1: Problem 3.1 What is 5ED4 - 07A4 when these values represent unsigned 16- bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

`0x5ED4` $-$ `0x07A4` $= 24276 - 1956 = 22320 =$ `5730`

2: Problem 3.3 Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

In binary representation, $(5ED4)_{16} = (0101\ 1110\ 1101\ 0100)_2$. Because in hexadecimal, each bit could map 16 numbers to 16 symbols $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)$, in other words, it's as 2^4 permutations with repetition. Hence, we could use two hex bits to represent one byte (instead of 8-bit in binary representation).

3: Problem 3.6 Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate $185 - 122$**. Is there overflow, underflow, or neither?**

Since $185 - 122 = 63$, thus the answer is neither.

4: In class we discussed an algorithm for implementing decimal addition of unsigned integers. Modify this algorithm so that it can be used to implement addition of unsigned hexadecimal integers. Use base 10 to represent any constants in your algorithm.

```
1         void Add(int addend1[], int addend2[], int sum[],
              int max_digit) {
3
              int carry = 0;
5             int temp;
              for (int digit = 0; digit < max_digit; digit++) {
7                  temp = addend1[digit] + addend2[digit] + carry;
                   sum[digit] = temp % 16;
9                  carry = temp/16;
              }
11
              sum[max_digit] = carry;
13            /*this will overwrite, say addend2[0],
              i.e., something is defined near the sum[],
15            it depends on your main function.
               If it overwrites,
17             try reassign carry to -1 of the sum[].*/
               //sum[-1] = carry;
19
}
```

5: In class we looked at an algorithm for implementing decimal subtraction of unsigned integers. This algorithm assumes that the minuend is greater than or equal to the subtrahend. What happens in the algorithm if the minuend is less than the subtrahend? Modify the algorithm so that it correctly finds the difference if the minuend is less than the subtrahend. Assume that the unary minus operator x = -y has been implemented for the arrays representing integers.

```
void Sub(int minu[], int subt[], int diff[], int max_digits) {
int power = 1;
  int d1=0;
  int d2=0;
  int temp[max_digits];
  for(int j= max_digits-1;j>=0; j--)
    {
      d1 += minu[j]*power;
      power *=10;
    }
  power =1;
    for(int j= max_digits-1;j>=0; j--)
    {
      d2 += subt[j]*power;
      power *=10;
    }
  if(d2>d1){
    for(int s = 0; s< max_digits; s++){
      temp[s] = minu[s];
      minu[s] = subt[s];
      subt[s] = temp[s];
    }
  }
  int update[max_digits];
  for (int digit = 0; digit < max_digits; digit++){
    update[digit]=0;
  }
  for (int digit = 0; digit < max_digits; digit++) {
  update[digit] = minu[digit];  // Array assignment
  }
  for (int digit = 0; digit < max_digits; digit++) {
    if (update[digit] < subt[digit]) {
      update[digit] += 10;
      int i = digit - 1;
      while (update[i] == 0) {
        update[i] = 9;
        diff[i] = update[i];
        i--;
      }
```

```
40        update[i]--;
          diff[i] = update[i];
42      }
        diff[digit] = update[digit] - subt[digit];
44    }
  }
```

6: In class we discussed when overflow can occur with addition and subtraction of signed integers. Our rules depended on the signs of the two operands: whether the signs are the same, or one sign is positive and one is negative. The rules didn't consider the case when one of the operands is zero. Is it possible that addition or subtraction will overflow if one of the operands is zero? Explain your answer.

(a) First of all, this problem is really interesting!

(b) Secondly, let's consider how many cases we could have:

We have two operands, A, and B, and each of them could have three choices–positive, zero, or negative.
And let's use these three symbols to represent those three choices: $\{+, 0, -\}$. Now, obviously, there are two groups, and let's write them down:

Group (i): one of the operands is negative; the other is nonnegative

$$(+,-), and (0,-). \tag{1}$$

We could count the number of them by using the following equation:

$$C_1^2 = 2 \tag{2}$$

Group (ii): the remaining cases (a combo of the same sign cases plus $(0,+)$)

$$(+,+),(-,-),(0,0), and (0,+). \tag{3}$$

Again, we could check whether there are any cases we are missing by counting the number of them by using the following equation:

$$\frac{C_2^3 C_1^1}{2} + 1 = 3 + 1. \tag{4}$$

And, in our class, we have seen the results of $(+,+), and (-,-)$. But, in the following proof, we will revisit the same results once again naturally.

For the Group (i), I proved a statement that is true and call it as our Theorem 1, then by applying Theorem 1 we could prove the result for the remaining cases which are in Group (ii) and called this result as the Theorem 2.

