

ECE 3724 Summer 2006 Test #2 –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. *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.

```
signed int i;
```

```
i = i << 1;
```

```
//do not worry about sign since it a left shift  
  
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.

```
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.

```
signed char i, k;

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

test:	True	False
k-i	N=1,V=0	N=0,V=0
	or	or
	N=0,V=1	N=1,V=1

```
loop_top:
    movf        i    , w
    subwf      k    , w
    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*.

```
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--;
    }
}
```

```
; 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”.

```
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);
```

```
}
```

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

```
CBLOCK 0x000    ;space for main
s1:7, s2:7
ENDC
```

```
lfsr FSR0, s1    ; Set up strA=&s1[0] in FSR0
movlw low s2     ; Set up ptrb=&k
movwf 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.

```
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:
```

g. (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).

```

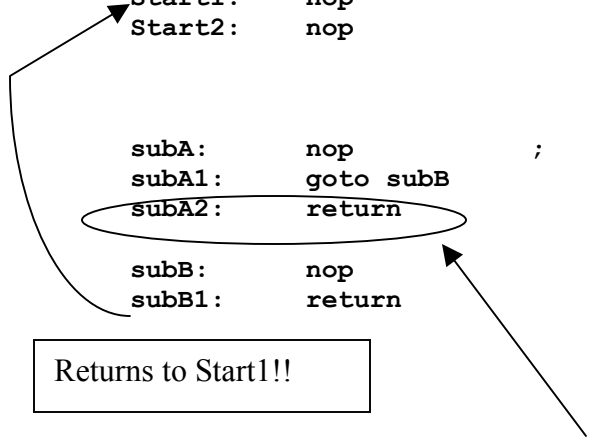
Start:    call subA
Start1:   nop
Start2:   nop

subA:     nop
subA1:    goto subB
subA2:    return
subB:     nop
subB1:    return
    
```

Returns to Start1!!

Label	Instruction
1: Start	call subA
2: SubA	nop
3: SubA1	goto subB
4: SubB	nop
5: SubB1	return
6: Start1	nop
7: Start2	nop

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.

```
long *ptrA;
int *ptrB;
signed int b;
signed long a;
char c[4];
char *ptrC;
```

```
CBLOCK 0x020
 ptrA:2, ptrB:2, b:2, a:4, c:4, ptrC:2
ENDC
```

```
a = -10;           // Note: value given in decimal
b = a >> 1;
ptrA = &a;
ptrB = &b;
ptrA = ptrA + 2;
ptrC = &c[3];
```

Location	Contents (MUST be given in hex)		
0x0020	<u>0x2E</u>	} ptrA = &a = 0x0026 ptrA = ptrA+2 = 0x26 + 2*4=0x26+8 = 0x2E (ptrA is pointer to long, so 2*sizeof(long)=2*4	
0x0021	<u>0x00</u>		
0x0022	<u>0x24</u>	} ptrB = &b = 0x0024	
0x0023	<u>0x00</u>		
0x0024	<u>0xFB</u>	} b = -10 >> 1 = -10/2 = -5 -5 = 0 - (+5) = 0x00 - 0x05 = 0xFB signextend to 16-bits, so b = 0xFFFF	
0x0025	<u>0xFF</u>		
0x0026	<u>0xF6</u>	} a = -10 -10 = 0 - (+10) = 0x00 - 0x0A = 0xF6 signextend to 16-bits, so a = 0xFFFF6	
0x0027	<u>0xFF</u>		
0x0028	<u>0xFF</u>		
0x0029	<u>0xFF</u>		
0x002A	<u>0x??</u>	} c[0]	
0x002B	<u>0x??</u>		c[1]
0x002C	<u>0x??</u>		c[2]
0x002D	<u>0x??</u>		c[3]
0x002E	<u>0x2D</u>	} ptrC = &c[3] = 0x002D	
0x002F	<u>0x00??</u>		

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

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

The target location is PC, so

$$PC = PC+2 + 2*N$$

$$N = (PC - (PC+2))/2 = -2/2 = -1 = 0xFF.$$

The format of the bra instruction is:

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

$$1101 0111 1111 1111 = 0xD7FF \text{ is the final answer}$$

e. Write assembly code for the following:

```
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
```

f. 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 $C=0 \parallel Z=1$ (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.

```
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.