More Assembly language implementation

This lab will get you to implement further uses of the Assembly language by introducing you to some arithmetic, logical and branching instructions. Below is a link to a website that provides some good references to the PIC18F instructions:

http://technology.niagarac.on.ca/staff/mboldin/18F_Instruction_Set/

PART A)

The first part is to implement a basic program to input a number from DIP switches, take its 1's complement and display the result out to the LEDs:

C Code:

```
void main()
{
char Input_A;
char Result;

        ADCON1 = 0x0f;
        TRISA = 0x??            // make sure PORT A is input
        TRISB = 0x??;           // make sure PORT B is output
        TRISC = 0x??;           // make sure PORT C is input
        TRISD = 0x??;           // make sure PORT D is output
        TRISE = 0x??;           // make sure PORT E is output

        while (1)
        {
                Input_A = PORTA;
                Input_A = Input_A & 0x0f;
                Result = (1's) Input_A;
                PORTB = Result;
        }
}
```

Use the lab3 Part A) as a baseline for the implementation of this part. Modify it to add the following handling:

1) Declare the two variables 'Input_A' and 'Result' as two memory locations.

2) Read the content of PORTA, mask in the lower 4-bit and store the result into the variable Input_A
3) With the value of PORTA from step 2) still remains in the W register, use the 'COMF' instruction to add W with 'Input_A'. Use the option to store back to W instead of memory (COMF Input_A, 0)
4) Next use 'MOVWF' to output to the variable 'Result'.
5) Finally, use 'MOVFF' to get the content of 'Result' into 'PORTB'.

When done, use the 4 switches connected to PORTA to set a number. Observe the result being displayed on the PORTB that should show the 1's complement of the number specified by the DIP switches.

Make sure that the connections of the DIP switches and the LEDs are implemented such a way that the MSB (most significant bit) of the number is on the leftmost side while the LSB is on the rightmost side. Also, don't forget that when a switch is turned ON, this means logic 0 and when it is off, the logic is 1.

PART B)

Repeat the same operation but add the test condition as follows:

```
void main()
{
char Input_A;
char Result;

        ADCON1 = 0x0f;
        TRISA = 0x??              // make sure PORT A is input
        TRISB = 0x??;             // make sure PORT B is output
        TRISC = 0x??;             // make sure PORT C is input
        TRISD = 0x??;             // make sure PORT D is output
        TRISE = 0x??;             // make sure PORT E is output

        while (1)
        {
                Input_A = PORTA;
                Input_A = Input_A & 0x0f;
                Result = (1's) Input_A;
                PORTB = Result;
                if (Zero flag == 1) Set PORTE.bit0 to 1
                else Clear PORTE.bit0 to 0
        }
}
```

In this exercise, we will add another test after the completion of the Complement operation. We need to check if the Zero (Z flag) is set through the use of the instruction BZ. If Z flag is 1, BZ will force a jump to a label where PORTE bit 0 is set to 1. If Z flag is 0, the instruction just below the BZ instruction will be executed. There clear PORTE bit 0 to be 0. When done go back to the main loop using the 'GOTO MAIN_LOOP' code.

To set a bit 'x' of a PORTy, you will use the instruction 'BSF PORTy,x'. To clear a bit 'x' of a PORTy, use 'BCF PORTy,x'.

The example below will show a typical implementation:

```
        BZ      LABEL1
        (place instruction here to clear PORTE bit 0 to 0)
        GOTO LABEL2
LABEL1:
        (Place instruction here to set PORTE bit 0 to 1)
LABEL2:
        GOTO MAIN_LOOP
```

When done, implement, compile and test the code on the board. Input a number such that the result displays 0 and the Z flag LED is turned on.

PART C)

We will now implement the new operation to add two numbers. Copy the routine developed in part B). Add codes to read a second input from PORTC and stored it into the variable 'Input_C'. Next, perform an addition between the two inputs 'Input_A' and 'Input_C' and stored the result into 'Result'. Also, display the result into PORTB.

```
void main()
{
char Input_A;
char Input_C;
char Result;

        ADCON1 = 0x0f;
        TRISA = 0x??                // make sure PORT A is input
        TRISB = 0x??;               // make sure PORT B is output
        TRISC = 0x??;               // make sure PORT C is input
        TRISE = 0x??;               // make sure PORT E is output

        while (1)
        {
                Input_A = PORTA;
                Input_A = Input_A & 0x0f;
                Input_C = PORTC;
                Input_C = Input_C & 0x0f;
                Result = Input_A + Input_C;
                PORTB = Result;
                if (Zero flag == 1) Set PORTE.bit0 to 1
                else Clear PORTE.bit0 to 0
        }
```

}

Use the instruction ‘ADDWF f,0’ where f is the memory location that has the value to add to the register W.

