De La Salle University – Manila

Gokongwei College of Engineering

Department of Electronics and Computer Engineering

Microprocessor and Microcontroller Systems and Design Laboratory

LBYEC3L

Laboratory Report #2

Programming Input and Output Pins

by

Dayrit, Bettina Gaille H.

Guevarra, Gia Kyla S.

Rodriguez, Mariah Venice A.

Sulit, Keane Dwight A.

LBYEC3L – EK1

I. Introduction

The ports in a microcontroller serve as a means of communication between the microcontroller and other components or devices [1]. These ports consist of several pins that can be programmed to perform specific functions, such as receiving data or controlling other devices. The function of a pin on a port can be set as either an input or output and can be done through the use of the TRIS register. This register contains a series of bits, and the combination of ones and zeros written to this register determines the direction of the pin.

The process of input initialization involves setting the ports of the microcontroller to receive data from external devices by configuring the TRIS bits to 1. On the other hand, output initialization involves setting the ports to function as outputs to drive external devices, which is achieved by setting the TRIS bits to 0. As for the port registers, these registers are used to set the output state of a port or read the input state of a port. These ports can be used as inputs or outputs, depending on the state of their associated data direction register (TRIS register). PORTA has an 8-bit width and operates in a bidirectional manner, with the TRISA register controlling its pins. On the other hand, the pins in PORT were controlled by a TRISB register [2]. For some microcontrollers, the port and TRIS registers have been separated into separate banks for the purpose of streamlined access and control. The port registers are in bank 1, and the TRIS registers are in bank 0 [3].

The objectives of the experiment are to gain understanding and knowledge on simple microcontroller instructions, particularly for the PIC16F84A microcontroller [4]. The following sub-objectives are as follows:

1. To be able to create a logical and working program using assembly language;
2. To be able to program input and output pins using assembly language; and
3. To be able to understand how machine language and assembly language works.

II. Methodology

A. Materials

The materials used for the conduct of the experiment are as follows:

Electronic Component	Quantity
PIC16F84A	1
LED-RED	1
Common Anode 7-Segment Display	1
1k Resistor	4
5 VDC Supply	1
7805 IC	1
Crystal Oscillator	1
SW-SPDT	4

Table 1. List of Electronic Components used for Simulation

Moreover, the experimenters utilized Proteus 8 Professional for circuit simulation and MPLAB IDE and Visual Studio Code for code development.

B. Procedure

Exercise #1:

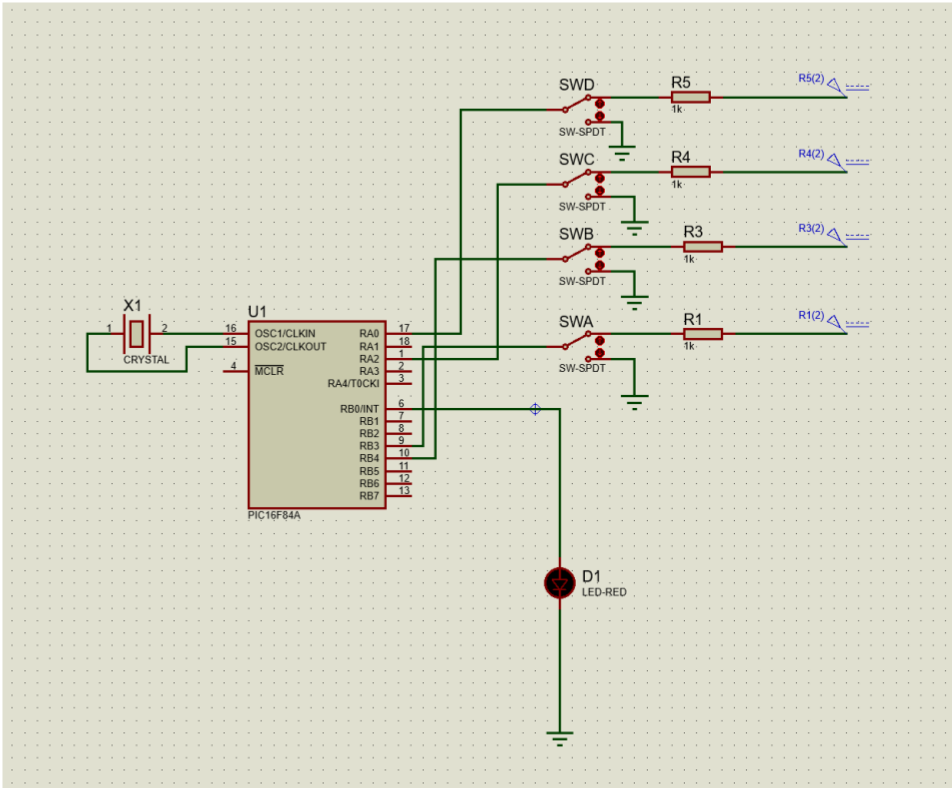


Figure 1. Circuit Configuration for Exercise #1

For Exercise #1, the task was to build both a code and a circuit that follows the logic gate XNOR; this should have 4 inputs (switches), RB3, RB4, RA2, and RA0, respectively. Its output should be found in the pin RB0, where an LED light serves as the indicator; turned on means that the logic is set at 1 while turned off means that the logic is set at 0. The first step of this exercise is to build the circuit proper in Proteus. Taking note of where the input and output pins are, RB3, RB4, RA2, and RA0 all have switches connected to them, while RB0 has a red LED connected to it. Ensure the proper grounding, as well as the proper connection to the power supply. A crystal oscillator is also connected at pins 15 and 16. After building the circuit, comes the programming. However, to first understand the concept of this exercise, it is essential to map out the truth table of the assigned logic gate to determine which configurations output a logic 1. As seen in Table 2, it can be observed that this specific logic gate, XNOR, has an odd parity, wherein an even number of 1s should have an output logic of 1.

XNOR 4-INPUT TRUTH TABLE					
SEGMENT	SW_A	SW_B	SW_C	SW_D	OUTPUT
PIN	*RB3*	*RB4*	*RA2*	*RA0*	*RB0*
	0	0	0	0	1
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	1
	0	1	1	0	1
	0	1	1	1	0
	1	0	0	0	0
	1	0	0	1	1
	1	0	1	0	1
	1	0	1	1	0
	1	1	0	0	1
	1	1	0	1	0
	1	1	1	0	0
	1	1	1	1	1

Table 2. 4-Input XNOR Truth Table

With the truth table established, the code can now be created; as seen in Table 2, where the inputs and outputs are mapped out, this will establish the first part of the code: Initialization. Hexadecimal number system is used upon coding. It is worth noting that 1's represents inputs, while 0's represent output—from Table 2, all coincide with the requirements of Exercise #1. After initializing comes the rest of the codes. The main code used for this is *BTFSS*, which tests the bit if the logic is at 1. When logic is 1, it skips the next line and goes over to the line after that to run; however, when logic is 0, it goes directly to the next line of code to run. The expected result of the code should test all four switches, eventually having 16 different configurations once the final switch is being tested.

INITIALIZATION									HEX
PORTA	-	-	-	*RA4*	*RA3*	*RA2*	*RA1*	*RA0*	0X05
	0	0	0	0	0	1	0	1	
PORTB	*RB7*	*RB6*	*RB5*	*RB4*	*RB3*	*RB2*	*RB1*	*RB0*	0X18
	0	0	0	1	1	0	0	0	

Table 3. Initialization Table for Exercise #1

Finally, once the code has shown "build succeeded" upon exporting, this is then imported to the Proteus file in the microcontroller for testing. Testing should cover all 16 different configurations of the switches according to the truth table, and its following expected output as indicated by the LED.