Theorem 1. *If one of the operands is nonnegative, and the other is negative, then the sum does not overflow.*

Theorem 2. *If and only if the sum is over flow, the operands are in the following two cases:*
Case a. $A < 0$, $B < 0$, Sum ≥ 0
Case b. $A \geq 0$, $B \geq 0$, Sum < 0
where we define Sum that is represented in binary pattern and produced by the adder. The correct result we use $A+B$ to represent.

Proof 1. *(in " =>" direction)*
To prove our Theorem 1 is rather trivial, we only need to take the sum of the two inequalities. Suppose the two operands A, and B are n-bit binary numbers, then we can write two inequalities, and the mathematical form of the overflow as follows:
if ($-2^{n-1} \leq A < 0$ and $0 \leq B < 2^{n-1}$) then $-2^{n-1} \leq A + B < 2^{n-1}$, in other words, it dosen't overflow. This result holds by the symmetry that if we exchange A and B.

Now, based on the First-oder logic, we could have a handy tool that if we know p implies q, then not q implies not p.

Proof 2. *(in " =>" direction)*
So, the "not q implies not p" of our Theorem 1 could give us the following cornerstone: It overflows iff. not "one of the operands is nonnegative, and the other is negative." That is one is as long as two operands are negative or negative, or two are zeros, or one is zero and one is positive, in short, both of the operands are in $(0,0), (+,+), (-,-),$ or $(0,+)$ cases. Apparently it indicates us to the Group (ii).

Since "not negative, $A < 0$, $B < 0$," and "not positive or zero, $A \geq 0$, $B \geq 0$," could be seen as two sub cases (or subgroups), we first consider the not positive or nonzero case, then consider the other.

Case a. Let's start from two legitimate inequalities:

$$0 \leq A < 2^{n-1}, \tag{5}$$

and

$$0 \leq B < 2^{n-1}. \tag{6}$$

The summation of the inequality equations 5 and 6 is

$$0 \leq A + B < 2(2^{n-1}) = 2^n, \tag{7}$$

Now, since we are in proving the " =>" direction, so we have the assumption that it dose overflow which means $A + B \geq 2^{n-1}$ by taking the negation of the q statement of our Theorem 1. Taking both conditions into account, we obtain the following:

$$2^{n-1} \leq A + B < 2^n. \tag{8}$$

Subtract 2^n in each sides:

$$2^{n-1} - 2^n \leq A + B - 2^n < 0, \tag{9}$$

$$=> (-1)((2^n) - 2^{n-1}) \leq A + B - 2^n < 0, \tag{10}$$

$$=> (-1)(2^n)(1 - 2^{-1}) \leq A + B - 2^n < 0, \tag{11}$$

$$=> (-1)(2^{n-1})(2 - 1) \leq A + B - 2^n < 0, \tag{12}$$

$$(-1)(2^{n-1}) \leq A + B - 2^n < 0, \tag{13}$$

Although we could use two's complement to represent the $A + B - 2^n$ (the Sum represented by the adder), we have already reached the result of the Case a that if it overflows and both operands are nonnegative (that is $0 \leq A$, and $0 \leq B$), then the sum would be less than zero.

Case b. Agian, let's start from two legitimate inequalities:

$$-2^{n-1} \leq A, \tag{14}$$

and

$$-2^{n-1} \leq B, \tag{15}$$

Take the summation of them:

$$-2^n \leq A + B, \tag{16}$$

Similarly, since we are in proving the " =>" direction, so we have the assumption that it dose overflow which means $A + B < -2^{n-1}$ by taking the negation of the q statement of our Theorem 1. Taking both conditions into account, we obtain the following:

$$-2^n \leq A + B < -2^{n-1}. \tag{17}$$

Let's start to massage the above inequalities a little bit:

$$0 \leq A + B + 2^n < 2^{n-1}. \tag{18}$$

Once again, though we could use two's complement to represent the $A + B + 2^n$ (the Sum represented by the adder), we have already reached the result of the Case b that if it overflows and both operands are nonpositive (that is $A < 0$, and $B < 0$), then the sum would be zero or positive.

(in "<=" direction)
This direction is rather trivial. Since in Case a. if $A < 0$, $B < 0$, then if the sum is larger than or equal to zero, obviously, it has already overflown. Likewise, in Case b. if $A \geq 0$, $B \geq 0$, and then if their summation is negative (in two's complement world, so it's possible), then again is a sign of overflow.

In summary, according to our above derivations and proofs, we could learn that if there is only one zero in one of the two operands and the other is negative, then by applying our Theorem 1, we could know unless the machine breaks the math, there is no chance to end up with overflows. On the other hand, if there are two operands are zero or one is zero and the other is positive, then there is a chance that according to our Theorem 2, if their sum is negative, then it does overflow (and since this is an iff statement, so vice versa).