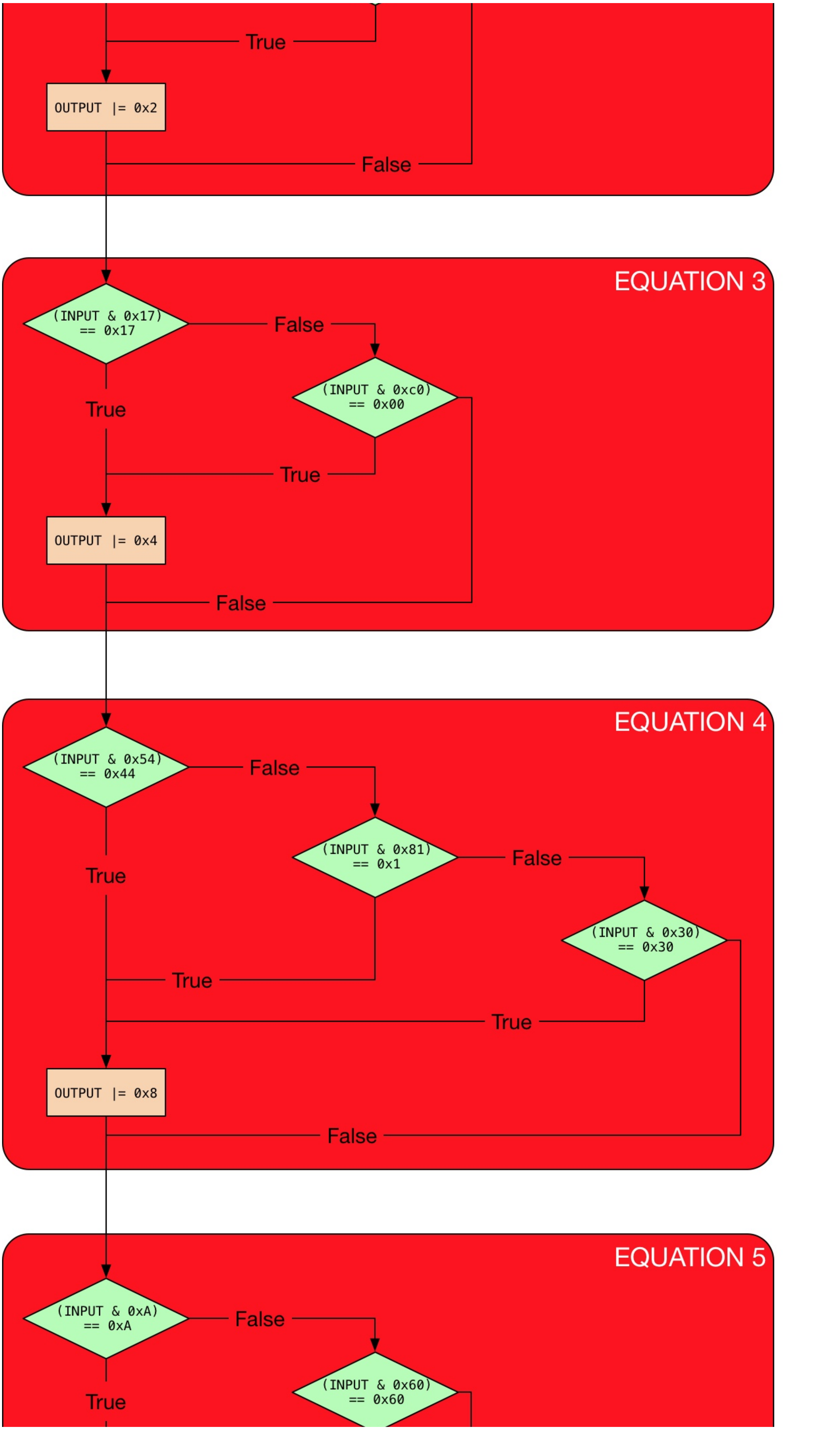
$$LED3 = SW2 \cdot \overline{SW4} \cdot SW6 + SW0 \cdot \overline{SW7} + SW4 \cdot SW5$$