Exercise #2:

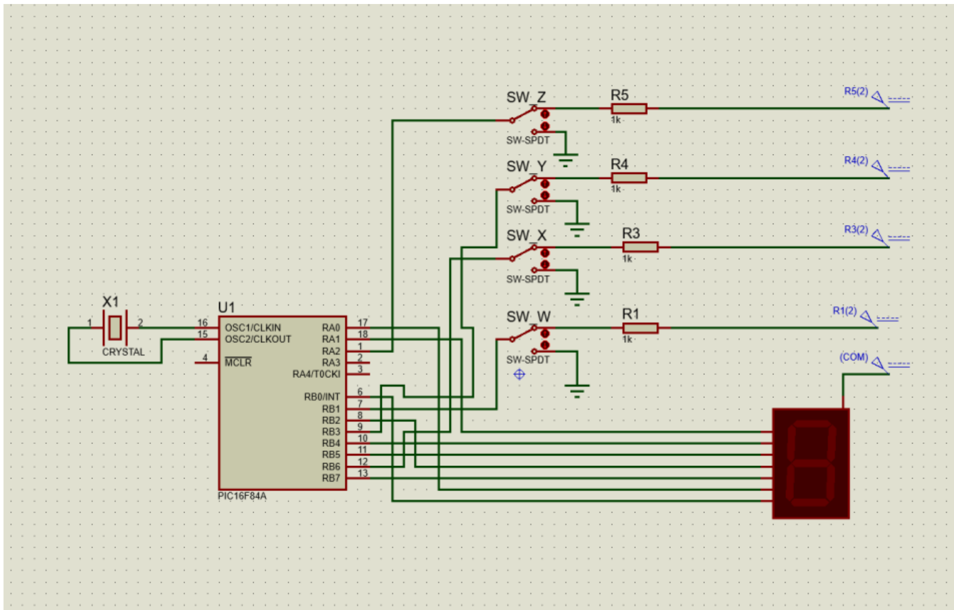


Figure 2. Circuit Configuration for Exercise #2

The objective of this stage of the experiment is to activate the 7-segment display to show decimal numbers from 0 to 9 and hexadecimal letters A to F. The circuit design employed for this exercise is depicted in Figure 2. It is noteworthy that the circuit from Exercise #1 was employed, with modifications made to the components and connections. Specifically, the LEDs were replaced with a 7-segment display and the connections of the switches were adjusted. The students were provided with specifications that demonstrate the connections of each individual pin. The switches were connected to pins RB1, RB6, RB3, and RA2. The connections for the 7-segment display were as follows: Pin A of the 7 segment was connected to RA1, Pin B was connected to RB4, Pin C was connected to RB5, Pin D was connected to RB2, Pin E was connected to RB7, Pin F was connected to RA0, and Pin G was connected to RB0. Moreover, a crystal oscillator was also connected to pins 15 and 16.

After constructing the circuit, the same steps as in the previous exercise were followed. The next step is creating a truth table for the BCD to common anode 7-segment decoder, which plays an important role in the design and verification of the logic circuit. The truth table sown in figure x displays all possible input combinations and the associated output states, enabling the student to evaluate the accuracy of the behavior of the decoder. Moreover, the truth table provides a clear understanding of the intended result, making it easier to write the code for the MPLAB.

BCD TO COMMON ANODE 7-SEGMENT DECODER												
DEC	SEGMENT	SW_W	SW_X	SW_Y	SW_Z	A	B	C	D	E	F	G
	PIN	*RB1*	*RB6*	*RB3*	*RA2*	*RA1*	*RB4*	*RB5*	*RB2*	*RB7*	*RA0*	*RB0*
0		0	0	0	0	0	0	0	0	0	0	1
1		0	0	0	1	1	0	0	1	1	1	1
2		0	0	1	0	0	0	1	0	0	1	0
3		0	0	1	1	0	0	0	0	1	1	0
4		0	1	0	0	1	0	0	1	1	0	0
5		0	1	0	1	0	1	0	0	1	0	0
6		0	1	1	0	0	1	0	0	0	0	0
7		0	1	1	1	0	0	0	1	1	1	1
8		1	0	0	0	0	0	0	0	0	0	0
9		1	0	0	1	0	0	0	0	1	0	0
A		1	0	1	0	0	0	0	1	0	0	0
b		1	0	1	1	1	1	0	0	0	0	0
C		1	1	0	0	0	1	1	0	0	0	1
d		1	1	0	1	1	0	0	0	0	1	0
E		1	1	1	0	0	1	1	0	0	0	0
F		1	1	1	1	0	1	1	1	0	0	0

Table 4. Truth Table of BCD to Common Anode 7-Segment Decoder

After constructing the circuit, the same steps as in the previous exercise were followed. The next step is creating a truth table for the BCD to common anode 7-segment decoder, which plays an important role in the design and verification of the logic circuit. The truth table displays all possible input combinations and the associated output states, enabling the student to evaluate the accuracy of the behavior of the decoder. Moreover, the truth table provides a clear understanding of the intended result, making it easier to write the code for the MPLAB.

INITIALIZATION									HEX
PORTA	-	-	-	***RA4***	***RA3***	***RA2***	***RA1***	***RA0***	0x04
	0	0	0	0	0	1	0	0	
PORTB	***RB7***	***RB6***	***RB5***	***RB4***	***RB3***	***RB2***	***RB1***	***RB0***	0x4A
	0	1	0	0	1	0	1	0	

Table 5. Initialization Table for Exercise #2

The process of creating the code in MPLAB is similar to what was done in a previous exercise. However, in this part, the task involves multiple outputs instead of just one. The initialization table shown in Table 5 helps ensure that the program runs correctly and prevents unexpected behavior or errors. Unlike in the previous exercise, the primary code utilized is BTFSC. The two functions, BTFSC and BTFSS, have distinct and opposite behavior. When the logic is set to 1, the BTFSC function will not skip the subsequent line of code. However, if the logic is cleared and set to 0, the next instruction will be skipped.

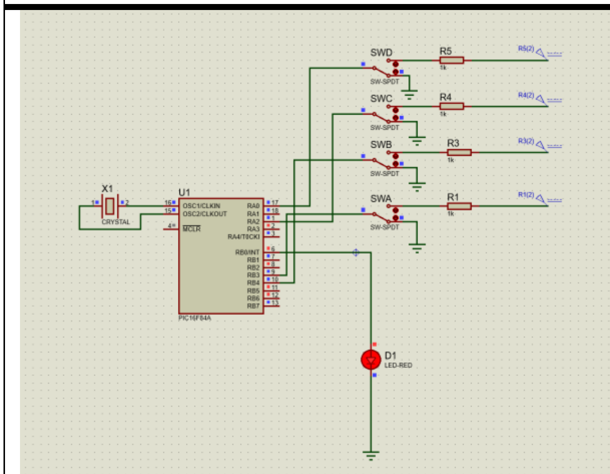
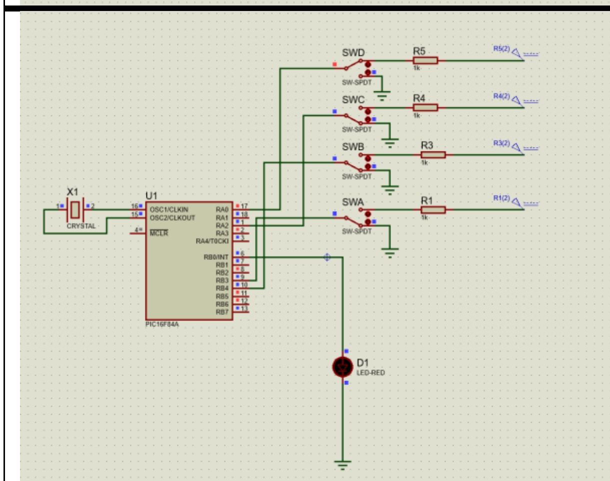
I/O CONFIGURATIONS									BIN	HEX
ZERO	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	0	1	0	1	1	1001011	4B
ONE	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	1	1	11111	1F
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	0	1	1	1	1	11001111	CF
TWO	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	1	11101	1D
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	1	0	1	0	1	0	1101010	6A
THREE	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	1	11101	1D
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	0	1	0	1	0	11001010	CA
FOUR	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	1	0	11110	1E
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	0	1	1	1	0	11001110	CE
FIVE	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	1	1	0	1	0	11011010	DA
SIX	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	1	1	0	1	0	1011010	5A
SEVEN	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	1	11101	1D
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	0	1	1	1	1	11001111	CF
EIGHT	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	0	1	0	1	0	1001010	4A
NINE	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	1	1	0	0	1	0	1	0	11001010	CA
A	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	0	1	1	1	0	1001110	4E
b	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	1	0	11110	1E
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	1	1	0	1	0	1011010	5A
C	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	1	1	1	0	1	1	1111011	7B
d	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	1	1	11111	1F
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	0	0	1	0	1	0	1001010	4A
E	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	1	1	1	0	1	0	1111010	7A
F	-	-	-	RA4	RA3	RA2	RA1	RA0		
	0	0	0	1	1	1	0	0	11100	1C
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0		
	0	1	1	1	1	1	1	0	1111110	7E

