CALIFORNIA STATE POLYTECHNIC UNIVERSITY, POMONA
COLLEGE OF ENGINEERING

ECE 3301L Spring 2024 Session 1 Microcontroller Lab **Felix Pinai**

LAB4
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 InA;
char Result;

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

        while (1)
        {
                InA     = PORTA;                // Read from PORT A
                InA     = InA & 0x0F;           // Mask out upper 4 bits
                Result  = (1's) InA;            // Take the 1's complement for the lower 4 bits
                Result  = Result & 0x0F;        // Mask out the upper 4 bits
                PORTC = 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 'InA' and 'Result' as two memory locations.

2) Start the program with the statement ORG 0 and add the label START:
3) Make sure to properly initialize the register ADCON1 with the correct value.
4) Based on the provided schematics, make sure to program the direction of the all ports used on the design. PORTA, PORTB and PORTE as all inputs while PORTC and PORTD as all outputs. Use the registers TRISA, TRISB, TRISC, TRISD and TRISE and set the correct value.
5) Read the content of PORTA using the instruction MOVF and the option ',W' to store the value into the W register.
6) Use the instruction ANLW to mask out the upper 4-bit of the W register.
7) Store the content of W into the variable **InA** with the instruction MOVWF.
8) use the 'COMF' instruction to do the complement of the variable '**InA**'. Use the option to store back to W instead of memory (COMF **InA**, W)
9) Now with W being the complement of 'InA', perform a masking operation to mask off the upper 4-bit of W. Use the instruction ANDLW instruction
10) Next use 'MOVWF' to output to the variable 'Result'.
11) Finally, use 'MOVFF' to copy the content of 'Result' into 'PORTC'.
12) Add an instruction to go back to the MAIN_LOOP
13) Don't forget to have the 'END' at the end of the program

```
#include <P18F4620.inc>
config OSC = INTIO67
config WDT = OFF
config LVP = OFF
config BOREN = OFF


InA      equ   0x20
InB      equ   0x21
Result   equ   0x22

   ORG         0x0000

START:
         MOVLW   0x0F            ;
         MOVWF   ADCON1          ; ADCON1 = 0x0F
         Add code here to initialize the TRISx registers

MAIN_LOOP:

Add more codes here
```

When done, compile the program and run it. Use the 4 switches connected to PORTA to set a number. Observe the result being displayed on the PORTC 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. I will check the arrangements of the DIP switches before any demo can be performed. If the orientations are not correct, your team will be asked to change the wirings until they are correct. Remember that this is the convention for all digital numberings and it is a must that the team understands that standard.

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 InA;
char Result;

        ADCON1 = 0x0F;
        TRISA = 0x??;            // make sure PORT A is input
        TRISB = 0x??;            // make sure PORT B is input
        TRISC = 0x??;            // make sure PORT C is output
        TRISD = 0x??;            // make sure PORT D is output
        TRISE = 0x??;            // make sure PORT E is input

        while (1)
        {
                InA     = PORTA;             // Read from PORT A
                InA     = InA & 0x0F;        // Mask out upper 4 bits
                Result  = (1's) InA;         // Take the 1's complement for the lower 4 bits
                Result  = Result & 0x0F;     // Mask out the upper 4 bits
                PORTC   = Result;
                if (Zero flag == 1) Set PORTD.bit0 to 1
                else Clear PORTD.bit0 to 0

        }
}
```

In this exercise, we will add another test after the Result of the Complement operation is executed.

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 PORTD bit 0 is set to 1 through the use of the instruction BSF. If Z flag is 0, the instruction just below the BZ instruction will be executed. There clear PORTD bit 0 to be 0 (use BCF instruction). 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’ where ‘x’ specify the bit location and ‘y’ indicates the port to use. To clear a bit ‘x’ of a PORTy, use ‘BCF PORTy,x’.

The example below will show a typical implementation:

```
        BZ      LABEL1                  ; if Z flag is set, branch to LABEL1
        GOTO LABEL2                     ; else branch to LABEL2

LABEL1:                                 ; this is where Z is set
        (place instruction here to set PORTD bit 0 to 1)
        GOTO TEST_DONE1

LABEL2:                                 ; this is where is Z is not set
        (place instruction here to clear PORTD bit 0 to 1)
        GOTO TEST_DONE1
```

Or this is an alternative shorter way:

```
        (place instruction here to clear PORTD bit 0 to 0)
        BZ      TEST_DONE1
        (place instruction here to set PORTD bit 0 to 1)
TEST_DONE1:
```

When done, implement, compile and test the code on the board. Input a number so that the result will display 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 now from PORTB and stored it into the variable ‘In_B’. Next, perform an addition between the two inputs ‘In_A’ and ‘In_B’ and stored the result into ‘Result’. Also, display the result into PORTC.