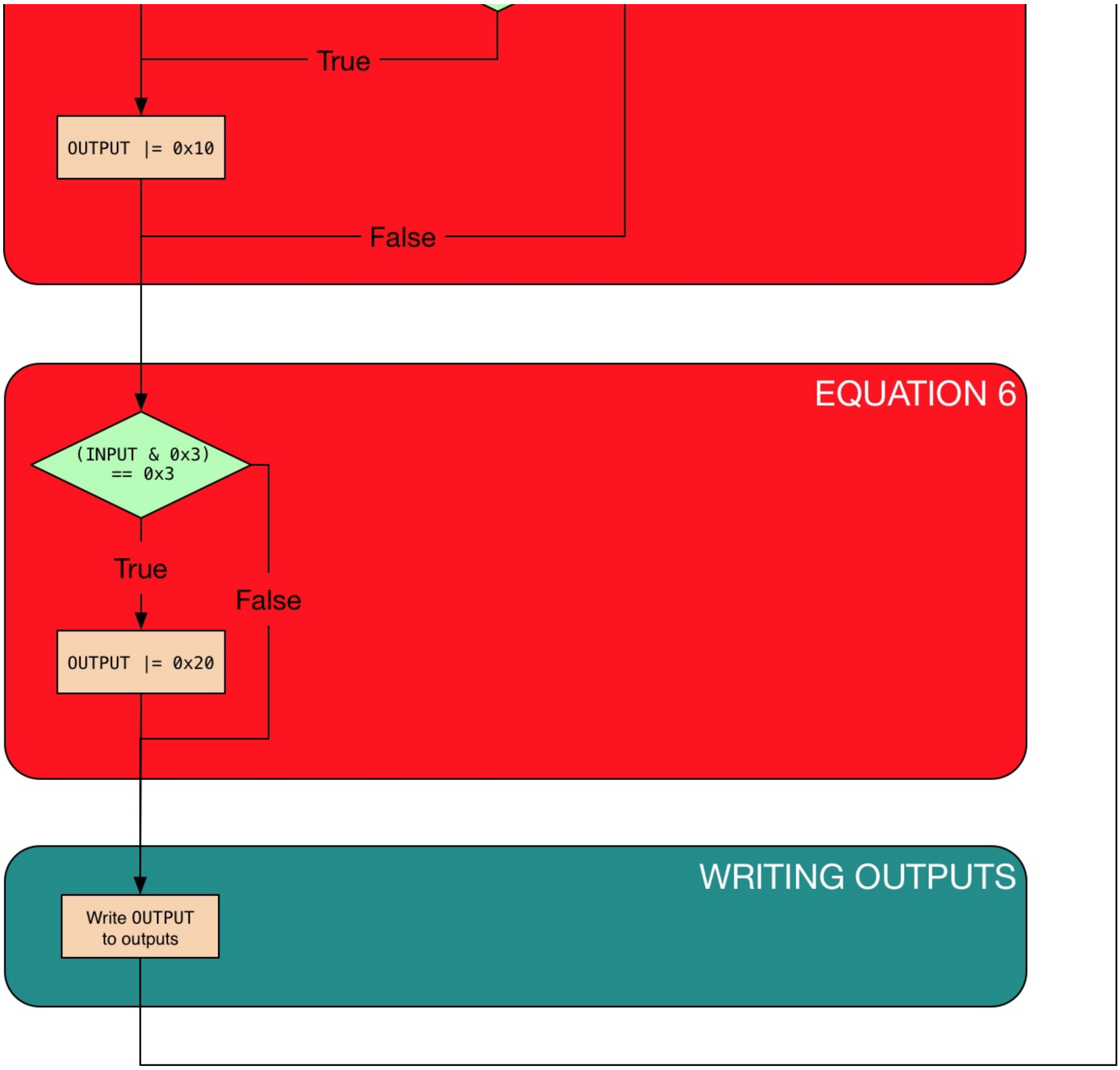
$$LED4 = SW1 \cdot SW3 + SW5 \cdot SW6$$

$$LED5 = SW0 \cdot SW1$$

The flowchart for system 2 is shown in diagram 3 below:




True
OUTPUT |= 0x2
False
EQUATION 3
(INPUT & 0x17) == 0x17
False
(INPUT & 0xc0) == 0x00
True
True
OUTPUT |= 0x4
False
EQUATION 4
(INPUT & 0x54) == 0x44
False
(INPUT & 0x81) == 0x1
False
(INPUT & 0x30) == 0x30
True
True
True
OUTPUT |= 0x8
False
EQUATION 5
(INPUT & 0xA) == 0xA
False
(INPUT & 0x60) == 0x60
True



And the code for system 2 is:

```
// Combinational System 2
#include "stm32f7xx.h"                    // Device header
#include "my_utils.h"

int main() {
    // initialization
    // activate peripheral
    RCC->AHB1ENR |= 1 << A | 1 << C | 1 << F | 1 << G | 1 << I;

    // set pin mode
    // set switches to input
    for (int i = 10; i > 5; i--)
        set_register_mode(F, i, 0);
    set_register_mode(A, 15, 0);
    set_register_mode(A, 8, 0);
    set_register_mode(A, 0, 0);
    // set leds to output
    set_register_mode(G, 7, 1);
    set_register_mode(G, 6, 1);
    set_register_mode(C, 7, 1);
```

```
    set_register_mode(C, 6, 1);
    for (int i = 3; i > -1; i--)
        set_register_mode(I, i, 1);

    uint portA, portF, INPUT, OUTPUT;

    while (1) {
        OUTPUT = 0;
      // get current state
        portA = GPIOA->IDR;
        portF = GPIOF->IDR;
        // create and combine inputs
        INPUT = (portA&(1<<0)) | (portA&(1<<8)) >> 7 | (portA&(1<<15)) >> 13 | (portF&0x7c0) >> 3;

        // process input
        if ((INPUT & 0xF) == 0xF || (INPUT & 0x60) == 0x60 || (INPUT & 0x80) == 0x00)
            OUTPUT |= 0x1;
        if ((INPUT & 0xC1) == 0x41 || (INPUT & 0x70) == 0x50)
            OUTPUT |= 0x2;
        if ((INPUT & 0x17) == 0x17 || (INPUT & 0xc0) == 0x00)
            OUTPUT |= 0x4;
        if ((INPUT & 0x54) == 0x44 || (INPUT & 0x81) == 0x01 || (INPUT & 0x30) == 0x30)
            OUTPUT |= 0x8;
        if ((INPUT & 0xA) == 0xA || (INPUT & 0x60) == 0x60)
            OUTPUT |= 0x10;
        if ((INPUT & 0x3) == 0x3)
            OUTPUT |= 0x20;

        // output
//      GPIOG->BSRR |= (OUTPUT & 0b11000000) | 0xFFFF0000;
        GPIOC->BSRR |= (OUTPUT & 0b00110000) << 2 | 0xFFFF0000;
        GPIOI->BSRR |= (OUTPUT & 0b00001111) | 0xFFFF0000;
    }
}
```

The code in system 2 is visually a lot more simpler than system 1. the only case here where the binary representation is more descriptive is when the `OUTPUT` is needed to be masked and directed to the registers. We also use the `GPIOx->BSRR` registers here for faster performance.

Compare Algorithm method against Truth Table method

Last but not least, we will compare different methods of implimenting combinational systems. The code for Combinational System 2 is an algorithmic one, where the processor steps through the instructions, performing the actions of comparing and then deciding the bits to flip. Another way to impliment the equations of system 2 is to have the corresponding bits generated before hand and have only a single line of table indexing code to process the inputs before writing the output.