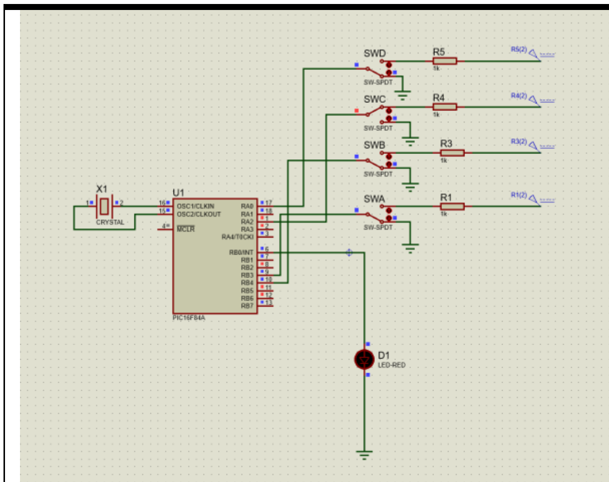
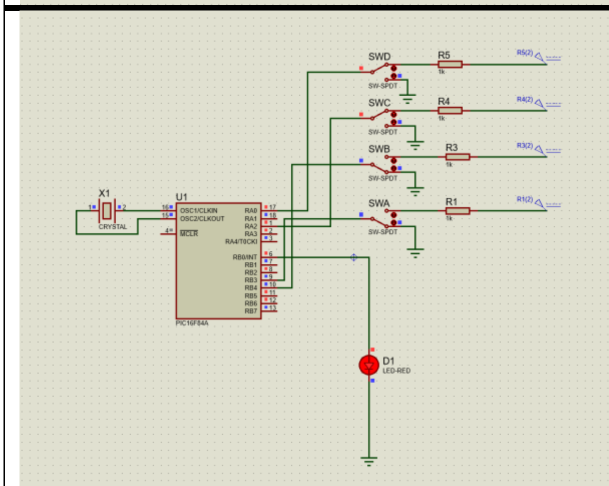
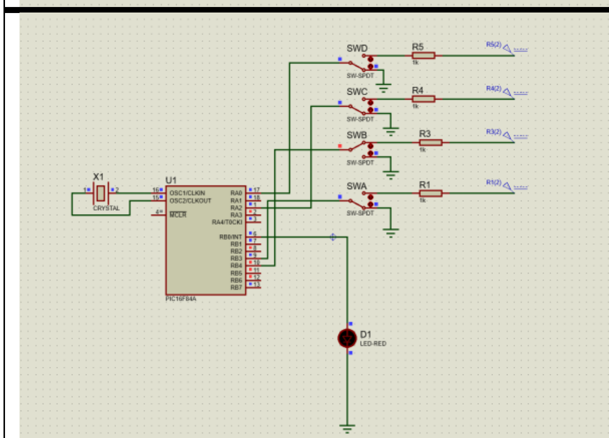
Table 6. I/O Configurations of each Segments

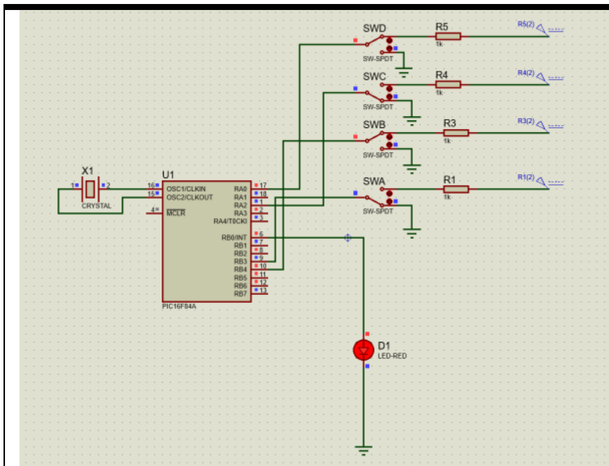
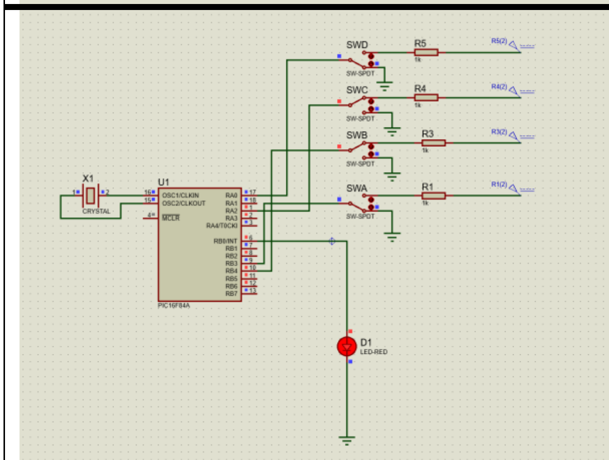
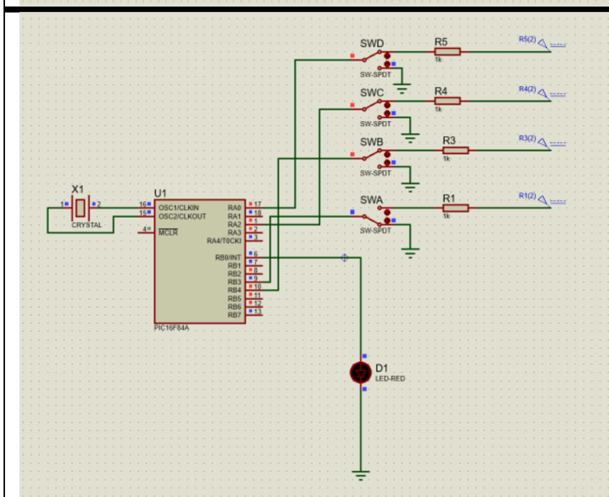
The next step after writing the code is to compile it, which involves checking for any errors in the code. If there are no errors, the program will be successfully built. Then, the compiled code is programmed into the microcontroller. The 7-segment display is then tested to confirm it is displaying the desired digits, with the I/O Configurations of each segment shown in Table 6.

