You may NOT use a calculator. Assume the following memory/register contents at the beginning of each instruction:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x035</td>
<td>0x7A</td>
</tr>
<tr>
<td>0x036</td>
<td>0x4B</td>
</tr>
<tr>
<td>0x037</td>
<td>0xFC</td>
</tr>
<tr>
<td>0x038</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

a. (3 pts) In the code below, give the FINAL values of FSR0, FSR1 and the final value of any changed memory locations after the instruction sequence is executed.

```
lfsr    FSR0, 0x38
lfsr    FSR1, 0x35
movff   PREDEC0, POSTINC1
```

First, there is no PREDEC addressing mode available, so my mistake. But assuming there is one available, then PREDEC0 means to pre-decrement FSR0, then access the contents of the memory location pointed to by FSR0. So, decrementing FSR0-- produces 0x037. POSTINC1 means to access the contents of the mem. location pointed to by FSR1, then increment FSR1. So the ‘movff’ instruction does the following move:

```
movff   0x037, 0x035     ; so location [0x037] → 0x035, or 0xFC → 0x035.
```

After the movff, the value of FSR1 is incremented (POSTINC1), so FSR1++ = 0x36.

The final results are:
- mem location 0x035 changed to 0xFC.
- Final value of FSR0 = 0x037.
- Final value of FSR1 = 0x036.

b. (3 pts) In the code below, give the FINAL values of FSR0 and the final value of any changed memory locations after the instruction sequence is executed.

```
lfsr    FSR0, 0x35
movlw   2
movff   PLUSW0, INDF0
```

PLUSW0 means to access the contents of the memory location pointed to by [FSR0 + W]. INDF0 means to access the contents of the mem. location pointed to by FSR0. So the ‘movff’ instruction does the following move:

```
movff   [0x035+W], 0x035    ; so location [0x035 + 2] → 0x035, or [0x037] → 0x035, or 0xFC → 0x035
```

The PLUSW0 mode does NOT change the value of FSR0. The final results are:
- mem location 0x035 changed to 0xFC.
- Final value of FSR0 = 0x035.
- Final value of W is 2.
c. (2 pts) Write the C function below in PIC18 assembly language. Assume that the value of iptr is PASSED into the function within the FSR0 register. What this code does is INCREMENT the value of the int pointed to by the pointer iptr.

    mysub (int *iptr) {
        (*iptr) ++;
    }

A solution:

    movlw    0                           ; WREG = 0 so can do 16-bit increment of [FSR0] + 1
    incf       POSTINC0,f          ; increment the LSByte of the INT pointed to by FSR0
        ; then have POSTINC0 increment FSR0 to point to MSByte
    addwfc   INDF0,f                 ; add 0 + CARRY to MSByte to complete 16-bit increment
    return

A common error:

    incf       INDF0,f          ; increment the value pointed to by FSR0
    return

This would be correct if the subroutine was declared as below, where *iptr is a pointer to a CHAR type (8-bit value) instead of an INT type.

    mysub (char *iptr) {  // 'char' would make the above code correct
        (*iptr) ++;
    }

However, because in the original code iptr is a pointer to an INT type, you need to do a 16-bit increment.

What if a CBLOCK had been used to pass the *iptr value like:

    CBLOCK  0x????
    iptr: 2
    ENDC

In this case, you would have had to write a bit more code in order to initialize FSR0 from the iptr contents:

    movff     iptr, FSR0L
    movff    iptr+1, FSR0H          ; FSR0 now contains the pointer passed in the iptr variable.
    movlw    0                           ; WREG = 0 so can do 16-bit increment of [FSR0] + 1
    incf       POSTINC0,f          ; then have POSTINC0 increment FSR0 to point to MSByte
    addwfc   INDF0,f                 ; add 0 + CARRY flag to MSByte to complete 16-bit increment
    return
What if the C code had been:

```c
mysub (int *iptr) {
    (*iptr) --;
}
```

This does a decrement on the INT pointed to by iptr. In this case, the solution would have been:

```assembly
movlw    0                           ; WREG = 0 so can do 16-bit decrement of [FSR0] + 1
decf     POSTINC0,f          ; decrement the LSByte of the INT pointed to by FSR0
        ; then have POSTINC0 increment FSR0 to point to MSByte
subwfb   INDF0,f                 ; subtract 0 + BORROW from MSByte to complete 16-bit decrement
return
```

What if the C code had been:

```c
mysub (unsigned int *iptr) {
    (*iptr) = (*iptr) >> 1;  // right shift this unsigned INT value
}
```

This does a right shift by 1 on the unsigned INT pointed to by iptr. In this case, a solution would have been:

```assembly
movf     POSTINC0,f         ; increment FSR0 to point to the MSB of *iptr since FSR0 is pointing to the
        ; LSByte because of the little endian storage of INTs. A rotate right must
        ; first shift the MSByte, then the LSByte
        bcf     STATUS,C           ; clear carry
        rrcf    POSTDEC0, f       ; rotate right MSByte, then decrement FSR0 to point to LSByte
        rrcf    INDF0,f                 ; rotate right LSByte
return
```