ECE 3724 Fall 2005 Test #2 – Jones / Reese

Net ID: _____ (no names, please)

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: (82 points)

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

```
signed int i;

i++;
```

```
movlw    0
incf     i, f
addwfc   i+1, f
```

b. (8 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.

```
int i, k;

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

```
    movf    i, w
    iorwf   i+1, w
    bnz  if_body
    movf    k, w
    iorwf   k+1, w
    bnz  end_if
if_body:
    ; operations
end_if:
```

c. (8 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.

```
signed char i, k;

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

```
loop_top:
     ...code for operation 1...
     ...code for operation 2...

     movf        i    , w
     subwf       k    , w
     bov         L1
     bn          loop_top ; if true, loop top
     bra         loop_exit    ; exit
L1:
     bnn         loop_top ; if true, loop top
loop_exit:
     ...rest of code...
```

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

```
void strrev(unsigned char* in, unsigned char* out, unsigned char length)
{
      out = out + length;
      while (length)
      {
            *out = *in;
            out--;
            in++;
            length--;
      }
}
```

```
; Parameter block for the strrev function
CBLOCK 0x010
  out:2, length   ; Space for “char *out”,
                  ; unsigned char length
                  ; parameters.
ENDC
```

```
     movff  out, FSR1L
     movff  out+1, FSR1H
     movf   length, w
     addwf  FSR1L, f
     movlw  0
     addwfc FSR1H, f   ;out=out+length, 16-bit add!
     movf   length, f
loop:
      bz     loop_end
     movff  POSTINC0, POSTDEC1
     decf   length, f
     bra    loop
loop_end: return
```

e. (8 points) Implement the main() code below in PIC assembly. Pass the value for "`int *ptra`" directly in FSR0. Pass the value for "`long *ptrb`" and "`char c`" using the `CBLOCK` space for "`a_sub`".

```
a_sub(int* ptra, long *ptrb, char c)
{
      // some code
}


main()
{
      char  n;
      int   i;
      long  k;

      // Some code that initializes
      // n, i, k

      a_sub(&i, &k, n);
}
```

```
CBLOCK 0x20       ; param. block
  ptrb:2, c       ;  for a_sub
ENDC


CBLOCK 0x30       ; param. Block for main()
  n:1, i:2, k:4
ENDC
```

```
lfsr  FSR0, i
movlw low k
movwf ptrb
movlw high k
movwf ptrb+1
movff n, c
call  a_sub
```

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. CAREFUL: the variables are LONG data type!!!!!!!!!!

```
signed long i, j;

if (i == j)
{
      ...operation 1...
      ...operation 2...
}
```

```
     movf     i, w
     subwf    j, w
     bnz  end_if
     movf     i+1, w
     subwf    j+1, w
     bnz  end_if
     movf      i+2, w
     subwf    j+2, w
     bnz  end_if
     movf     i+3, w
     subwf    j+3, w
     bnz  end_if
if_body:
       ; operation 1
       ; operation 2
end_if:
```

g. (4 points) Write a PIC18 assembly language code fragment to implement the following. CAREFUL: the variables are LONG data type!!!!!!!!!!

```
signed long a, b;

b = b – a;
```

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

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.

```
unsigned char a;
unsigned char* ptra;
signed long b;
signed long *ptrb;
signed int c;

a = 192;                // Note: value given in decimal
b = a – 195;            // Note: value given in decimal
c = b << 2;             // Note: value given in decimal

ptra = &a;
ptrb = &b;
ptrb = ptrb + 2;
```

```
CBLOCK 0x060
  a, ptra:2, b:4, ptrb:2, c:2
ENDC
```

Location	Contents (**MUST** be given in hex)	
0x0060	0xC0	a (1 byte) = 192 decimal = 0xC0 hex
0x0061	0x60	ptra low byte ptra = 0x0060 (address of a)
0x0062	0x00	ptra high byte
0x0063	0xFD	b lowest byte
0x0064	0xFF	.
0x0065	0xFF	. b = –3 = 0xFFFFFFFD hex
0x0066	0xFF	b highest byte
0x0067	0x6B	ptrb low byte. ptrb = address of b (0x0063) + 2*4 (size of a long)
0x0068	0x00	ptrb high byte = 0x006B
0x0069	0xF4	c low byte c = –3 << 2 = 0xFFF4 hex
0x006A	0xFF	c high byte

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 = 0x04

Memory:

0x0150	0x93
0x0151	0xD9
0x0152	0x3F
0x0153	0x88
0x0154	0xE1

i. (4 points)

```
lfsr  FSR1, 0x0150
movff PLUSW1, 0x0153
```

FSR1 = 0x0150

Location 0x153 = 0xE1

j. (4 points)

```
lfsr  FSR1, 0x0152
movff 0x0154, PREINC1
```

FSR1 = 0x153

Location 0x153 = 0xE1

k. (4 points)

```
lfsr  FSR1, 0x0151
movff POSTINC1, 0x0154
```

FSR1 = 0x152

Location 0x154 = 0xD9

l. (4 points) (somewhat of a trick question; be aware that FSR0L occupies location 0xFE9)

```
lfsr  FSR1, 0x0153
movff INDF1, FSR0L
```

FSR1 = 0x0153

Location 0xFE9 = 0x88

Part II: (18 points) Answer 6 of the next 9 questions. Cross out the 2 question you do not want graded. Each question is worth 3 points.

a. How many bits wide is the FSR0 register? Why is it this particular width? Therefore, how many different memory locations can it access? Why?

The FSR0 register is 12 bits wide, in order to access all $2^{12} = 4096$ data memory locations the PIC contains.

b. In an *n*-bit 2's complement number, what is the largest positive number? What is the smallest negative number? Why is the range of the positive and negative numbers different?

The largest positive number is $+2^{n-1} - 1$. The smallest negative number is -2^{n-1}. The number 0 is taken away from the positive range, leaving it one number small than the negative range.

c. Why is the signed shift right different than an unsigned shift right? Why is a signed shift left the same as an unsigned shift left?

A signed right shift involves shifting a 0 for positive numbers and a 1 for negative number into the most significant bit of the result, while an unsigned shift right always shifts a 0 into the MSB. For both a signed and unsigned left shift, a 0 is always shifted into the least significant bit of the result. If the sign bit changes as a result of the left shift, then overflow occurs, so there is no need to try to maintain the sign bit for a left shift (this is why the signed left and unsigned left are the same).

d. How is a `call` different from an `rcall`? When MUST you use a `call` instead of an `rcall`?

A call is a 4-byte wide instruction which can transfer control to any point within the entire program memory. A relative call (rcall) is a two-byte wide instruction which can only transfer control to instructions within -1024 to +1023 of the rcall.

e. Why do `call` and `rcall` instructions use the stack? (Hint: think about why a `goto` or `bra` instruction cannot be used to implement a subroutine call)?

The return address must be recorded on the stack in order to jump back to the caller after finishing the execution of a subroutine. Using a stack allows call of a subroutine from within a subroutine. A goto or bra instruction does not record the return address, leaving a subroutine unable to return to the proper location.

f. For the comparison P >= Q, the required subtraction operation is P – Q. Give the N,V flag settings for the TRUE case and give two different numerical values (IN HEX!!) for P, Q that show why both flag cases are needed.

V = 0 and N = 0 (example: P = +2, Q = +1, P >= Q,
so P – Q = +2 – (+)1 = 1, this is in the positive range, so V=0, N=0.
Hex P-Q = 0x02 – 0x01 = 0x01

V = 1 and N = 1 (example: P = +127 (0x7F), Q = –1 (0xFF), P >= Q, so
P-Q = +127 – (-1) = +127 + +1 = +128, this is V = 1 as +128 is outside the positive range.
P – Q = 0x7F – 0xFF = 0x80, the MSb = 1, so N = 1. So this is the case V=1, N=1.

g. A 16-bit comparison test (>, >=) requires only a single 16-bit subtraction and a C flag check. However, the equality test (p == q) CANNOT be written as:

```
movf        q, w
subwf       p, w
movf        q+1, w
subwfb      p+1, w      ;;sub with borrow
bz          p_equal_q
```

where the code at p_equal_q is executed if p is equal q. Give numerical values for p, q that proves that this code does not work and explain why.

When P = 0x0001 and Q = 0x0000, Z = 0 after the movf/subwf pair, which indicates that the lower bytes are unequal. However, after the movf/subwfb pair, Z is reset to Z = 1, indicating that the high bytes are equal. The bz incorrectly assumes that if the high bytes are equal, the entire number must be equal and therefore branches incorrectly. A bnz end_if after the movf/subf pair would fix this problem.

h. If the PIC's data memory were expanded to 2^{16} locations, how would the size of the FSR1 register change? Would it require a FSR1U (upper byte), in addition to the pre-existing FSR1H (high byte) and FSR1L (low byte)? Why or why not?

The FSR register must be grown from 12 bits to 16 bits to accommodate this additional data address space. However, 16 bits can be stored in two 8 bit pieces, FSR1L and FSR1H and therefore does not require an FSR1U.