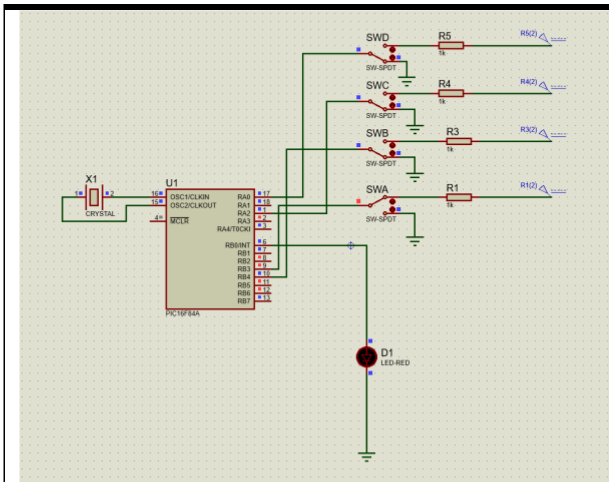
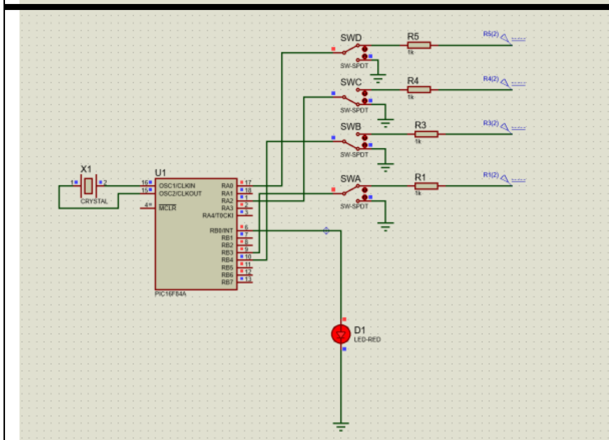
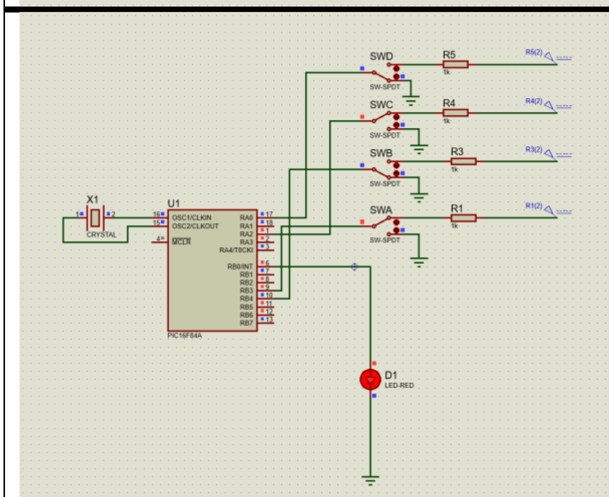
III. Results and Discussion

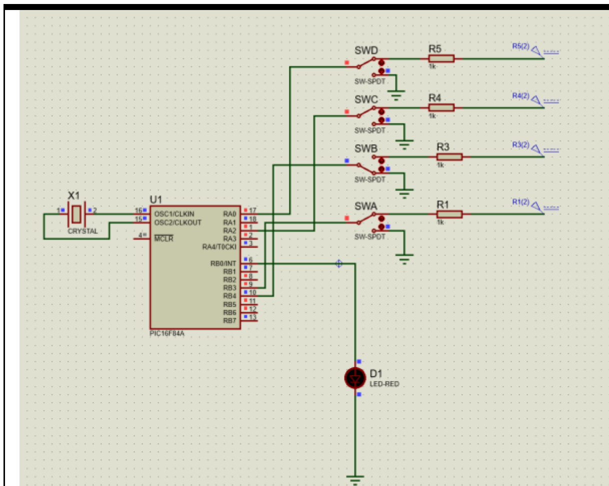
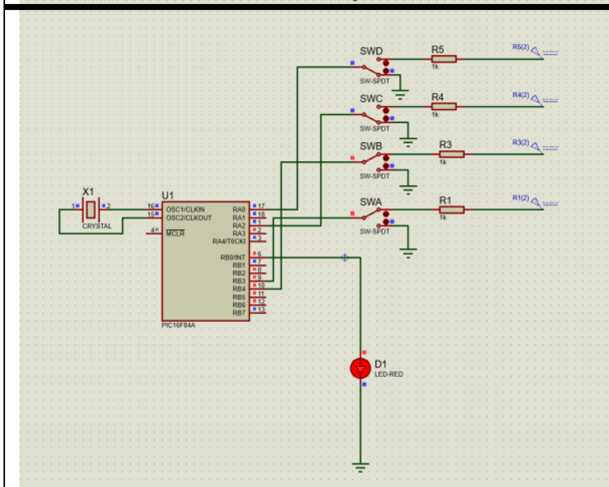
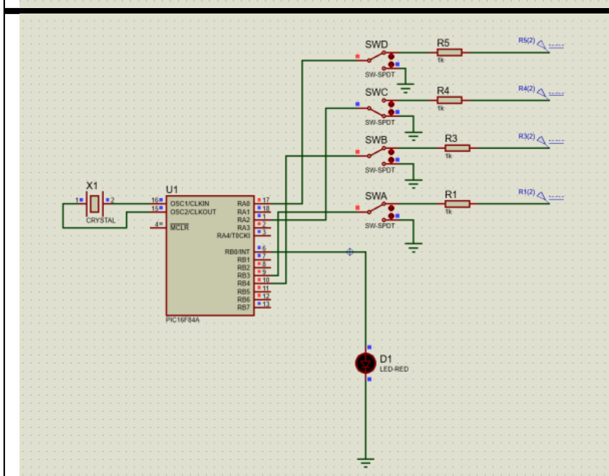
The following section displays the results gathered from the experiment.

Result	Description
	When the input is 0000, the LED will light up.
	When the input is 0001, the LED will not light up.

	When the input is 0010, the LED will not light up.
	When the input is 0011, the LED will light up.
	When the input is 0100, the LED will not light up.

	When the input is 0101, the LED will light up.
	When the input is 0110, the LED will light up.
	When the input is 0111, the LED will not light up.

	When the input is 1000, the LED will not light up.
	When the input is 1001, the LED will light up.
	When the input is 1010, the LED will light up.

	When the input is 1011, the LED will not light up.
	When the input is 1100, the LED will light up.
	When the input is 1101, the LED will not light up.

