**INEL 4206. 2nd EXAM. APRIL 29, 2011**
**Commented Solutions**

**Notation:** A register and contents are written in the form "*RegisterName = Contents*", with contents in hexadecimal notation; like SP = 0506h, R5=0x32AF. As for memory reference, "[*Address*] = *Memory Contents*"; Contents may be in byte size (covering only one physical address) as in [0FFE0h]= 2Ah, or in word size covering in fact two physical addresses, like for example [0xF2FE] = 0A234h.

**Remark:** For some problems two or more methods are mentioned. Anyone (or even another one) is acceptable as a response.

**Problem 1**. A memory location has as contents the byte 91h. In the information process, this datum has a meaning which depends on the convention used. What is the meaning for each of the following cases:

a. Decimal nonnegative integers coded in normal binary form
b. Signed integers coded in two's complement form
c. Real nonnegative numbers in fixed format F3.5
d. Real nonnegative numbers in fixed format F4.4
e. Real signed numbers in fixed format F4.4
f. Sub interval of the continuous interval between 0 and 3.

**Solution** : Transforming from hexadecimal to binary notation, 91h = 10010001B. We can start from this notation to work the interpretation:

a. Decimal nonnegative integers coded in normal binary form:

Method 1: By sum expansion of binary number:

$$10010001B = 1 \times 2^7 + 1 \times 2^4 + 1 = 128 + 16 + 1 = 145$$

Method 2: By sum expansion of hex number:

$$91h = 9 \times 16 + 1 = 144 + 1 = 145$$

b. Signed integers coded in two's complement form:

Method 1: by sum expansion

$$10010001B \rightarrow -1 \times 2^7 + 1 \times 2^4 + 1 = -128 + 16 + 1 = -111$$

Method 2: since the most significant bit is 1, we know the number is negative, the negative of the two's complement number 01101111B = 6FH = 6×16 + 15 = 96+15 = 111. Hence, the result is **-111**.

Method 3: Also by the two's complement information, but this time using the fact that we are working with eight bits, so from (a) we can calculate the number as -$(2^8 - 145)$= -111.

c. Real nonnegative numbers in fixed format F3.5:
Method 1: by sum expansion

$$100.10001B = 2^2 + 2^{-1} + 2^{-5} = 4 + 0.5 + 0.03125 = 4.53125$$

Method 2: By division using $145/2^5 = 145/32 = 4.53125$

d. Real nonnegative numbers in fixed format F4.4:
Method 1: by sum expansion

$$1001.0001B = 2^3 + 2^0 + 2^{-4} = 9 + 0.0625 = 9.0625$$

Method 2: By division using $145/2^4 = 145/16 = 9.0625$

e. Real signed numbers in fixed format F4.4:
Method 1: by sum expansion

$$1001.0001B \Rightarrow -2^3 + 2^0 + 2^{-4} = -8 + 1 + 0.0625 = -6.9375$$

Method 2: By division using -111/$2^4$ = -111/16 = -6.9375

f. Sub interval of the continuous interval between 0 and 3.

Solution: The total length of the interval is 3, and it is going to be divided in $2^8 = 256$ sub intervals, each of length 3/256. The decimal equivalent of 91h is 145. Hence, the sub interval represented with 91h is that between 145×(3/256) and 146×(3/256).
Using decimals, the sub interval represented is (1.69921875, 1.7109375)

**Problem 2**. Coding real numbers (those with integer and fractional part) is always a source of error because of the limited number of bits that can be used.

a. Using 6 bits in format F2.4 for nonnegative numbers, what is the code for 2.637?

b. When the binary representation is converted back to decimal form, what is the error with respect to the original number 2.637?

c. What is the minimum number of bits in the fractional part required to have an error less than 0.001 for any number coded in fixed format, assuming that the fractional part in the decimal expression has at most three digits? (Hint: consider the step in representations)

**Solution** To convert from decimal to binary,integer and fractional parts are worked separately.

a. Using 6 bits in format F2.4 for nonnegative numbers, what is the code for 2.637?