```
void main()
{
char InA;
char In_B;
char Result;

        ADCON1 = 0x0F;
        TRISA = 0x??;           // make sure PORT A is input
        TRISB = 0x??;           // make sure PORT B is input
        TRISC = 0x??;           // make sure PORT C is output
        TRISD = 0x??;           // make sure PORT D is output
        TRISE = 0x??;           // make sure PORT E is input
```

```
    while (1)
    {
        InA     = PORTA;            // Read from PORT A
        InA     = InA & 0x0F;       // Mask out upper 4 bits
        InB     = PORTB;
        InB     = InB & 0x0f;
        Result  = InA + InB;
        PORTC   = Result;

        if (Zero flag == 1) Set PORTD.bit0 to 1
        else Clear PORTD.bit0 to 0
    }
}
```

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

To prepare for the other parts below, the label 'TEST_DONE1' from part B should be renamed as 'TEST_DONE2' because we are going to re-use all codes and it would be good to have different labels for each operation.

When completed, set two numbers for inputs and check the result shown on the 5 LEDs connected to PORTC. The fifth LED of PORTC 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,W'.

Verify that the operation is implemented properly.

PART E)

Replace the ADD operation on part C) by doing the 'OR' operation with the instruction 'IORWF f,W'.

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.
d)

To implement the operation, here are some steps:

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

```
CPFSGT      InA, 1
 (go here if less or =)
(go here if greater)
```

d) If greater than 9, add 6 to W and then done
e) If less or =, then do nothing.

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:

SUB_COMP:
SUB_ADD:
SUB_AND:
SUB_OR:
SUB_BCD:

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

Important note: In each new subroutine just created, there is a test for the 'Z' flag with the associated labels. Since the same test is performed in each subroutine, those

labels will be duplicated. To avoid errors, those labels must be enumerated since you might end up with branching conditions that are too far.

Here is a typical implementation:

SUB_ COMP:

(code from the COMP implementation)

RETURN

SUB_ 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 PORTE bits 2 and 0. These three switches will select what function to execute as follows:

PORT E			**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	OR 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 4-6 of the PORTD to show what routine is being executed. For example, task '001' is for the 'ADD' function. The RGB LED connected to PORTD bits 4-6 should also show the value '001' (equivalent to the RED color). 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       PORTE, 2            ; skip next line if PORT E bit 2 is clear (0)
        GOTO        PORTE2_EQ1
        GOTO        PORTE2_EQ0

PORTE2_EQ1:
        GOTO        TASK_BCD

PORTE2_EQ0:
        BTFSC       PORTE, 1
        GOTO        PORTE21_EQ01
        GOTO        PORTE21_EQ00

PORTE21_EQ01:
        BTFSC       PORTE, 0
        GOTO        (find the routine that is associated with PORTE bit being 1)
        GOTO        (find the routine that is associated with PORTE bit being 0)
        ….

TASK_COMP:
        BCF         PORTD, 6    ; This is to clear the Blue LED of the RGB
        BCF         PORTD, 5    ; This is to clear the Green LED of the RGB
        BCF         PORTD, 4    ; This is to clear the LED LED of the RGB
        CALL        SUB_COMP
        GOTO        MAIN_LOOP

TASK_ADD:
        BCF         PORTD, 6
        BCF         PORTD, 5
        BSF         PORTD, 4
        CALL        SUB_ADD
        GOTO        MAIN_LOOP
```

(add the other tasks here)

When the entire implementation is completed, set the three switches on PORTE to select a logical/arithmetic function. Depending on the function selected, enter either the value of one the operand(s) and check whether the result is correct. Also, check the logic state of the Z flag.