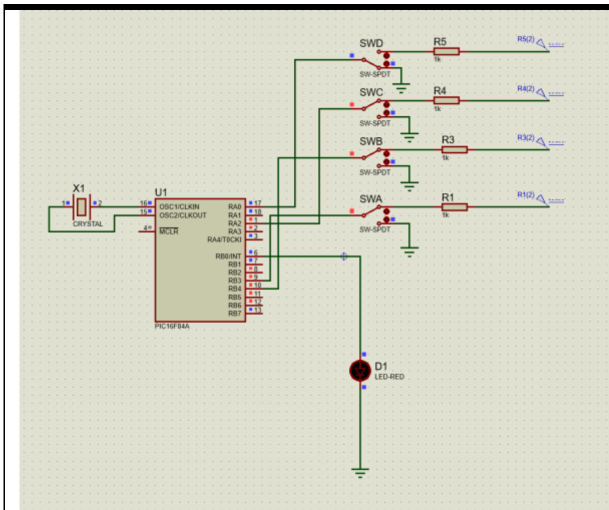
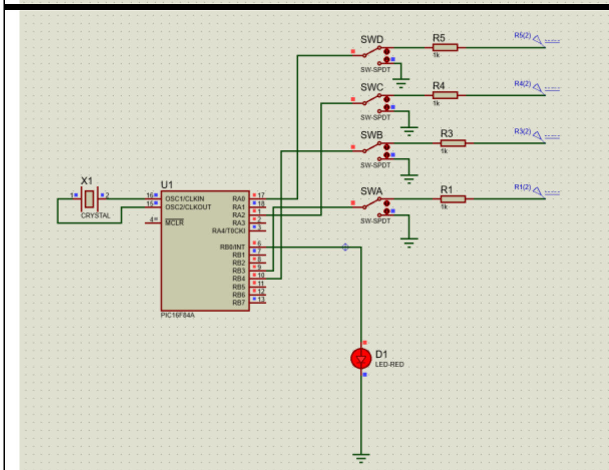
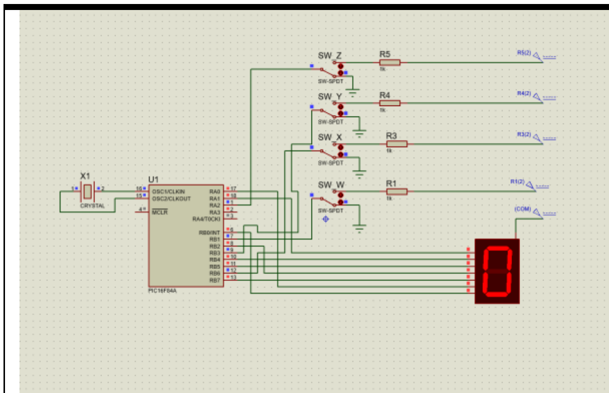
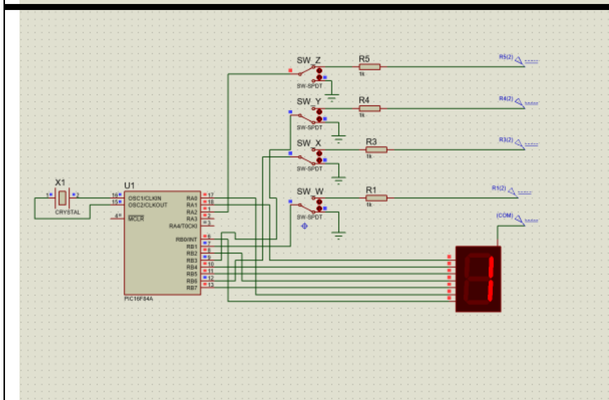
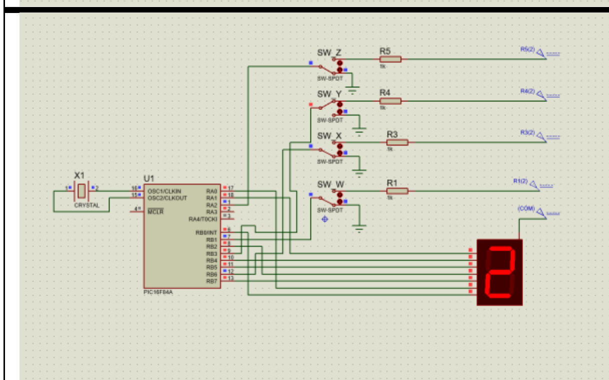
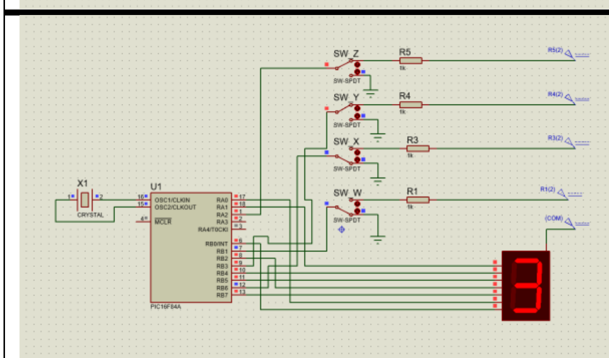
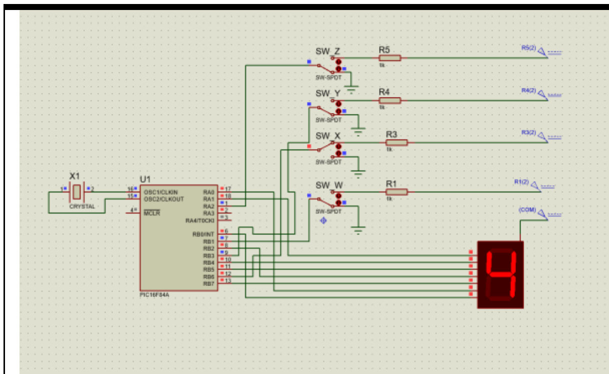
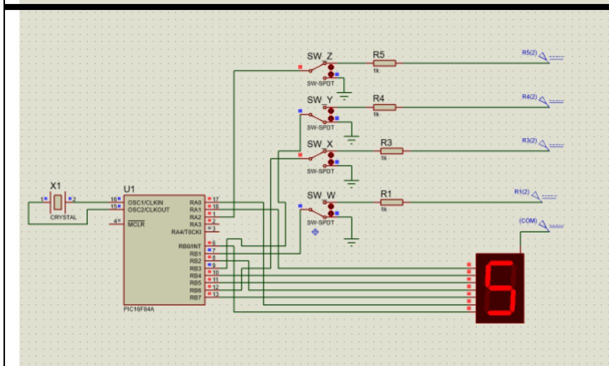
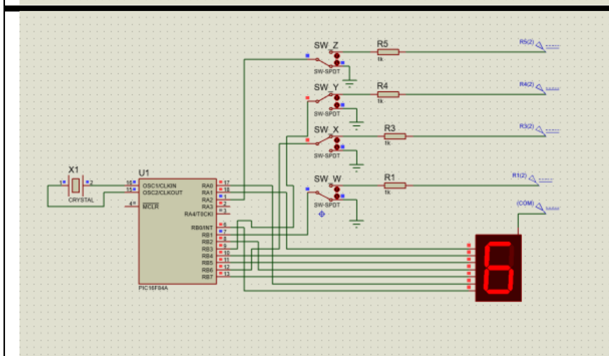
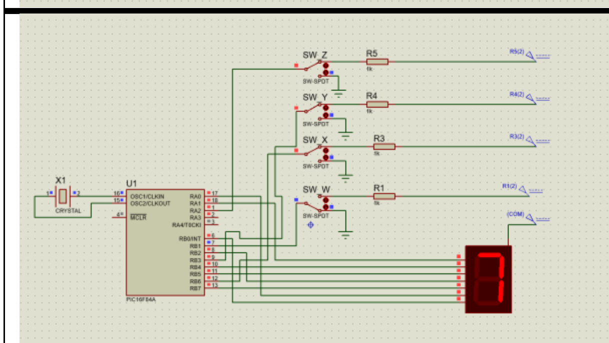
	When the input is 1110, the LED will not light up.
	When the input is 1111, the LED will light up.

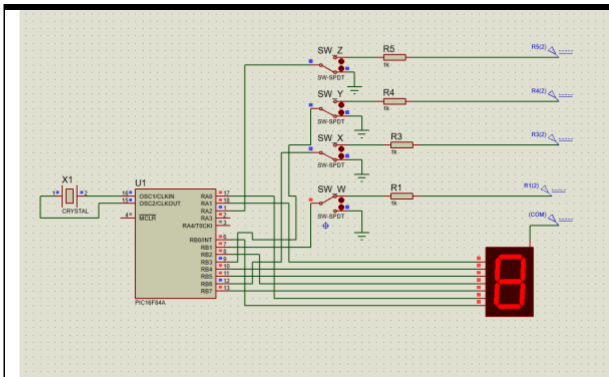
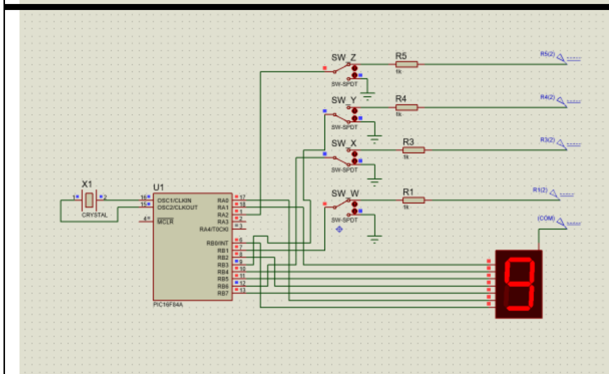
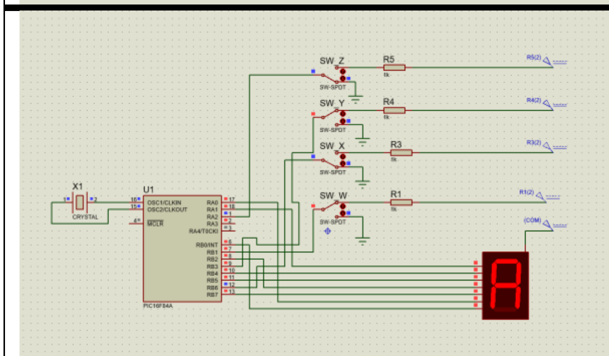
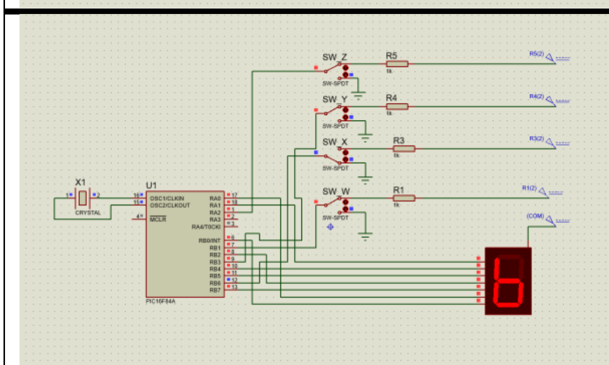
Table 7. Results of Exercise #1 Simulations