Integer part: 2 = 10B. For the fractional part:

$$0.637\text{x}2 = 1.274$$
$$0.274\text{x}2 = 0.548$$
$$0.548\text{x}2 = 1.096$$
$$0.096\text{x}2 = 0.192$$

Hence, the result of conversion is **10.1010B**

b. When the binary representation is converted back to decimal form, what is the error with respect to the original number 2.637?

Conversion of the binary representation gives 10.1010B = 2 + .5 + .125 = 2.625, with an error of 2.637-2.625=0.012.

c. What is the minimum number of bits in the fractional part required to have an error less than 0.001 for any number coded in fixed format, assuming that the fractional part in the decimal expression has at most three digits? (Hint: consider the step in representations)

The step should be less than 0.001 to guarantee that error. That means that $2^{-N} < .001$, or equivalently $2^N > 1000$. That is N=10.

**Ten bits in the fractional part will always guarantee an error less than .001**

**Problem 3**. We know from Mathematics that one way to compare two numbers to see which one is greater is to apply subtraction: A > B if and only if the subtraction A - B is positive. When working with digital systems we usually do the same operation, and use the flags in the status register to decide comparison. There is however, a special warning that we should take into account: in number representations we can only use 0's and 1's. Let the two representations to compare be P = 1101 1011 and Q = 0110 0100.

a. The decision on which number is greater will depend on the convention. State when we can conclude that P>Q and when that Q > P.
b. Subtraction is realized through two's complement addition. For P - Q, determine the Carry, the Sign and the Overflow flags. How would you conclude that P > Q and how that Q > P?
c. Repeat for the operation Q - P using two's complement.
d. Are the conclusions of comparison are similar?

**Solution**

a. The number representations are esentially for unsigned or signed situations. If the numbers are unsigned, then P>Q, since the most significant bit of P is 1 while that of Q is 0. On the other hand, for signed numbers Q>P because the most significant bits stand for the sign, negative por P and positive for Q.
b. Subtraction is realized through two's complement addition. For P - Q, determine the Carry, the Sign and the Overflow flags. How would you conclude that P > Q and how that Q > P?

   The operation P-Q carried out as sum of P with the two's complement of Q means:

$$\begin{array}{r} 11011011 + \\ 10011100 = \\ \hline 1\ 01110111 \end{array}$$

   The flags are as follows: Z=0 because the result is not 0; C=1, N=0. When the two terms are considered signed number, they are negative numbers because the most significant bit is 1; however, the addition yields a positive number, which means that there is an overflow. Hence V=1.

   I) Unsigned case: C=1 means that the subtraction P-Q does not need a borrow, so P>Q.

   II) Signed case: If there had been no overflow, a positive result would have meant P>Q, which we know is not the case. Hence, we should say that a positive sign with overflow (N=0, V=1) should mean that in signed numbers Q>P (subtrahend greater than minuend)

   (Explaining in other way: The following operations make sense: [NegativeNumber - PositiveNumber= NegativeNumber] and [PositiveNumber - NegativeNumber = PositiveNumber]. If the subtrahend and the minuend have different signs, we expect the result to have the same sign as the minuend, otherwise the operation is out of range. The fact of operation being out of range and result

being positive, means that the subtrahend is positive and minuend negative, so subtrahend > minuend)

c. Repeat for the operation Q - P using two's complement.

The operation Q-P carried out as sum of Q with the two's complement of P means:

$$\begin{array}{r} 01100100 + \\ 00100101 = \\ \hline 10001001 \end{array}$$

The flags are as follows: Z=0 because the result is not 0; C=0, N=1. When the two terms are considered signed number, they are positive numbers because the most significant bit is 1; however, the addition yields a negative number, which means that there is an overflow. Hence V=1.

I) Unsigned case: C=0 means that the subtraction Q-P needs a borrow, so Q<P or, equivalently, P>Q.