When completed, set two numbers for inputs and check the result shown on the 5 LEDs connected to PORTB. The fifth LED of PORTB will show the overflow of the result of the addition of two 4-bit numbers.

PART D)

Replace the ADD operation on part C) by doing the ‘AND’ operation with the instruction ‘ANDWF f,0’. Verify that the operation is implemented properly.

PART E)

Replace the ADD operation on part C) by doing the ‘XOR’ operation with the instruction ‘XORWF f,0’. Verify that the operation is implemented properly.

PART F)

One last routine is to take a 4-bit input number and convert into a BCD number which is the decimal equivalent of the input number.

To do the conversion, the input number is checked against the value 0x09. If it is greater than 9, then add a constant 0x06 to it. If it is less than 9, then no addition of the constant is needed. For example:

a) If input = 0x08, then output = 0x08 (no change)
b) If input = 0x0b, then output = 0x0b + 0x06 = 0x11. 0x0b has the decimal equivalent of 11
c) If input = 0x0d, then output = 0x0d + 0x06 = 0x14 because 0x0d is 14 in decimal.

To implement the operation, here are some steps:

a) Read the input into the variable ‘Input_A’
b) Load a constant 0x09 into W
c) Use the instruction CPFSGT (see reference) to compare the value in ‘Input_A’ against the W register (that contains 0x09). If ‘‘Input_A’is greater than 0x09, the next instruction below this instruction is executed. Otherwise, the next instruction is skipped:

```
CPFSLT        Input_A, 1
(go here if greater or =)
(go here if less)
```

PART G)

Take the basic code of each of the five functions implemented above and group them into five different sets of code. Call each group by the name of the function it performs like:

```
SUBROUTINE_COMP:
SUBROUTINE_ADD:
SUBROUTINE_AND:
SUBROUTINE_XOR:
SUBROUTINE_BCD:
```

At the end of each group where the instruction 'GOTO MAIN_LOOP' is called, replace that line by the line 'RETURN'.

Here is a typical implementation:

```
SUBROUTINE_ COMP:

        (code from the COMP implementation)

        RETURN


SUBROUTINE_ ADD:

        (code from the ADD implementation)

        RETURN
```

Next, start at the beginning of the program with a basic loop that will constantly check three new switches connected to PORTD bits 2 and 0. These three switches will select what function to execute as follows:

PORT D			**Action**
Bit_2	Bit_1	Bit_0	
0	0	0	1's complement
0	0	1	ADD operation
0	1	0	AND operation

0	1	1	XOR operation
1	0	0	BCD conversion

Use the ‘BTFSC’ instructions to do the decoding of the five tasks to jump to. Once the differentiation is done, we will have five different labels, each for each task. At each task, first use the BCF and BSF to set the three bits 3-5 of the PORTD to show what routine is being executed. For example, task ‘001’ is for the ‘ADD’ function. The LEDs connected to PORTD bits 3-5 should also show the value ‘001’. Next, use the ‘CALL’ instruction to call the respective subroutine that was created for each task. It will force the execution of the appropriate routine for that task. After the ‘CALL’ was executed, a GOTO MAIN_LOOP instruction should be called in order to go back to the main loop. Here is a typical implementation:

```
MAIN_LOOP:
        BTFSC       PORTD, 2
        GOTO        PORTD2EQ1
        GOTO        PORTD2EQ0

PORTD2EQ1:
        GOTO        TASK_COMP

PORTD2EQ0:
        BTFSC       PORTD, 1
        GOTO        PORTD21EQ01
        GOTO        PORTD21EQ00

PORTD21EQ01:
        BTFSC       PORTD, 0
        GOTO
        GOTO
        ….

TASK_COMP:
        BCF         PORTD, 5
        BCF         PORTD, 4
        BSF         PORTD, 3
        CALL        SUBROUTINE_COMP
        GOTO        MAIN_LOOP

TASK_ADD:
        BCF         PORTD, 5
        BSF         PORTD, 4
        BCF         PORTD, 3
        CALL        SUBROUTINE_ADD
        GOTO        MAIN_LOOP
```

…

Note: PORTD has a mixture of inputs and outputs. The first three bits are used as inputs to read in the operation to be executed. The next three bits are setup as outputs to show what function is being executed. Make sure that the TRISD register is setup properly in order to reflect this condition.