The code that uses a tab as an index of all possible outputs is as follows:

```
// Combinational System 2 using TAB
#include "stm32f7xx.h"                    // Device header
#include "my_utils.h"
```

```
int main() {
    // initialization
    // activate peripheral
    RCC->AHB1ENR |= 1 << A | 1 << C | 1 << F | 1 << G | 1 << I;

    // set pin mode
    // set switches to input
    for (int i = 10; i > 5; i--)
        set_register_mode(F, i, 0);
    set_register_mode(A, 15, 0);
    set_register_mode(A, 8, 0);
    set_register_mode(A, 0, 0);
    // set leds to output
    set_register_mode(G, 7, 1);
    set_register_mode(G, 6, 1);
    set_register_mode(C, 7, 1);
    set_register_mode(C, 6, 1);
    for (int i = 3; i > -1; i--)
        set_register_mode(I, i, 1);


    uint portA, portF, INPUT, OUTPUT;
    uint TAB[1<<0x8] = {0};


    // generate truth table for equation set 2
    for (int INPUT = 0; INPUT < 1<<0x8; INPUT++) {
        if ((INPUT & 0xF) == 0xF || (INPUT & 0x60) == 0x60 || (INPUT & 0x80) == 0x00)
            TAB[INPUT] |= 0x1;
        if ((INPUT & 0xC1) == 0x41 || (INPUT & 0x70) == 0x50)
            TAB[INPUT] |= 0x2;
        if ((INPUT & 0x17) == 0x17 || (INPUT & 0xc0) == 0x00)
            TAB[INPUT] |= 0x4;
        if ((INPUT & 0x54) == 0x44 || (INPUT & 0x81) == 0x01 || (INPUT & 0x30) == 0x30)
            TAB[INPUT] |= 0x8;
        if ((INPUT & 0xA) == 0xA || (INPUT & 0x60) == 0x60)
            TAB[INPUT] |= 0x10;
        if ((INPUT & 0x3) == 0x3)
            TAB[INPUT] |= 0x20;
    }

    while (1) {
      // get current state
        portA = GPIOA->IDR;
        portF = GPIOF->IDR;
        // create and combine inputs
        INPUT = (portA&(1<<0)) | (portA&(1<<8)) >> 7 | (portA&(1<<15)) >> 13 | (portF&0x7c0)
>> 3;

        // process input
        OUTPUT = TAB[INPUT];

        // output
//      GPIOG->BSRR |= (OUTPUT & 0b11000000) | 0xFFFF0000;
        GPIOC->BSRR |= (OUTPUT & 0b00110000) << 2 | 0xFFFF0000;
```

```
        GPIOI->BSRR |= (OUTPUT & 0b00001111) | 0xFFFF0000;
    }
}
```

As we can see, code is very short, only a few lines in the while loop. Lets compare and contrast the two versions of system 2 and their run times. For the test, we will use the timing function of the ST-Link debugger and the uVision IDE to see how our long a loop takes. We will take the timing by averaging the execution time across 1,0000,0000 loops. To do that, we can simply add a variable and an if statement in our while loop:

```
// ... more code above
    uint count = 1000000;
    while (1) {
        // ... our while loop here
        count++;
        if (count > 1000000)
            count = 0;
    }
```

And break on the line when we reset the count. The instructions added to the while loop is marginal, and shouldn't affect our runtime drastically, and we can get better averages this way. The raw data from the measurements are shown below:

System 2	System 2 TAB
0.00013169	0.00140413
7.78067613	6.31419900
15.99253506	12.62699212
24.11248081	13.93978687
32.17530294	25.25258069
40.11324562	31.59587906
48.12219694	37.90867356
56.30996481	44.22146744
64.49773700	50.53425987
72.68549913	56.86930800
80.76834175	63.35801806

The average of those measurements are 44.255798 seconds for System 2, and 34.2621165 seconds for System 2. This means across 10 measurements, there is about 1 second difference for every 1000000 loops of the program, meaning the truth table is 0.00000999 seconds (9.99 microseconds) faster per loop. This is expected because using a tab should theoretically be faster than multiple if and else statements.

Summary

In summary, we learned that it is often very benifitial to setting a default state upon entering any loop we've created. The reset itself is part of the output that we will write, so it can often simplifies our code by catching all the false branch conditions, saving us to manually set the bits. Additionally we also see that often with a proper sized program, writing hex numbers are more effecient to read and understand, unless the bits are very graphically oriented. And finally, there are definitly ways to speed up our program's execution time, making our combinational system more responsive overall through optimizations like making a static truth table compared to the processor checking the conditions. There are other things that can be tested, like compareing the speed difference of `ODR` VS `BSRR` registers when it comes to how fast we can write the output. These could be some new discoveries to be done.