

ECE 3724 Fall 2005 Test #2 – Jones / Reese (Circle one)

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. *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. Caution – *i* is a *signed int* !!!

```
signed int i;
```

```
i = i >> 1;
```

```
bcf STATUS, C // Assume number is unsigned
btfsc i+1, 7 // If signed, then
bsf STATUS, C // shift a 1 in instead of a 0
rrcf i+1, f // Shift MSB,
rrcf i, j // then LSB
```

- b. (8 points) Write a PIC18 assembly code fragment to implement the following. The code of the while{} body has been left intentionally blank; I am only interested in the comparison test. For the while{} body code, just use a couple of dummy instructions so I can see the start/begin of the while{} body.

```
unsigned int i, k;

while (!i && k) {
    ...operation 1...
    ...operation 2...
}
```

```
while_top:
    movf i, w
    iorwf i+1, w
    bnz end_while // if i is true, done
    movf k, w
    iorwf k+1, w
    bz end_while // if k is false, done

    ; ...operation 1...
    ; ...operation 2...

    bra while_top
end_while
```

- c. (8 points) Write a PIC18 assembly code fragment to implement the following. The code of the while{} body has been left intentionally blank; I am only interested in the comparison test. For the while{} body code, just use a couple of dummy instructions so I can see the start/begin of the while{} body.

```
signed char i, k;

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

```
loop_top:
    movf        k    , w
    subwf       i    , w
    bov     L1
    bnn     loop_body    ; if true, loop body
    bra     loop_exit    ; exit
L1:
    bnn     loop_exit    ; exit

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

    bra     loop_top

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

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

```
void replace_chars(unsigned char* source, unsigned char to_find, unsigned
char* replace, unsigned char length)
```

```
{
while (length)
{
if (*source == to_find)
{
*source = *replace;
replace++;
}
source++;
length--;
}
}
```

```
; Parameter block for the replace_chars function
CBLOCK 0x010
replace:2, to_find, length ; Space for
; uchar* replace, to_find,
; length parameters.
ENDC
```

```
replace_chars:
    movff replace, FSR0L // Copy *replace ptr
    movff replace+1, FSR0H // to FSR0
while_top:
    movf length, f
    bz end_while
    movf INDF1, w // get *source
    subwf to_find // compare to to_find
    bnz skip_replace
    movff POSTINC0, INDF1 // *source = *replace,
// replace++
skip_replace:
    movf POSTINC1, w // source++ (shortcut)
    decf length, f
    bra while_top

end_while:
    return
```

- e. (8 points) Implement the main() code below in PIC assembly. Pass the value for “char \*ptrb” directly in FSR0. Pass the value for “int \*ptrb” and “char c” using the CBLOCK space for “a\_sub”.

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

```
main()
{
```

```
    char i, n;
    int k;
```

```
    // Some code that initializes
    // n, i, k
    // Don't include this in your assembly; assume it's done elsewhere
```

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

```
}
```

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

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

```
lfsr FSR0, n        ; Set up ptrb=&n in
FSR0
movlw low k         ; Set up ptrb=&k
movwf ptrb
movlw high k
movwf ptrb+1
movff i, c          ; Set c=i
rcall a_sub         ; 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.

```
signed int i, j;
```

```
if (i != j)
```

```
{
```

```
    ...operation 1...
    ...operation 2...
}
```

```
    movf i, w
    subwf j, w      ; if low i != low j,
    bnz if_body    ; then goto
if_body
    movf i+1, w
    subwf j+1, w   ; if high i == high
j,
    bz end_if     ; then goto end_if

if_body:
    ...code for operation 1...
    ...code for 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; // xor operation
```

```
movf a, w  
xorwf b, f  
movf a+1, w  
xorwf b+1, f  
movf a+2, w  
xorwf b+2, f  
movf a+3, w  
xorwf 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.

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

```
CBLOCK 0x026
    a:4, ptra:2, b:2, ptrb:2, c
ENDC
```

```
c = -37;           // Note: value given in decimal
a = -23;          // Note: value given in decimal
b = a - 15;       // Note: value given in decimal
c = c >> 2;       // Note: value given in decimal
```

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

Location	Contents ( <b><u>MUST</u></b> be given in hex)	
0x0026	_____ 0xE9 _____	}
0x0027	_____ 0xFF _____	
0x0028	_____ 0xFF _____	
0x0029	_____ 0xFF _____	
		<b>a = -23 = 0xFFFFFFFFE9</b>
0x002A	_____ 0x26 _____	}
0x002B	_____ 0x00 _____	
		<b>ptra = 0x0026 (location of a in memory)</b>
0x002C	_____ 0xDA _____	}
0x002D	_____ 0xFF _____	
		<b>b = -23 - 15 = 0xFFDA</b>
0x002E	_____ 0x30 _____	}
0x002F	_____ 0x00 _____	
		<b>ptrb = 0x002C (location of a in memory) + 2*2 = 0x0030</b>
0x0030	_____ 0xF6 _____	}
		<b>c = -37 &gt;&gt; 2 = 0xDB &gt;&gt; 2 = 0xF6</b>

