COL100 Assignment 1

Due date: 20 December 2021

In these assignments, you will be asked to design several algorithms and implement them as functions in Python. All Python functions must be documented with comments explaining the purpose of the function, the types and meanings of the input arguments, and the type and meaning of the return value. For example:

```
def mean(a, b):
    # The mean of two numbers.
    #
    # Parameters:
    # a: float -- the first number
    # b: float -- the second number
    #
    # Returns: float -- the mean of a and b

    return (a + b)/2
```

Part 1

Instead of the decimal numeral system that we are all familiar with, computers typically represent numbers using the *binary* numeral system. In binary, there are only two digits (0 and 1) instead of ten (0, 1, …, 9), and each place value increases by a factor of two. For example, the number of dots below

is written as 14 (meaning 1 ten and 4 ones) in decimal, and 1110 (meaning 1 eight, 1 four, 1 two, and 0 ones) in binary.

In Python, let us use Boolean values (i.e. values of type `bool`) to represent bi-

nary digits, with `False` representing 0 and `True` representing 1. A tuple of four Booleans `(d0,d1,d2,d3)` will represent the binary numeral $d_3d_2d_1d_0$, which in turn represents the number $d_3 \times 2^3 + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$. Note that we have chosen the binary digits, a.k.a. *bits*, to be in reverse order in the tuple. Thus, the number above (1110 in binary) will be represented as `(False,True,True,True)`.

1. The so-called "full adder" takes three bits and returns their sum as a 2-bit numeral. Implement a Python function `add(a,b,c)` that performs this operation using the logical operators `and`, `or`, `not`, and returns the sum as a tuple `(s0,s1)`.

   Example: In binary, 1 + 0 + 1 = 10, so your function should behave as

   ```
   >>> add(True, False, True)
   (False, True)
   ```

   Hint: Consider the following "half adder" that adds only two bits:

   ```
   def hadd(a,b):
       return ((a or b) and not (a and b), a and b)
   ```

   Can you see why this works? Consider all four possibilities for the input values, such as `hadd(True,True) = (False,True)` representing 1 + 1 = 10 in binary. Try building your full adder using the half adder as a starting point.

2. Implement a 4-bit adder, `add4(a0,a1,a2,a3, b0,b1,b2,b3, c)`. This function should take two 4-bit numerals $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$ along with a carry bit c, and return a 5-bit numeral representing $a_3a_2a_1a_0 + b_3b_2b_1b_0 + c$.

   Example: Eleven (1011) plus nine (1001) is twenty (10100), so your function should give

   ```
   >>> add4(True,True,False,True, True,False,False,True, False)
   (False,False,True,False,True)
   ```

   Hint: Consider calling your full adder function several times and chaining the results together. You can use something like `(x,y) = add(a,b,c)` to get the individual components out of the returned tuple.

3. Take the last two digits of your entry number and find the remainder when divided by 4. Depending on the result, you get one of the following operators:

$$0 \to\ <, \qquad 1 \to\ \leq, \qquad 2 \to\ >, \qquad 3 \to\ \geq .$$

Implement a comparator `cmp(a0,a1,a2,a3, b0,b1,b2,b3)` that compares two 4-bit numerals according to your assigned operator, and returns `True` or `False` depending on whether the comparison holds. For example, if your entry number is 2021CS12345, then your operator is $\leq$, and your function should return `True` if $a_3a_2a_1a_0 \leq b_3b_2b_1b_0$ and `False` otherwise.

Example: If your operator is $\leq$, your function should verify that eleven is *not* less than or equal to nine by returning

```
>>> cmp(True,True,False,True, True,False,False,True)
False
```

So far, using 4 bits we can only represent nonnegative integers between 0 and 15. To represent negative integers, there are many different encodings we could choose. Probably the most intuitive way is *sign-magnitude form*, where we add a extra bit to represent sign. So a tuple of Booleans `(d0,d1,d2,d3,s)` represents the number $(-1)^s \times d_3d_2d_1d_0$; for example fourteen is now `(False,True,True,True,False)`, while minus-fourteen is `(False,True,True,True,True)`.

4. Implement a 4-bit subtractor `sub4(a0,a1,a2,a3, b0,b1,b2,b3)` that takes two 4-bit numerals and returns their difference $a_3a_2a_1a_0 - b_3b_2b_1b_0$ in sign-magnitude form.

Example: Since nine (1001) minus eleven (1011) is minus-two (-10), your function should produce

```
>>> sub4(True,False,False,True, True,True,False,True)
(False,True,False,False,True)
```

Hint: Use the comparator you wrote in the previous question.

Part 2

5. Implement an 8-bit adder using two 4-bit adders. Rather than writing a function that takes 17 Boolean arguments, have it accept each 8-bit numeral as a single tuple, `add8(a, b, c)`, and return a pair `(s, cout)` containing an 8-bit numeral and a new carry.

   Example: Sixty-three (111111) plus one (1) is sixty-four (1000000), so we expect

   ```
   >>> a = (True,True,True,True,True,True,False,False)
   >>> b = (True,False,False,False,False,False,False,False)
   >>> add8(a, b, False)
   ((False,False,False,False,False,False,True,False), False)
   ```

   Hint: You can get the components of the input tuples by assigning local variables inside the function, for example

   ```
   def add8(a, b, c):
       (a0,a1,a2,a3,a4,a5,a6,a7) = a
       ...
   ```

6. Implement a multiplier `mul4(a, b)` that takes two 4-bit numerals and computes their 8-bit product using the repeated addition algorithm. Use your function `add8` for addition, and `sub4` for decreasing by 1.

   Example: Eleven (1011) multiplied by nine (1001) is ninety-nine (1100011, because it equals sixty-four + thirty-two + two + one). So we want

   ```
   >>> mul4((True,True,False,True), (True,False,False,True))
   (True,True,False,False,False,True,True,False)
   ```