Since XNOR is an odd parity, the parity of the given binary must be even in order for the LED to light up. With this, the code implemented in the PIC16F84a microcontroller satisfies the given condition. The binary equivalent of 0 to 15 were tested in the in the microcontroller through the code implemented. As seen in Table 7, the data gathered shows that whenever the binary has even number of 1s, the LED will light up and will serve as the parity bit to satisfy the XNOR condition of having an odd parity. Otherwise, the LED is switched off.

Result	**Description**

	When BCD input is 0000, the output is 0_{10}
	When BCD input is 0001, the output is 1_{10}
	When BCD input is 0010, the output is 2_{10}
	When BCD input is 0011, the output is 3_{10}

	When BCD input is 0100, the output is 4_{10}
	When BCD input is 0101, the output is 5_{10}
	When BCD input is 0110, the output is 6_{10}
	When BCD input is 0111, the output is 7_{10}

<table>
<tr><td>
</td><td>When BCD input is 1000, the output is 8_{10}</td></tr>
<tr><td>
</td><td>When BCD input is 1001, the output is 9_{10}</td></tr>
<tr><td>
</td><td>When BCD input is 1010, the output is A_{16}</td></tr>
<tr><td>
</td><td>When BCD input is 1011, the output is B_{16}</td></tr>
</table>

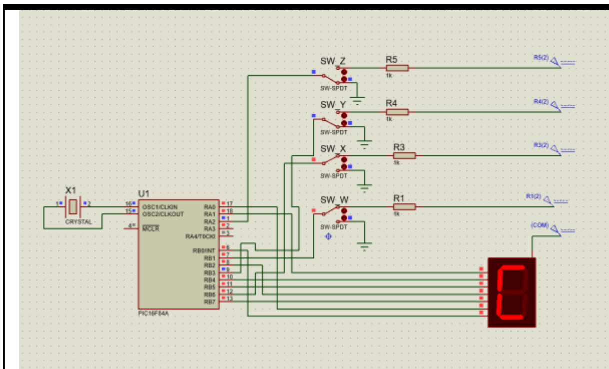
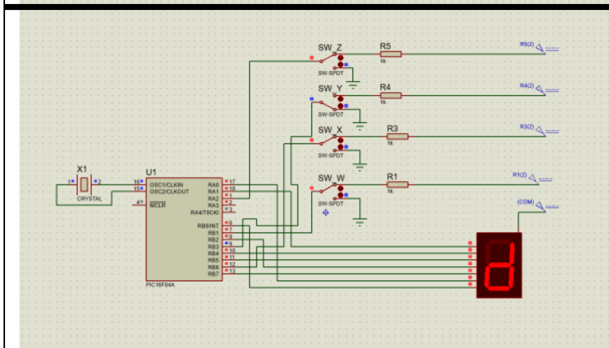
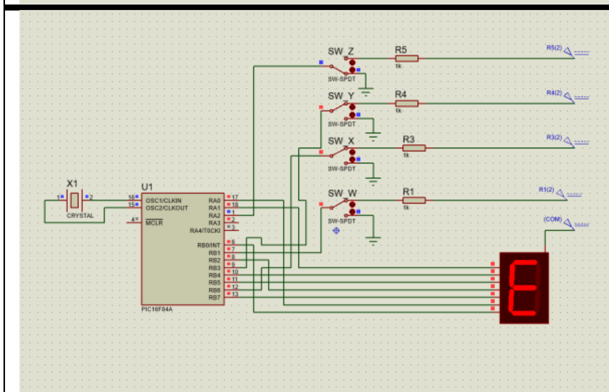
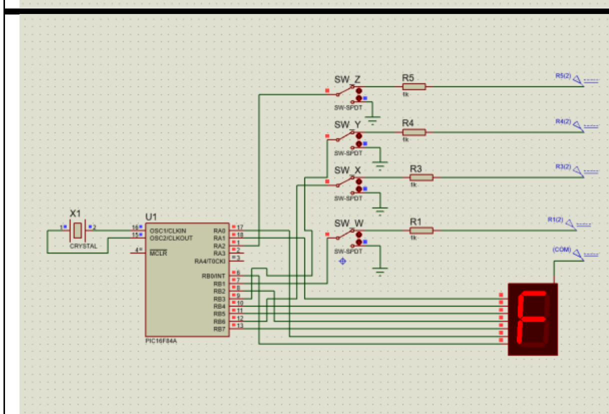
	When BCD input is 1100, the output is C_{16}
	When BCD input is 1101, the output is D_{16}
	When BCD input is 1110, the output is E_{16}
	When BCD input is 1111, the output is F_{16}

Table 8. Results of Exercise #2 Simulations

In the second exercise, the task was to display in a common anode 7-segment display the numbers 0 to 9 and letters A to F. However, the pinouts where the 7-segment display must be connected to the microprocessor are specified. As seen in Table 6 of the procedures section, certain pinouts of the microprocessor are specified to be connected to the pinouts of the 7-segment display. With this, the configurations of the code to display the required outputs are modified according to the output pinouts.

IV. Analysis and Conclusion

After the creation of the codes in MPLAB and building the circuit in Proteus, with Exercise #1, the group has achieved the objectives. First, they were able to create a logical and working program, as observed with the results shown in the previous section where it coincides with the expected output, as detailed in the truth table. Another indication of this is the "build succeeded" as the codes were saved and exported. Next was that the students were able to program input and output pins; from the previous experiment where the group used binary for coding the input and output pins, they now utilized hexadecimal as means to initialize the input and output pins. Lastly, they were able to understand how machine language and assembly language works as seen through the successful runs and achieved requirement of the exercise proper.

With all these said, the key findings and learnings of the group are as follows; it is essential to be organized and to be systematic upon coding what was required in Exercise #1; given that it has four (4) different inputs, it is expected that working with a logic gate (XNOR), the ending conditions are 16 different combinations of the input logic. Therefore, it is needed that an individual must be organized upon coding in order to keep up with the flow of the code, as well as for easier troubleshooting. In line with this, a new line of code was used, "*BTFSS"*, wherein it can cause additional confusion to the individual if not tracked properly. Additionally, another key learning from the group was proper technique of initializing using different number systems, namely binary and hexadecimal. Again, from the previous experiment, although the group used the same codes *MOVLW* and *MOVFW,* binary was used to initialize the inputs and outputs, now, the binary was converted to hexadecimal where instead of the *B''* to symbolize that it is binary, it uses *0X* to symbolize that it is in hexadecimal—proper conversion to hexadecimal according to where the 1's (inputs) and 0's (outputs) are. Lastly, as briefly stated earlier, the code *BTFSS* was heavily used in the programming part of this laboratory exercise. The students analyzed that this is comparable to an if-else statement, or for the context of how it was used in the codes, a nested if-else statement.

