A Method for bit flipping:

nor gate: this sounds like "not or" but it actually is the operation of the "**not**" of the "**or**"

Suppose we have two inputs **A** and **B**

Let us compute the value of nor in the following way:

A	**B**	**(A OR B)**	**(NOT {A OR B} AKA "NOR")**
0	**0**	**0**	**1**
0	**1**	**1**	**0**
1	**0**	**1**	**0**
1	**1**	**1**	**0**

So in this problem, if I am asking you to use "NOR" you need to figure out what that particular gate where that particular gate gets used.

Let us fix the value for Input B equal to zero. We see the following cases:

A	**B**	**(A OR B)**	**("NOR")**
0	**0**	**0**	**1**
1	**0**	**1**	**0**

Notice anything about the relationship between the NOR result and the input A? We can use this gate to flip the bits of input A by simply setting B = 0 and using A **NOR** B! Recall that this operation in MIPS is bitwise, so it will do it to every bit pair individually.

Since the method for two's complement of a number is to use bit flipping and then adding the constant 1, start with using the method above!

More on two's complement:

Suppose the number in base 10/ decimal is -3.

The binary value (of 3) is 0011

Then the bit flipped value is 1100

And by adding 1 we get 1101

Though 1101 looks like 13, I am hiding something (a lot of ones)! More on this later. Suppose we call the function **T** the method of computing two's complement.

Then **T(3) =** -3 (in base 10) **=** 1101 (in binary)

Consequently, we can use this little trick for subtraction! Suppose we have 7 – 3. Then we know this equals 7 + **T(3).**

I will show this using all 4 bytes (a full word).

7 in binary: 00000000 00000000 00000000 00000111

3 in binary: 00000000 00000000 00000000 00000011

T(3): 11111111 11111111 11111111 1111**1101**

Notice that before I was only showing you the **1101** part from before. However, what I did not show was all the 1 values in front. These come from the fact that we flip **all** the bits!

So now, in computing 7 + **T(3)** I will go bit by bit. Notice the carry row, which happens when we add 1+1. Just like when we add 35+5 in base 10, we must compute 5+5=10, which gives us a 0 in the ones place and a 1 in the **carry** (10s place), which we add to the 3, giving us 40. We start the carry row with a *zero* (ital).

7	00000000	00000000	00000000	00000111
T(3)	11111111	11111111	11111111	11111101
C (carry)	**11111111**	**11111111**	**11111111**	**1111111*0***
Sum =7+T(3)+C	00000000	00000000	00000000	00000100

Notice that the carry continues to happen (seen by the huge number of 1s)! As a result, the value of the two's complement gives this "infinite carry" process, which is kind of an incredible binary hack! This is what allows us to perform subtraction (using addition only!), since we see in the **sum** column the value **00000100** = 4 = 7 – 3! Thus, this demonstrates that two's complement is logically equivalent to the negative because it works in computing subtraction!