Dear All,

Mea culpa. I gave an assignment that I thought to be feasible but which turned out to be quite tough. That's why I am writing this blog to show you how to work towards the solution step by step.

Phase 1: the program that is in the presentation is as shown on the right::

```
def Nand(p,q):
    return 0 if p&q else 1
def Or(p,q):
    return Nand(Nand(p,p),Nand(q,q))
def And(p,q):
    return Nand(Nand(p,q),Nand(p,q))
def Xor(p,q):
    r = Nand(p,q)
    return Nand(Nand(p,r),Nand(r,q))
def bit(b):
    return 0 if b=='0' else 1
calc = input('addition?\n')
bins = calc.split('+')
for i, bin in enumerate(bins):
    bins[i] = ([0]*(8-len(bin))) + \
        [ bit(c) for c in bin ]
A,B = bins
C=[0]*9
S=[0]*8
S[-1]=Xor(A[-1],B[-1])
C[-2]=And(A[-1],B[-1])
for i in range(-2,-9,-1):
    S[i] = Xor(Xor(A[i],B[i]),C[i])
    C[i-1] = Or(And(Xor(A[i],B[i]),C[i]),
                And(A[i],B[i]))
print(''.join(str(b) for b in S))
```

According to the assignment, we first have to perform the addition in a separate function (the part that calculates S for given A and B):

```
def add(A,B):
    C=[0]*9
    S=[0]*8
    S[-1]=Xor(A[-1],B[-1])
    C[-2]=And(A[-1],B[-1])
    for i in range(-2,-9,-1):
        S[i] = Xor(Xor(A[i],B[i]),C[i])
        C[i-1] = Or(And(Xor(A[i],B[i]),C[i]),
                    And(A[i],B[i]))
    return S
```

This code is unchanged from the original program. We have only put it in a function that when given bit-lists A and B as arguments, returns the sum S.

```
sign=1
if '+' in calc:
    bins = calc.split('+')
elif '-' in calc:
    bins = calc.split('-')
    sign = -1
```

The next step (2nd bullet) looks whether the input contains a "-" or not: if so, the input is a subtraction so we have to take the 2's-complement of the second argument. We store the fact whether the input contains a + or a - in a helper variable called 'sign'. Sign is 1 if the input contains a + and otherwise -1.

In the third bullet we have to calculate the twos-complement of a bit-list. Twos-complement is computed by inverting each bit, and adding 1 to the result. So we write a function called Not, using a Nand gate, and we add another function called Nots which inverts each bit in a bit-list:

```
def Not(a):
    return Nand(a,a)
def Nots(A):
    return [Not(a) for a in A]
```

Adding one to the result is surprisingly easy, because we already have a function 'add' (we just created it above). Adding 1 to a bitlist A simply calls that function with the variable and the binary representation of 1: `add(A,[0,0,0,0,0,0,0,1])`
In principle we are done with phase 1.

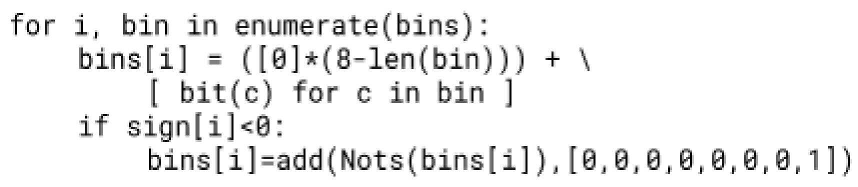
In phase 2 we must accept both positive and negative inputs. To handle all situations we introduce a function 'parse' which analyses the input. So, parse(calc) accepts the input calculation (as a string), and returns (as before) a list of two binary numbers (represented as bitlists). Where each number has been converted to two's-complement as appropriate. In the red text in phase 1 above, we used a single variable 'sign', but since there are two inputs we now use a list variable for sign.
If the very first character of the input is '-', the first sign is -1, and we must subsequently ignore that first character:

```
sign = [1,1]
if calc[0]=='-':
    calc = calc[1:]
    sign[0] = -1
```

```
if '+-' in calc:
    bins = calc.split('+-')
    sign[1] = -1
elif '-' in calc:
    bins = calc.split('-')
    sign[1] = -1
else:
    bins = calc.split('+')
```

Then, we must distinguish three situations regarding the remaining input. It either contains '+-', meaning addition of a negative number, or a sole '+' meaning addition, or a sole '-' meaning subtraction. For subtraction or for addition of a negative number (which is the same thing), we only need to remember to use a negative sign for the second argument (sign[1])

Now we can finally perform the calculation. It is as before, except where we replace an argument with its twos-complement as appropriate.



```
for i, bin in enumerate(bins):
    bins[i] = ([0]*(8-len(bin))) + \
        [ bit(c) for c in bin ]
    if sign[i]<0:
        bins[i]=add(Nots(bins[i]),[0,0,0,0,0,0,0,1])
```

As mentioned, don’t feel bad if you didn’t come up with all this on your own. All in all it is tough, but there is nothing here that we haven’t seen before, it is just being used in new and different ways. That is programming.

You must take the time and go through this blog in order to understand each step.

```
def Nand(p,q):
      return 0 if p&q else 1
def Or(p,q):
      return Nand(Nand(p,p),Nand(q,q))
def And(p,q):
      return Nand(Nand(p,q),Nand(p,q))
def Xor(p,q):
      r = Nand(p,q)
      return Nand(Nand(p,r),Nand(r,q))
def Not(a):
      return Nand(a,a)
def bit(b):
      return 0 if b=='0' else 1
def add(A,B):
      C=[0]*9
      S=[0]*8
      S[-1]=Xor(A[-1],B[-1])
      C[-2]=And(A[-1],B[-1])
      for i in range(-2,-9,-1):
            S[i] = Xor(Xor(A[i],B[i]),C[i])
            C[i-1] = Or(And(Xor(A[i],B[i]),C[i]),
                              And(A[i],B[i]))
      return S
def Nots(A):
      return [Not(a) for a in A]
def parse(calc):
      sign = [1,1]
      if calc[0]=='-':
            calc = calc[1:]
            sign[0] = -1
      if '+-' in calc:
            bins = calc.split('+-')
            sign[1] = -1
      elif '-' in calc:
            bins = calc.split('-')
            sign[1] = -1
      else:
            bins = calc.split('+')
      for i, bin in enumerate(bins):
            bins[i] = ([0]*(8-len(bin))) + \
                  [ bit(c) for c in bin ]
            if sign[i]<0:
                  bins[i]=add(Nots(bins[i]),[0,0,0,0,0,0,0,1])
      return bins

calc = input('addition?\n')
A,B = parse(calc)
print(''.join(str(b) for b in add(A,B)))
```