Upon testing, one of the observed pitfalls was the grounding and the connection to the power supply of the circuit. Initially, the students assumed that the error came from the code, thus debugging and troubleshooting of the program ensued. However, no errors were found in the code itself, so the researchers looked into the circuit built. From there, it was observed that even when the switches are turned on, there is no supply of power coming from the Vcc—hence error occurred. Adjustments were done so that the supply gives power to all four switches—with this, success of both the codes and the circuit happened. With the success of this exercise, the recommendations of the students include: trying and exploring more logic gates outside the assigned logic gate, XNOR. Moreover, one can modify the number of programmed input and output. For example, a student can decrease the inputs to three (3), however, they should also increase the outputs to two (2), instead of just one LED.

Similar to Exercise #1, Exercise #2 deals with programming input and output pins in PIC microcontrollers. Although in this case, the experimenters needed to deal with multiple outputs. Given these modified conditions, the experimenters still met the objectives of the experiment. The built program is logical and working as seen in the above results. Flipping the switches means the user is setting the BCD input. With these inputs, the code was able to decode it into lighting up a 7-segment common anode display from 0-F. Observing the codes for this exercise, it can be inferred that the logic is similar to the overall flow of the first exercise of this experiment. Hence, this paved the way into enhancing the group's capabilities in programming input and output pins of PIC microcontrollers. Furthermore, it also expounded the group's knowledge on the difference of assembly and machine language, which was also covered way back the previous experiment.

The main difference of Exercise #2 is that the experimenters are now dealing with multiple outputs as compared to the previous one. Exercise #2 also highlighted the other way of bit testing in PIC microcontrollers – *BTFSC*. *BTFSC* translates to *"bit test f, skip if clear"*. The instruction tests the bit in a specific address and skips a line in the written assembly code if the status is currently cleared. For both exercises, this specific instruction was utilized for conditional branching. Since there are multiple inputs (A, B, C, and D for Exercise #1 and W, X, Y, and Z for Exercise #2) in the circuit, several cases are needed to be implemented within the code. The goal of the experimenters were to cover all 16 possible cases of the input (2^n; $2^4 = 16$ where n is the number of inputs). With all that taken into consideration, the role of the *BTFSC* instruction is to test the bits of the BCD value, taken from the user input, and identify which segment displays should be turned on or off in order to show the right digit. Looking at the results in the previous section, all BCD values from 0-9, as well as the extra hexadecimal letters were properly lit up.

The common pitfalls of the experimenters for this part of the experiment were particular in coding the right address for the I/O configuration of the segment displays and the input loops. The first encountered bug involved some numbers and letters that are incorrectly lit up, while the other expected outputs are working fine. Upon debugging, the group discovered that some of the literals are incorrect. Another bug occurred upon resolving the first one. This time, 9_{10} and A_{16} were improperly displayed. Although it is similar to the first bug, the case this time is that the input being bit tested is inappropriately coded. Resolving this issue, the experimenters carefully made sure that the addressing of the inputs and outputs are executed accurately.

The experiment covered all aspects of programming input and output pins of the PIC16F84A. A recommendation is to implement the interfacing exercises in other microcontrollers with more ports for flexibility. The experimenters also gained skills in assembly programming, particular in PIC microcontrollers. It is notable to use meaningful labels for bit testing, especially if dealing with conditional branching, as encountered in both of the exercises of this experiment. Also, initialization of pins whether it is being used as inputs or outputs must be carefully done to ensure that the overall testing of the experiment plant is smooth and seamless. Loops, if possible, must also be implemented instead of coding the branches judiciously as the logical flow is hard to follow when there are too many branches, making the overall testing slow and prone to errors. Lastly, source control such as GitHub are recommended to be used in succeeding experiments so that the source files are properly documented to easily backtrack bugs. It also allows to keep track of changes made by each collaborator in the code, and in some instances, revert to prior versions of the source file. This provides sufficient backup, recovery and contingency plans in more complicated exercises of the course.

All in all, the objectives of the experiment were met as the experimenters were able to explore the programming of the input and output pins of a PIC microcontroller. Moreover, the experimenters were able to create logical and fully functional programs using assembly language, program a hardware circuit implemented via simulation, and further understand how the coded assembly language are translated into compiled machine language to drive digital systems.

V. References

[1] “PIC microcontrollers : chapter 2 - Microcontroller PIC16F84,” *www.matidavid.com*. http://www.matidavid.com/pic/picbook_site/2_05chapter.htm (accessed Feb. 08, 2023).
[2] “inputoutput-ports,” *MIKROE*. https://www.mikroe.com/ebooks/pic-microcontrollers-programming-in-c/inputoutput-ports (accessed Feb. 08, 2023).
[3] “The TRIS and PORT registers - Microchip PIC microcontroller,” *www.pcbheaven.com*. http://www.pcbheaven.com/picpages/The_TRIS_and_PORT_registers/ (accessed Feb. 08, 2023).
[4] Microchip Technology, “PIC16F84A Data Sheet,” 2001. Available: https://ww1.microchip.com/downloads/en/devicedoc/35007b.pdf

VI. Appendix

1) Exercise #1. Lighting an LED using 4-Input XNOR Implementation

```
;======================================;
; Author: Charmbix                     ;
; Version: 2.0                         ;
; Course: LBYEC3F - EK1                ;
; Title: Experiment 2: Exercise #1     ;
;======================================;

; INITIALIZING

BSF 03h, 5
MOVLW 0x05
MOVWF 85h
BCF 03h, 5

BSF 03h, 5
MOVLW 0x18
MOVWF 86h
BCF 03h, 5

; ==============================

; Test SW1
Start:
BTFSS 06h, 3
goto SW1_OFF ; SW1=0
goto SW1_ON ; SW1=1

; Test SW2
SW1_OFF: ; WHEN SW1=0
BTFSS 06h, 4
goto SW2_OFF_OFF ; SW1=0, SW2=0
goto SW2_OFF_ON; SW1=0, SW2=1

SW1_ON: ; WHEN SW1= 1
BTFSS 06h, 4
```

```
goto SW2_ON_OFF ; SW1=1, SW2=0
goto SW2_ON_ON ; SW1=1, SW2=1

; Test SW3
SW2_OFF_OFF: ; WHEN SW1=0, SW2=0
BTFSS 05h, 2
goto SW3_OFF_OFF_OFF ; SW1=0, SW2=0; SW3=0
goto SW3_OFF_OFF_ON ; SW1=0, SW2=0; SW3=1

SW2_OFF_ON: ; WHEN SW1=0, SW2=1
BTFSS 05h, 2
goto SW3_OFF_ON_OFF ; SW1=0, SW2=1; SW3=0
goto SW3_OFF_ON_ON ; SW1=0, SW2=1; SW3=1

SW2_ON_OFF: ; WHEN SW1=1, SW2=0
BTFSS 05h, 2
goto SW3_ON_OFF_OFF ; SW1=1, SW2=0; SW3=0
goto SW3_ON_OFF_ON ; SW1=1, SW2=0; SW3=1

SW2_ON_ON: ; WHEN SW1=1, SW2=1
BTFSS 05h, 2
goto SW3_ON_ON_OFF ; SW1=1, SW2=1; SW3=0
goto SW3_ON_ON_ON ; SW1=1, SW2=1; SW3=1

; Test SW4
SW3_OFF_OFF_OFF: ; WHEN SW1=0, SW2=0; SW3=0
BTFSS 05h, 0
goto ON ; SW1=0, SW2=0; SW3=0; SW4= 0
goto OFF ; SW1=0, SW2=0; SW3=0; SW4= 1

SW3_OFF_OFF_ON: ; WHEN SW1=0, SW2=0; SW3=1
BTFSS 05h, 0
goto OFF ; SW1=0, SW2=0; SW3=1; SW4= 0
goto ON ; SW1=0, SW2=0; SW3=1; SW4= 1

SW3_OFF_ON_OFF: ; WHEN SW1=0, SW2=1; SW3=0
BTFSS 05h, 0
```

```
goto OFF ; SW1=0, SW2=1; SW3=0; SW4= 0
goto ON ; SW1=0, SW2=1; SW3=0; SW4= 1

SW3_OFF_ON_ON: ; WHEN SW1=0, SW2=1; SW3=1
BTFSS 05h, 0
goto ON ; SW1=0, SW2=1; SW3=1; SW4= 0
goto OFF ; SW1=0, SW2=1; SW3=1; SW4= 1

SW3_ON_OFF_OFF: ; WHEN SW1=1, SW2=0; SW3=0
BTFSS 05h, 0
goto OFF ; SW1=1, SW2=0; SW3=0; SW4= 0
goto ON ; SW1=1, SW2=0; SW3=0; SW4= 1

SW3_ON_OFF_ON: ; WHEN SW1=1, SW2=0; SW3=1
BTFSS 05h, 0
goto ON ; SW1=1, SW2=0; SW3=1; SW4= 0
goto OFF ; SW1=1, SW2=0; SW3=1; SW4= 1

SW3_ON_ON_OFF: ; WHEN SW1=1, SW2=1; SW3=0
BTFSS 05h, 0
goto ON ; SW1=1, SW2=1; SW3=0; SW4= 0
goto OFF ; SW1=1, SW2=1; SW3=0; SW4= 1

SW3_ON_ON_ON: ; WHEN SW1=1, SW2=1; SW3=1
BTFSS 05h, 0
goto OFF ; SW1=1, SW2=1; SW3=1; SW4= 0
goto ON ; SW1=1, SW2=1; SW3=1; SW4= 1

ON:
BSF 06h, 0
goto Start

OFF:
BCF 06h, 0
goto Start

END
```

