You may NOT use a calculator. You may use only the provided reference materials. If a binary result is required, give the value in HEX. Assume all variables are in the first 128 locations of bank 0 (access bank) unless stated otherwise. For any signed right shifts, assume that the sign bit is preserved.

Part I: (82 points)

a. (4 points) Write a PIC18 assembly language code fragment to implement the following.

```assembly
signed int i;
// do not worry about sign since it a left shift
i = i << 1;
bcf STATUS, C // shift in a zero
rlcf i, f // Shift LSB,
rlcf i+1, i // then MSB
```

b. (6 points) Write a PIC18 assembly code fragment to implement the following. The code of the loop body has been left intentionally blank; I am only interested in the comparison test. For the loop body code, just use a couple of dummy instructions so I can see the start/begin of the loop body.

```assembly
unsigned long i; // THIS IS A LONG!!!!!!!!!!!!!!!
do {
    ...operation 1...
    ...operation 2...
}while (i != 0);
loop_top:
    ; ...operation 1...
    ; ...operation 2...
    // now test i
    movf i, w // four bytes since i is long
    iorwf i+1, w
    iorwf i+2, w
    iorwf i+3, w
    bnz loop_top // i nonzero, loop back
end_while:
```
c. (8 points) Write a PIC18 assembly code fragment to implement the following. The code of the loop body has been left intentionally blank; I am only interested in the comparison test. For the loop body code, just use a couple of dummy instructions so I can see the start/begin of the loop body.

```assembly
; start of loop
loop_top:
    movf i, w ; operation 1...
    subwf k, w ; operation 2...
    bov L1
    bn loop_body ; if true, loop body
    bra loop_exit ; exit

L1:
    bn loop_exit ; exit (FALSE cond)!
    (exit on V=1, N=1)

loop_body:
    ...code for operation 1...
    ...code for operation 2...
    bra loop_top

loop_exit:
    ...rest of code...
```

d. (8 points) Implement the `strswap()` function given below. Assume FSR0 already contains the pointer value for `char *strA` on function entry but that the pointer value for `char *strB` is passed in the CBLOCK. In the subroutine, you can use either FSR1 or FSR2 to implement the pointer operations for `char *strB`.

```c
#include <asm/psw.h>

void strswap(unsigned char* strA, unsigned char* strB, unsigned char length)
{
    char tmp;
    while (length)
    {
        tmp = *strB; // save strB value
        *strB = *strA; // replace strB value
        *strA = tmp; // replace strA value
        strA++; // next strA location
        strB++; // next strB location
        length--;
    }
}
```

```assembly
; Parameter block for the strswap function
CBLOCK 0x040
    length, strB:2, tmp  ; Space for parameters
ENDC

strwap:
    movff strB, FSR1L // Copy *strB ptr
    movff strB+1, FSR1H // to FSR1
while_top:
    movf length, f
    bz end_while
    movff INDF1,tmp // copy *strB to tmp
    movff INDF0,POSTINC1 // copy *strA to *strB, strB++
    movff tmp,POSTINC0 // copy tmp to *strA, strA++
    decf length, f
    bz while_top
end_while:
    return
```
e. (8 points) Implement the main() code below in PIC assembly. Pass the value for “char *strA” directly in FSR0. Pass the value for “char *strB” and “char length” using the CBLOCK space for “strswap”.

```assembly
void strswap(unsigned char* strA, unsigned char* strB, unsigned char length)
{
    // some code
}
char *s1[]="Hello!";
char *s2[]="olleh!"
main()
{
    strswap(&s1[0], &s2[0], 6);
}
```

```
lfsr FSR0, s1 ; Set up strA=&s1[0] in FSR0
movlw low s2 ; Set up ptrb=&k
movwf strB
movlwf strB
movlw high s2
movwf strB+1 ;copy address of &s2[0] to strB
movlw 6 ;w = 6
movwf length ;length = 6
rcall strswap ;call subroutine
```