II) Signed case: If there had been no overflow, a negative result would have meant P>Q, which we know is not the case. Hence, we should say that a negative sign with overflow (N=1, V=1) should mean that in signed numbers Q>P (subtrahend greater than minuend)

(Explained in another way: Following the same reasoning as before, the result is of the same sign as the subtrahend, negative. Because of the overflow, we know that the minuend is positive. Hence, subtrahend < minuend)

d. Are the conclusions of comparison are similar?

Yes. In both cases the unsigned case resulted in P>Q and the unsigned case in Q>P.

**Problem 4**. Using an MSP430, programmer needs to add three signed data of byte size stored, respectively, in memory locations 0200h, 0201h and 0202h, and store the result immediately after the given data. Since the data can be any between the two limits of representation, it is rightly concluded that the sum should be done in word size.

a. What are the bounds for the numbers to be added and what is the address ? Why should the result be stored at address 0204h?

b. As it is commonly done, the programmer decides to use two sets of data to check for the correctness of the algorithm. The first set, decimal numbers, is -122, -108, 10. The second set is 107, 56, -80. In each case, when memory is examined, what should the programmer see (in hex notation) in memory to verify that the program functioned correctly?

c. The algorithm that the programmer thought (not necessarily optimized) to realize the operation is as follows:

**Step 1:** Initialize sum in a register R8 to 0,

**Step 2:** Point to address 0200h using R5

**Step 3:** Repeat 3 times, using R8 as counter:
Step 3.1: Load data in R6 incrementing pointer.
Step 3.2: convert data to word
Step 3.3: update sum

**Step 4:** Store the sum.

Assuming that the first set of testing data has been introduced, check if the following set of instructions do the work by explicitly giving the contents (in hex notation) of registers R5, R6, R7 and R8 as well as of memory locations 0204h and 0205h after each instruction (If unknown, use XXXX), following the flow of the program. Do not forget to repeat the loop!

```
        xor     R7,R7
        mov     #0200h,R5
        mov     #3,R8
LOOP:   mov.b   @R5+,R6
        sxt     R6
        add     R6,R7
        dec     R8
ELP:    jnz     LOOP        ;go back to LOOP if the counter is not zero.
        mov     R7,&0204h
```

d. BONUS: The programmer decides that the code can be more useful, extendible to N numbers, if storing of result is done at 0200h. D.1) What instructions would be changed and how? D.2) If data is to be kept after the addition, what strategy should be followed and which instruction then changed, and how.

**Solution** :

a. What are the bounds for the numbers to be added? Why should the result be stored at address 0204h?

Since the terms to be added are signed bytes, the lower bound is -128 and the upper bound +127. Hence, the addition may result in a number outside this range, so an extension to word is needed. Words are stored in even addresses; 0204h is the first even address after 0202h.

b. As it is commonly done, the programmer decides to use two sets of data to check for the correctness of the algorithm. The first set, decimal numbers, is -122, -108, 10. The second set is 107, 56, -80. In each case, when memory is examined, what should the programmer see (in hex notation) in memory to verify that the program functioned correctly?

**First set of validation data** : Converting to bytes: -122 = -128 + 6 ⇒ 10000110 = 86h; -108 = -128 + 20 ⇒ 1001 0100 = 94h; 10 ⇒ 0Ah. When sign extension to convert to 16-bit words is applied: -122 ⇒ FF86h, -108⇒FF94h and 10⇒000Ah. The addition yields FF86h + FF94h + 000Ah = 1FF24h, which in signed interpretation means -220. Therefore, the addition is correct and the result after the program is run should yield:

[0200h]=86, [0201h]=94, [0202h]=0A, [0204h]= 24, [0205h]=FF.