2) Exercise #2. BCD to Common Anode 7-Segment Decoder

```
;======================================;
; Author: Charmbix                     ;
; Version: 3.0                         ;
; Course: LBYEC3F - EK1                ;
; Title: Experiment 2: Exercise #2     ;
;======================================;

; Initialization
BSF 03h, 5
MOVLW 0x04
MOVWF 85h
MOVLW 0x4A
MOVWF 86h
BCF 03h, 5

; Outputs
ZERO:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x4B
MOVWF 06h
    goto START

ONE:
MOVLW 0x1F
MOVWF 05h
MOVLW 0xCF
MOVWF 06h
    goto START

TWO:
MOVLW 0x1D
MOVWF 05h
MOVLW 0x6A
MOVWF 06h
```

```
    goto START

THREE:
MOVLW 0x1D
MOVWF 05h
MOVLW 0xCA
MOVWF 06h
    goto START

FOUR:
MOVLW 0x1E
MOVWF 05h
MOVLW 0xCE
MOVWF 06h
    goto START

FIVE:
MOVLW 0x1C
MOVWF 05h
MOVLW 0xDA
MOVWF 06h
    goto START

SIX:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x5A
MOVWF 06h
    goto START

SEVEN:
MOVLW 0x1D
MOVWF 05h
MOVLW 0xCF
MOVWF 06h
    goto START

```

```
EIGHT:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x4A
MOVWF 06h
    goto START

NINE:
MOVLW 0x1C
MOVWF 05h
MOVLW 0xCA
MOVWF 06h
    goto START

A:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x4E
MOVWF 06h
    goto START

Bseg:
MOVLW 0x1E
MOVWF 05h
MOVLW 0x5A
MOVWF 06h
    goto START

C:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x7B
MOVWF 06h
    goto START

D:
MOVLW 0x1F
```

```
MOVWF 05h
MOVLW 0x4A
MOVWF 06h
    goto START

E:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x7A
MOVWF 06h
    goto START

F:
MOVLW 0x1C
MOVWF 05h
MOVLW 0x7E
MOVWF 06h
    goto START

; Inputs
START:
BTFSC 06h, 1
    goto w_ON ; 1 _ _ _
    goto w_OFF ; 0 _ _ _

w_OFF:
BTFSC 06h, 6
    goto x_ON ; 0 1 _ _
    goto x_OFF ; 0 0 _ _

x_OFF:
BTFSC 06h, 3
    goto y_ON ; 0 0 1 _
    goto y_OFF ; 0 0 0 _

y_OFF:
BTFSC 05h, 2
```

```
    goto z_ON ; 0 0 0 1
    goto z_OFF ; 0 0 0 0

z_OFF:
    goto ZERO ; 0 0 0 0

z_ON:
    goto ONE ; 0 0 0 1

y_ON:
BTFSC 05h, 2
    goto y_ON_z_ON ; 0 0 1 1
    goto y_ON_z_OFF ; 0 0 1 0

y_ON_z_OFF:
    goto TWO ; 0 0 1 0

y_ON_z_ON:
    goto THREE ; 0 0 1 1

x_ON:
BTFSC 06h, 3
    goto x_ON_y_ON ; 0 1 1 _
    goto x_ON_y_OFF ; 0 1 0 _

x_ON_y_OFF:
BTFSC 05h, 2
    goto x_ON_y_OFF_z_ON ; 0 1 0 1
    goto x_ON_y_OFF_z_OFF ; 0 1 0 0

x_ON_y_OFF_z_OFF:
    goto FOUR ; 0 1 0 0

x_ON_y_OFF_z_ON:
    goto FIVE ; 0 1 0 1

x_ON_y_ON:
```

```
BTFSC 05h, 2
    goto x_ON_y_ON_z_ON ; 0 1 1 1
    goto x_ON_y_ON_z_OFF ; 0 1 1 0

x_ON_y_ON_z_OFF:
    goto SIX ; 0 1 1 0

x_ON_y_ON_z_ON:
    goto SEVEN ; 0 1 1 1

w_ON:
BTFSC 06h, 6
    goto w_ON_x_ON ; 1 1 _ _
    goto w_ON_x_OFF ; 1 0 _ _

w_ON_x_OFF:
BTFSC 06h, 3
    goto w_ON_x_OFF_y_ON ; 1 0 1 _
    goto w_ON_x_OFF_y_OFF ; 1 0 0 _

w_ON_x_OFF_y_OFF:
BTFSC 05h, 2
    goto w_ON_x_OFF_y_OFF_z_ON ; 1 0 0 1
    goto w_ON_x_OFF_y_OFF_z_OFF ; 1 0 0 0

w_ON_x_OFF_y_OFF_z_OFF:
    goto EIGHT ; 1 0 0 0

w_ON_x_OFF_y_OFF_z_ON:
    goto NINE ; 1 0 0 1

w_ON_x_OFF_y_ON:
BTFSC 05h, 2
    goto w_ON_x_OFF_y_ON_z_ON ; 1 0 1 1
    goto w_ON_x_OFF_y_ON_z_OFF ; 1 0 1 0

w_ON_x_OFF_y_ON_z_OFF:
```

```
    goto A ; 1 0 1 0

w_ON_x_OFF_y_ON_z_ON:
    goto Bseg ; 1 0 1 1

w_ON_x_ON:
BTFSC 06h, 3
    goto w_ON_x_ON_y_ON ; 1 1 1 _
    goto w_ON_x_ON_y_OFF ; 1 1 0 _

w_ON_x_ON_y_OFF:
BTFSC 05h, 2
    goto w_ON_x_ON_y_OFF_z_ON ; 1 1 0 1
    goto w_ON_x_ON_y_OFF_z_OFF ; 1 1 0 0

w_ON_x_ON_y_OFF_z_OFF:
    goto C ; 1 1 0 0

w_ON_x_ON_y_OFF_z_ON:
    goto D ; 1 1 0 1

w_ON_x_ON_y_ON:
BTFSC 05h, 2
    goto w_ON_x_ON_y_ON_z_ON ; 1 1 1 1
    goto w_ON_x_ON_y_ON_z_OFF ; 1 1 1 0

w_ON_x_ON_y_ON_z_OFF:
    goto E ; 1 1 1 0

w_ON_x_ON_y_ON_z_ON:
    goto F ; 1 1 1 1

END
```