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 = 0x03

Memory:

0x01AD	0x6F
0x01AE	0x2A
0x01AF	0x59
0x01B0	0x0B
0x01B1	0x96

i. (4 points)

```
lfsr FSR1, 0x01AE
movff PLUSW1, 0x01AE
```

FSR1 = 0x01AE

Location 0x01AE = 0x96

j. (4 points)

```
lfsr FSR1, 0x01AF
movff 0x01AD, PREINC1
```

FSR1 = 0x01B0

Location 0x01B0 = 0x6F

k. (4 points)

```
lfsr FSR1, 0x01B0
movff POSTDEC1, 0x01B1
```

FSR1 = 0x01AF

Location 0x01B1 = 0x0B

l. (4 points)

```
lfsr FSR1, 0x01AF
movff INDF1, 0x01B0
```

FSR1 = 0x01AF

Location 0x01B0 = 0x59

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. If a new PIC was designed with 8KB of file registers instead of the 4KB present in the 18F242, how many bits wide should the FSR0 register be? Why?

**13 bits, since  $2^{13} = 8K$  means we need 13 address bits to access all of the 8K of memory.**

- b. What will happen when the following code is executed on the PIC?  
`here: rcall here`

**Stack overflow after 32 executions.**

- c. When doing addition with signed numbers, when does an overflow occur? What flag is used in unsigned numbers which prevents overflow problems?

**Adding a two positive numbers whose sum is greater than the maximum positive number representable causes overflow. Likewise, adding two negative numbers whose sum is smaller than the most negative number representable causes overflow. The carry flag holds the equivalent of an overflow when performing unsigned arithmetic.**

- d. How is a `bra` different from a `goto`? When MUST you 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.**

- e. Why are three registers necessary to determine the address in program memory stored at the top of the stack (TOSU, TOSH, TOSL), when just two registers compose FSR0 (FSR0L, FSR0H)?

**Three registers provide  $3 \cdot 8 = 24$  bits of address, while two registers provide  $2 \cdot 8 = 16$  bits. Program memory is 20 addressable bits wide, so 2 bytes don't provide enough address bits, forcing the use of three bytes. Data memory is 12 bits wide, making 2 bytes sufficient.**

- f. In what cases can signed numbers be correctly compared by examining only the C (carry) flag? When does this fail? Give an example of each case.

**If the two numbers have the same sign, they can be compared using only the C (carry) flag. For example,  $-4 < -3$  since  $-4 - (-3) = -1$  is the same as  $0xFC - 0xFD = 0xFF$  which gives  $C = 0$  and  $N = 1$ . Likewise,  $3 < 4$  since  $3 - 4 = -1$  is the same as  $0x03 - 0x04 = 0xFF$  which gives  $C = 0$  and  $N = 1$ . However,  $-4 < 3$  since  $-4 - 3 = -7$  gives  $N = 1$  but becomes  $0xFC - 0x03 = 0xF9$  produces  $C = 1$  ( $-4 > 3$ , or  $252 > 3$  decimal). Two numbers of opposite sign which causes an overflow can also be successfully compared using only the carry flag. For example,  $-128 < 1$  gives  $N = 0$ ,  $V = 1$ , and  $C = 0$  (less than flags).**

- g. Why do some instructions, such as `goto`, `call`, and `lfsr`, require 4 bytes of program memory when most others require only two bytes?

**The `goto`, `call`, and `lfsr` instructions require 20 bits for a program memory address (`goto` and `call`) or 12 bits for data memory address (`lfsr`). Two-byte instructions provide space for only 8 address bits, drawing the remaining bits from the BSR. Four bytes are necessary to provide space for the required additional address bits.**

- h. The C `long` data type occupies 4 bytes. If a new C data type called `extralong` was defined which occupies 5 bytes, what address does it point to after executing the following code?

```
extralong *ptr1;
ptr1 = 0;           // Make pointer point to address 0x000
ptr1 = ptr1+5;     // What address does it now point it?
```

**The resulting address is (size of `extralong`)\*(number to be added) which is  $5 * 5 = 25$ . Therefore, `ptr` points to address 25 (decimal) or 0x019 (hex).**