EE2026 (Part 1)
Tutorial 2 - Solutions

1. (a) $(250)_{10} = (11111010)_2$ (being integer, just apply the iterated division by 2 and take the remainder)

(b)

(i). $11111010(\textbf{signed magnitude}) \longrightarrow -122$ (sign is negative because MSB = 1, magnitude is simply expressed by the remaining bits 1111010)

(ii). $11111010(\textbf{1's}) \xrightarrow{\textbf{complement.}} 00000101(\textbf{magnitude}) \longrightarrow -5$

Indeed, sign is negative since MSB = 1.

Now, let us evaluate the magnitude. Being negative, the given number 11111010 represents the 1's complement A^* of the magnitude A, by definition of 1's complement representation. By definition of 1's complement, we have $A^* = 2^n - 1 - A$, hence the magnitude A is equal to $A = 2^n - 1 - A^*$ (i.e., the magnitude is obtained from the 1's complement by simply evaluating the 1's complement of the latter). Hence, the magnitude results to the 1's complement of 11111010, which is 00000101.

(iii). $11111010(\textbf{2's}) \xrightarrow{-1} 11111001(\textbf{1's}) \xrightarrow{\textbf{complement}} 00000110(\textbf{magnitude}) \rightarrow -6$

Same considerations apply here. The only difference is that $A^* = 2^n - A$, hence $A = 2^n - A^*$. Again, this means that the magnitude of the 2's complement representation of a negative number is simply obtained by evaluating its 2's complement.

2. (a) (-1) + 45

```
   11111111
 + 00101101
 100101100 ——→ 44
```

100101100 $\longrightarrow 44$

(Adding these two numbers causes a carry over into the 9th bit position, which is ignored in the 8-bit arithmetic system.)

(b) (-128) + (-60)

```
   10000000
 + 11000100
   01000100 ——→ 68
```

This example is particularly interesting since it considers the case of an "overflow", i.e. the result is constrained to have the same number of bits (bit width) as the operands, and hence the result can be out of the range that is covered by the 2's complement representation with 8 bits ($-2^{8-1} \ldots 2^{8-1}-1$, i.e. -128...127).

Being -128 the minimum value that can be represented with 8-bit 2’s complement representation, subtracting 60 clearly leads to a result that is beyond the range, and an overflow occurs.

Now, the question is how to detect an overflow in an addition in a computer, where usually the bit width of the result is the same as the operands. To answer the question, we first observe that the sum of a positive and a negative number in the above range is always within the same range. In other words, the overflow can occur only if the two operands have the same sign.
When the operands have the same sign, the result should clearly have the same sign. To better understand, let us assume the two operands are positive (same considerations hold for negative numbers). As long as the result is within the correct range, its MSB in 2’s complement representation will be 0, being a positive number. If the result exceeds the maximum positive number that is within the range (i.e., 011...11), its MSB will become 1 and the result will hence represent a negative number (which is clearly incorrect).

Hence, overflow occurrence can be simply checked as follows:
- compare the sign (i.e., MSB) of operands
    - if it is different, no overflow occurs (OVERFLOW = 0)
    - if it is the same, compare the sign (MSB) of the result
        - if the MSB of the result is the same as the MSB of the operands, no overflow occurred (OVERFLOW = 0)
        - otherwise, overflow was occurred, which will be signaled by raising OVERFLOW = 1 (i.e., the computer performs the calculations, providing the result, as well as the OVERFLOW signal to confirm the correctness of the result or not).

3. $(00100)_{SM} = (00100)_{2's}$ [the number is positive]
$(10100)_{2's} + (00100)_{SM} = (10100)_{2's} + (00100)_{2's} = (11000)_{2's}$
Convert to integers and add to verify your result!

4. $\underbrace{0100}_{4}\underbrace{0110}_{6}\underbrace{0010}_{2}\underbrace{0011}_{3}$
= 4623