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.

Part I: (70 pts)

a. (5 pts) Write a PIC18 assembly code fragment to implement the following.

```assembly
signed int i, k;

i = k >> 1;

bcf STATUS,C ; shift in '0'
movf k+1,f ; test sign bit on int value
bnn skip ; skip if postive
bsf STATUS,C ; negative, shift in '1'
skip
rrcf k+1,w ; shift MSByte
movwf i+1 ; save new MSByte
rrcf k,w ; shift LSBYTE
movwf i ; save new LSBYTE
```

b. (8 pts) 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
int i, k;

if (i != k) {
  ..operation 1...
  ..operation 2....
}

movf i,w ;
subwf k,w ; test if LSBYTEs are equal
bnz if_body ; if not equal, know i!=k so do if_body
movf i+1,w
subwf k+1,w
bz end_if ; skip if both LSBYTEs and MSBYTES are equal
if_body
  ...operation 1...
  ...operation 2....
end_if
```
c. (8 pts) Write a PIC18 assembly code fragment to implement the following:

```assembly
loop_top:
  ...code for operation 1...
  ...code for operation 2....
  movf  _k_,w
  subwf  _j__,w    ;do j-k
  bov
  L1
  bn
  loop_top  ;true loop top
  bra  loop_exit ;exit

L1
  bnn
  loop_top  ;true loop top

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

```
signed char j, k;

  do{
    operation 1...
    operation 2...
  }while(k > j)

For k > j, do j – k.
If k > j is true, then j – k will be a negative number (N=1, V=0). If
overflow occurs, then number will be positive (N=0, V = 1).
```

d. (8 pts) Implement the doadd subroutine in PIC18 assembly language. Assume the
parameters have been initialized by the calling function. Do NOT forget that this is a
subroutine!!!!!!

```c
// doadd function
doadd (unsigned int *ptra, unsigned int *ptrb){
  *ptra = *ptra + *ptrb;
}
```

```assembly
;solution A, goes with ;solution A, problem E
;uses ptra, ptrb parameters
  movff ptra, FSR0L
  movff ptra+1, FSR0H
  movff ptrb, FSR1L
  movff ptrb+1,FSR1H
  movf  POSTINC1,w
  addwf POSTINC0,f    ;add LSBYTE
  movf  POSTINC1,w
  addwfc POSTINC0,f ;add MSByte
  return

;solution B, goes with ;solution B, problem E
;ignores ptra, ptrb parameters
;assumes calling routine inits
;FSR0 and FSR1
  movf  POSTINC1,w
  addwf POSTINC0,f    ;add LSBYTE
  movf  POSTINC1,w
  addwfc POSTINC0,f ;add MSByte
  return
```
e. (8 pts) Implement the following in PIC18 assembly, which is a call to the subroutine `doadd` of the previous problem. The assembly code should work regardless of where the parameter block for main is located. The `&p` and `&q` passes the addresses of variables `p` and `q` to the `doadd` subroutine (these are the `*ptra`, `*ptrb` parameters).

```assembly
main() {
    int p, q;
    //call function
    doadd(&p, &q);
}
```

f. (8 pts) 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
int i, k;

if (i || k) {
    ..operation 1...
    ..operation 2...
}
```
g. (5 pts) Write a PIC18 assembly code fragment to implement the following:

```c
signed int s, p, q;

s = p - q;
```

```assembly
movf    q, w        ;
subwf   p, w        ; w = p - q, LSByte
movwf   s           ; save result
movf    q+1, w      ; w = p - q, MSByte, subtract with borrow
subwfb  p+1, w      ; w = p - q, MSByte, subtract with borrow
movwf   s+1         ; save result
```
Assume the following memory contents at the START of EACH of these code fragments for problems g to h.

```
CBLOCK 0x015A
s:1, p:1, q:1,            ; char s,p,q;
r:2,                         ; unsigned int r;
t:4                           ;unsigned long t
ENDC
```

Assume the following initializations:

- `s = 0x39;`
- `p = 0x5A;`
- `q = 0xA5;`
- `r = 0x3044;` (this will be stored in little ENDIAN order!!)
- `t = 0xA5DC39FF;` (this will be stored in little ENDIAN order!!)

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 0x15B = 0x??)

**h. (5 pts)**

```
lfsr FSR1, s
movff PLUSW1,p
```

- FSR1 = ___0x15A___ (plusW1 does not change FSR1)
- Location ___0x15B___ = ____0xA5____

FSR1 initialized to `s` address of 0x15A.
The movff moves 0x15A+2 contents to location p, 0x15B. So contents of 0x15C copied to location 0x15B. FSR1 unchanged at 0x15A.

**i. (5 pts)**

```
movlw low q
movwf FSR1L
movlw high q
movwf FSR1H
movff POSTDEC1, s
```

- FSR1 = ___0x15B____
- Location ___0x15A____ = __0xA5____

FSR1 initialized to `q` address, 0x15C.
movff moves 0x15C to `s` address (0x15A). So 0x15A changed to 0xA5.
FSR0 is then decremented to 0x15B.

**j. (5 pts)**

```
movff r+1, r
```

- Location ___0x015D_ = ___0x30___
Copy contents of location 0x015E to location 0x15D.

**k. (5 pts)**