f. (6 points) Write a PIC18 assembly code fragment to implement the following. The code of the if{} body has been left intentionally blank; I am only interested in the comparison test. For the if{} body code, just use a couple of dummy instructions so I can see the start/begin of the if{} body.

```assembly
signed int i, j;
if (i == j)
{
    ...operation 1...
    ...operation 2...
}
```

```
movf i, w
subwf j, w ; if low i != low j,
bnz end_if ;then skip if_body
movf i+1, w
subwf j+1, w ; if high i !=high j,
bnz end_if ;then skip if_body
if_body:
    ...code for operation 1...
    ...code for operation 2...
end_if:
```
(6 pts) Starting at instruction “Start:”, fill in the table with the order in which instructions are executed (give the label and instruction as shown, the first instruction is filled in).

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Start</td>
<td>call subA</td>
</tr>
<tr>
<td>2: SubA</td>
<td>nop</td>
</tr>
<tr>
<td>3: SubA1</td>
<td>goto subB</td>
</tr>
<tr>
<td>4: SubB</td>
<td>nop</td>
</tr>
<tr>
<td>5: SubB1</td>
<td>return</td>
</tr>
<tr>
<td>6: Start1</td>
<td>nop</td>
</tr>
<tr>
<td>7: Start2</td>
<td>nop</td>
</tr>
</tbody>
</table>

This return statement is never executed.

Returns to Start1!!

Diagram:

```
Start:     call subA
Start1:    nop
Start2:    nop

subA:      nop
subA1:     goto subB
subA2:     return

subB:      nop
subB1:     return
```

This return statement is never executed.
h. (20 points) After the execution of ALL of the C code below, fill in the memory location values. Assume little-endian order for multi-byte values.

```c
CBLOCK 0x020
ptra:2, ptrb:2, b:4, a:4, c:4, ptrc:2
ENDC

long *ptra;
int *ptrb;
signed int b;
signed long a;
char c[4];
char *ptrc;

a = -10;   // Note: value given in decimal
b = a >> 1;
ptra = &a;
ptrb = &b;
ptra = ptra + 2;
ptrc = &c[3];
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents (MUST be given in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0020</td>
<td>0x2E</td>
</tr>
<tr>
<td>0x0021</td>
<td>0x00</td>
</tr>
<tr>
<td>0x0022</td>
<td>0x24</td>
</tr>
<tr>
<td>0x0023</td>
<td>0x00</td>
</tr>
<tr>
<td>0x0024</td>
<td>0xFB</td>
</tr>
<tr>
<td>0x0025</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0026</td>
<td>0xF6</td>
</tr>
<tr>
<td>0x0027</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0028</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0029</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x002A</td>
<td>0x??</td>
</tr>
<tr>
<td>0x002B</td>
<td>0x??</td>
</tr>
<tr>
<td>0x002C</td>
<td>0x??</td>
</tr>
<tr>
<td>0x002D</td>
<td>0x??</td>
</tr>
<tr>
<td>0x002E</td>
<td>0x2D</td>
</tr>
<tr>
<td>0x002F</td>
<td>0x00??</td>
</tr>
</tbody>
</table>

- `ptrA = &a = 0x0026
  - `ptrA = ptra+2 = 0x26 + 2*4 = 0x26+8 = 0x2E
    - (ptrA is pointer to long, so 2*sizeof(long)=2*4

- `ptrB = &b = 0x0024
  - `b = -10 >> 1 = -10/2 = -5
    - -5 = 0 – (+5) = 0x00 – 0x05 = 0xFB
      - signextend to 16-bits, so b = 0xFFFB

- `a = -10
  - -10 = 0 – (+10) = 0x00 – 0x0A = 0xF6
    - signextend to 16-bits, so a = 0xFFF6

- `c[0]`:
  - `c[1]`:
  - `c[2]`:
  - `c[3]`:

- `ptrc = &c[3] = 0x002D`
For each of the following problems, give the FINAL contents of changed registers or memory locations. Give me the actual ADDRESSES for a changed memory location (e.g. Location 0x0100 = 0x??). Assume these memory/register contents at the BEGINNING of EACH problem.

W register = 0x02

Memory:
0x0100  0x03  
0x0101  0x01  
0x0102  0xB2  
0x0103  0xA5  
0x0104  0xF2

i. (4 points)
lfsr FSR1, 0x0102
movff PLUSW1, 0x0100

FSR1 = ____0x102_____
Location 0x0100 = 0xF2

j. (4 points)
lfsr FSR1, 0x0100
movff PREINC1, 0x0104

FSR1 = 0x101_____
Location 0x104 = 0x01

k. (4 points)
lfsr FSR1, 0x0104
movff 0x0100, POSTDEC1

FSR1 = 0x103_____
Location 0x104 = 0x03

l. (4 points)
movff 0x100,FSR1L
movff 0x101,FSR1H
movff POSTINC1, 0x0102

FSR1 = 0x104_____
Location 0x102 = 0xA5
Part II: (18 points) Answer 6 of the next 8 questions. Cross out the 2 question you do not want graded. Each question is worth 3 points.

a. Why are the FSR0, FSR1, FSR2 registers 12-bits long? Be explicit.

Because the FSRx registers hold addresses that point into data memory, which contains 4096 locations ($2^{12}$), thus you need 12 bits to specify the address.

b. What return address is pushed on the stack for the following code?

0x0204  call  0x100

The call instruction is 4 bytes long, so the return address PC+4 = 0x204+4 = 0x208

c. Write an addition of two 2’s complement 8-bit numbers that will produce the following flag conditions: $V = 1$, $N = 1$, $C = 0$, $Z = 0$.

$0x7F + 0x01 = 0x80$ (positive+positive = negative, overflow! $V=$, $N=1$, $C=0$, $Z=0$)

d. Give the machine code for the following instruction:

here: bra here

Branch offset (N)computed as

$$\text{target location} = \text{PC} + 2 + 2*N$$

The target location is PC, so

$$\text{PC} = \text{PC} + 2 + 2*N$$

$$N = \frac{\text{PC} - (\text{PC} + 2)}{2} = -2/2 = -1 = 0xFF.$$  

The format of the bra instruction is:

1101 0nnn nnnn nnnn where (nnn..nn) is the offset. Since the offset is 0xFF, the nnn bits are all ‘1’s. so

1101 0111 1111 1111 = 0xD7FF is the final answer
e. Write assembly code for the following:

```assembly
long a, b;
a = a - b;
```

```
movf b,w
subwf a,f
movf b+1,w
subwfb a+1,f
movf b+2,w
subwfb a+2,f
movf b+3,w
subwfb a+3,f
```

d. When would I have to use a `goto` instead of a `bra`?

The `bra` instruction can branch locally (within +1023 / -1024 instructions) in program memory while the `goto` instruction can move to any point in program memory. So, you have to use a `goto` if the target location is too far away for a `bra`.

g. When does return address stack overflow occur on the PIC18?

The return address stack only has 31 locations. If you make 32 ‘call’s without a return, the return address stack will overflow.

h. In the code below, the comparison `k > p` is tested by doing `k – p`, and using the `false` case of `C=0 || Z=1` (borrow (k is less than p) or zero (k is equal to p)). However, this test does not work in the code below. Why? Be EXPLICIT, describe cases that it will work and cases that it won’t work.

```c
unsigned int k, p
while (k > p) {
    //loop body
}
```

```
loop_top:
    movf p,w
    subwf k,w ;k-p LSByte
    movf p+1,w
    subwfb k+1,w ;k-p MSByte
    bnc loop_exit ;c=0 exit
    bz loop_exit ;z=1 exit
loop_body
    instr1....
    bra loop_top
loop_exit:
```

The problem with the code above is the Zero test – when the 16-bit subtraction is done, the Zero flag reflects the status of the MSByte subtraction only. Thus if `k = 0x10FF, p = 0x1000`, we have `K > P`, yet the test above will get `Z=1` when the MSB `k (0x10) – MSB p (0x10) = 0x00`, causing the loop body to be skipped, which is incorrect.