**Second set of validation data** : Converting to bytes: 107 = 64+32+11 ⇒ 0110 1011 = 6Bh; 56 = 32 + 16 + 8 ⇒ 0011 1000 = 38h; -80=-128 + 32+16 ⇒ 1011 0000= B0h. When sign extension to convert to 16-bit words is applied: 107 ⇒ 006Bh, 56⇒0038h and -80⇒FFB0h. The addition yields 006Bh + 0038h + FFB0h = 10053h, which in signed interpretation means 83. Therefore, the addition is correct and the result after the program is run should yield:

[0200h]=6B, [0201h]=38, [0202h]=B0, [0204h]= 53, [0205h]=00.

c. The algorithm that the programmer thought (not necessarily optimized) to realize the operation is as follows:

**Step 1:** Initialize sum in a register R8 to 0,

**Step 2:** Point to address 0200h using R5

**Step 3:** Repeat 3 times, using R8 as counter:
- Step 3.1: Load data in R6 incrementing pointer.
- Step 3.2: convert data to word
- Step 3.3: update sum

**Step 4:** Store the sum.

Assuming that the first set of testing data has been introduced, check if the following set of instructions do the work by explicitly giving the contents (in hex notation) of registers R5, R6, R7 and R8 as well as of memory locations 0204h and 0205h after each instruction (If unknown, use XXXX), following the flow of the program. Do not forget to repeat the loop!

The flow goes according to the table below. The data is [0200h]=86, [0201h]=94, [0202h]=0A

| | | | R5 | R6 | R7 | R8 | [0204h] | [0205h] |
|---|---|---|---|---|---|---|---|---|
| | **xor** | R7,R7 | XXXX | XXXX | 0000 | XXXX | XX | XX |
| | **mov** | #0200h,R5 | 0200 | XXXX | 0000 | XXXX | XX | XX |
| | **mov** | #3,R8 | 0200 | XXXX | 0000 | 0003 | XX | XX |
| LOOP: | **mov.b** | @R5+,R6 | 0201 | 0086 | 0000 | 0003 | XX | XX |
| | **sxt** | R6 | 0201 | FF86 | 0000 | 0003 | XX | XX |
| | **add** | R6,R7 | 0201 | FF86 | FF86 | 0003 | XX | XX |
| | **dec** | R8 | 0201 | FF86 | FF86 | 0002 | XX | XX |
| ELP: | **jnz** | LOOP | 0201 | FF86 | FF86 | 0002 | XX | XX |
| LOOP: | **mov.b** | @R5+,R6 | 0202 | 0094 | FF86 | 0002 | XX | XX |
| | **sxt** | R6 | 0202 | FF94 | FF86 | 0002 | XX | XX |
| | **add** | R6,R7 | 0202 | FF94 | FF1A | 0002 | XX | XX |
| | **dec** | R8 | 0202 | FF94 | FF1A | 0001 | XX | XX |
| ELP: | **jnz** | LOOP | 0202 | FF94 | FF1A | 0001 | XX | XX |
| LOOP: | **mov.b** | @R5+,R6 | 0203 | 000A | FF1A | 0001 | XX | XX |
| | **sxt** | R6 | 0203 | 000A | FF1A | 0001 | XX | XX |
| | **add** | R6,R7 | 0203 | 000A | FF24 | 0001 | XX | XX |
| | **dec** | R8 | 0203 | 000A | FF24 | 0000 | XX | XX |
| ELP: | **jnz** | LOOP | 0203 | 000A | FF24 | 0000 | XX | XX |
| | **mov** | R7,&0204h | 0203 | 000A | FF24 | 0000 | 24 | FF |

d. BONUS: The programmer decides that the code can be more useful, extendible to N numbers, if storing of result is done at 0200h. D.1) What instructions would be changed and how? D.2) If data is to be kept after the addition, what strategy should be followed and which instruction then changed, and how?

D1) **mov #3,R8** is changed to **mov #N,R8**, where N is the number of terms; and **mov R7,&0204h** is changed to **mov R7,&0200h**

D2) Data should be stored starting at 0204h instead of 0200h, and **mov #0200,R5** changed to **mov #0204,R5**