```
lfsr FSR1, t
movff POSTINC1,s
```

- FSR1 = __0x160____
- Location _0x015A__ = __0xFF_

FSR1 initialized to `t` address, 0x15F.
movff moves 0x15F to `s` address (0x15A).
So 0x15A changed to 0xFF.
FSR0 is then incremented to 0x160.
Part II: (30 pts) Answer 10 out of the next 12 questions. Cross out the 2 questions that you do not want graded. Each question is worth 3 pts.

1. What return address is pushed on the stack for the instruction CALL 0x0300 if the location of the call instruction is 0x0154?
   
   new PC = Old PC + 4 (because CALL is 2 instruction words or 4 bytes)
   = 0x0154 + 4 = 0x0158

2. The value 0xED is a two’s complement, 8-bit number. What is the decimal value?
   
   Because MSdigit is > 7, number is negative. Compute magnitude as
   0x00 – 0xED = 0x 13. Magnitude in decimal is 0x13 = 19. Final answer: - 19.

3. Give the value of –6 as a 16-bit two’s complement number.
   
   +6 = 0x06. In 8 bits, 0 – 6 = -6, so 0x00 – 0x06 = 0xFA.
   In 16-bits, sign extend by adding ‘F’ digits. Final answer: 0xFFFFA

4. Give the V, N flag settings after the operation 0x80 + 0x7F.
   
   0x80 + 0x7F = 0xFF. Result is negative as MSbit is ‘1’.
   Negative + Positive cannot produce overflow. Final answer: V = 0, N = 1

5. Give the V, N flag settings after the operation 0x7F + 0x10.
   
   0x7F + 0x10 = 0x8F. Result is negative as MSbit is ‘1’.
   Positive + Positive produces negative, so overflow. Final answer: V = 1, N = 1
6. In the code below, what is the value of \( i \) when the loop is exited? Give the value in HEX!!

```c
signed char i;
i = 0x01;
while (i > 0) {
    i = i << 1;
}
```

Each time through loop, \( i \) is shifted left.
Progression of \( i \) values is 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80.
At this point \( i \) is a negative number, so the signed comparison of while(\( i \) > 0) is false, loop is exited.
Final \( i \) value is 0x80.

7. For the C code and CBLOCK show below, what is the value of \( \text{ptr} \) after the statement ‘\( \text{ptr}++ \)’? Careful, \( \text{ptr} \) is pointer to type \textit{int}.

```c
int *\text{ptr};
char a[4];
int b[4];
\text{ptr} = b;
\text{ptr}++;
```

```plaintext
CBLOCK 0x200
ptr:2, a:4, b:8
```

\text{ptr} address is 0x200.
\text{a} address is 0x200+2 = 0x202.
\text{b} address is 0x202+4 = 0x206

The assignment \( \text{ptr} = \text{b} \) gives \( \text{ptr} \) a value of 0x206. The \text{ptr}++ operation computes 0x206+2 = 0x208 because \( \text{ptr} \) is a pointer to type \textit{int}, which is 2 bytes in size. Final answer: 0x0208

8. Write the CBLOCK that allocates space for the C variables below in a similar manner as done for problem 7.

```c
long *\text{ptr};
char a[4];
long b[4];
\text{ptr} = b;
\text{ptr}++;
```

```plaintext
CBLOCK 0x200
ptr:2, a:4, b:16
```

The size of \( \text{ptr} \) DOES NOT CHANGE even though \( \text{ptr} \) is now points to a \textit{LONG} instead of an \textit{INT}. This is because \( \text{ptr} \) contains an address, which does not change in size. The number of bytes needed for \text{b} is now 4*4 = 16, as each long is 4 bytes in size.

9. Write a simple PIC18 code fragment that will force return address stack underflow.

```c
; assumes there is no previous ‘call’ active
 goto SUBA
 .................
SUBA
 RETURN
```

Execute a RETURN without a corresponding CALL.
10. Give the machine code for the ‘bov 0x208’ instruction below given the locations shown:

<table>
<thead>
<tr>
<th>location</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0200</td>
<td>bov 0x208</td>
</tr>
<tr>
<td>0x0202</td>
<td>???</td>
</tr>
<tr>
<td>0x0204</td>
<td>???</td>
</tr>
<tr>
<td>0x0206</td>
<td>???</td>
</tr>
<tr>
<td>0x0208</td>
<td>incf 0x002,f</td>
</tr>
</tbody>
</table>

OFFSET = (branch target – (PC +2))/2
= (0x0208 - (0x200+2))/2 = (0x208-0x202)/2 = 6/2 = 3 = 0x03

Encoding from datasheet: 1110 0100 nnnn nnnn (nnnn is the 8bit displacement)
Final answer: 0x E403.

11. Write a PIC18 assembly code fragment to implement the following.

```assembly
signed long k,j;

k = k & j;

; a LONG is 4 bytes!!!!!
movf j,w
andwf k,f ;LSByte
movf j+1,w
andwf k+1,w
movf j+2,w
andwf k+2,w
movf j+3,w ;MSByte
andwf k+3,w
```

12. When does a call instruction have to be used instead of an rcall instruction?

When the distance to the CALL target exceeds the 11 bit displacement, which allows +1023 instruction words forward or –1024 instruction words backwards.