CS52 - Assignment 5

Due Friday 3/11 at 5pm

https://xkcd.com/643/

Getting Started

Note, for this assignment (and for this class in general) we will be using python version 2.7.x (2.6.x is probably fine too). There is a version 3.X available, however, it isn't as widely adopted and has enough differences that you'll need to use the 2.7.x version. This should come installed on most laptops, though you can also download the latest version from `https://www.python.org/`.

The starter files can be found at:

`http://www.cs.pomona.edu/~dkauchak/classes/cs52/assignments/assign5/`

I've provided you with two files

- `assign5-starter.txt`: Some start code in python that has the base classes that you'll be extending.
- `assign5.out`: A text file with the correct output from part 3 of the assignment.

Create a directory in your `cs052` folder for assignment 5 and put both of these files in there. Rename `assign5-starter.txt` to be `assign5.py` (make sure to change the extension from .txt to .py), put

your name at the top of the file and then start working on the assignment at the bottom of that file. *You should not edit any of the code I have provided; only add your own.*

Before starting, make sure you understand *all* of the code provided in the starter, particularly what each of the classes represents and what each of the methods do. These are the two general classes:

- A `Gate` has input nodes and one output node. It is manipulated via two methods. `evaluate` reads the input nodes and returns the value for that gate. `notify` obtains the resulting value for the current inputs by calling `evaluate` and, if necessary, sets the output node to the correct value.
- A `Node` is a junction. (Think of a gate as having wires for input and output; a node is where the wires are connected.) A node maintains its state, which is two things: 1) the current value being send along the Node as a bit (0 or 1) and 2) a list of all gates whose inputs are connected to that node. There are three methods: `get_state` which returns the current bit value, `set_state` which changes the state and, if necessary, calls `notify` for all the gates in the input list (which will propagate the change to the gates), and `connect` which adds a gate to the notification list.

Once a circuit has been created (see the class notes for examples) computation happens by:

1. You call the `set_state` method of an input node.
2. If the state is different than the previous state of the node, it calls the `notify` method for all gates that the node is an input for.
3. Each connected gate (in the `notify` method), then, recalculates its output through `evaluate`. If the output is different than it's pervious value, the gate will then propagate the signal by calling `set_state` on the output node.
4. The output nodes may also be input notes to other gates, so this process then repeats until the output of all of the gates has been updated.

Python Circuits

1. **[6 points]** The `gate`way to happiness

   To start with we will build up a few basic gates and then build our adders for doing circuit-based addition.

   (a) We have provided class definitions for three gates already:
   - `NotGate1`: a *not* gate with 1 input
   - `AndGate`: a variable input size *and* gate (constructed with a *List* of input nodes)
   - `AndGate2`: a two input *and* gate

   Using these as templates, write additional gate definitions. You must include at least:

- `OrGate`: a variable input size *or* gate
- `OrGate2`: a two input *or* gate
- `XorGate2`: a two input size *xor* gate

Note the suffix of the class name denotes the number of inputs the gate takes (none for variable number of inputs).

(b) Using these gates, create a class called `HalfAdder` that implements a half adder. The `HalfAdder` will *not* extend any of the gate classes. Instead, it should have a constructor (i.e `__init__` method) that takes in the appropriate input and output nodes and then uses the gate constructors to create a half adder.

(c) Create a class called `FullAdder` that implements a full adder. Like the `HalfAdder` class, it will not extend any gate class, but will instead use them in the constructor. Your `FullAdder` *must* be constructed from `HalfAdder`s (and possibly a few other gates).

2. **[6 points]** This problem is a great "addition" to the assignment

Create a class called `Adder` that represents a four-bit ripple-carry adder, as described in class and the reading. The constructor for the adder should have five parameters (three are for input and two are for output):

- a List of four `Nodes` for the first input,
- a List of four `Nodes` for the second input,
- a `Node` for the D bit that differentiates between addition and subtraction,
- a List of four `Nodes` for the output, and
- a List of four `Nodes` for the flags C, Z, N, and V, in that order.

Like the `Adder` and `HalfAdder`, this class will not extend any classes and the constructor should be the only method that you would call outside the class, though you might write helper methods that are used internally.

Hints/requirements/observations:

- All of the computation *must* happen via circuits (i.e. `Node`s and `Gate`s, e.g. you should not try and calculate the output flags using any mathematical operators.
- As a side effect of this, once you have instantiated an instance of your `Adder` all of the "computation" happens by setting the input `Nodes` and then reading the output `Nodes`. Specifically, no methods will be called of the `Adder` class. The circuit elements will automatically adapt to any changes in the input via the mechanism described above.
- Remember the definitions of the flags:

  C, the carry flag is the carry-out bit (or the borrow bit in the case of subtraction) from the most significant full adder;

  Z, the zero flag, is 1 if result is zero and 0 otherwise;

  N, the sign flag, is s_k, the high-order bit of the result; and

  V, the overflow flag, is 1 if, in the most significant full adder, the input bits are the same *and* they differ from the sum bit.

3. **[6 points]** Let's take this circuit for a spin

   Demonstrate that your adder works by using it to generate values for a table of *all* of the results in the following form.

```
a= 0 b= 0  sum= 0,CZNV=0100  diff= 0,CZNV=0100
a= 0 b= 1  sum= 1,CZNV=0000  diff=15,CZNV=1010
a= 0 b= 2  sum= 2,CZNV=0000  diff=14,CZNV=1010
a= 0 b= 3  sum= 3,CZNV=0000  diff=13,CZNV=1010
...
a=15 b=14  sum=13,CZNV=1010  diff= 1,CZNV=0000
a=15 b=15  sum=14,CZNV=1010  diff= 0,CZNV=0100
```

   The complete table has 256 lines and is in the file `assign5.out` available from the starter.

   The output requires careful formatting. Your table should be character-for-character identical to the contents of that file. There is one blank space before the `b`, two blanks before the words `sum` and `diff`, and one blank before each single-digit integer.

   Put the code to print the output at the very bottom of the file—so that the printing is done immediately without the need to call another function. One way to test your result against the solution file is to copy the correct file to your working directory and then issue the command

```
python < assign5.py | diff - asgt05.out
```

   The output from that command will tell the differences between your result and the solution file. If there is no output, then your result matches the file exactly—no news is good news.

   Hints:

   - You will need to investigate Python's output facilities. See the links to the Python documentation at the bottom of the course web page.
   - Consider how to break up this part of the assignment into smaller chunks and write helper functions, e.g. functions that translate back and forth between lists of `Nodes` and integers. If you find yourself writing a lot of code and/or copying and pasting a lot, there's probably an easier way to do it.
   - Note, as per our discussion in the last problem, you should only be creating *one* instance of your `Adder` and then changing the intputs to this adder to generate the result. Do not create a new `Adder` for each line in the file!

When you're done

Make sure your code runs without giving any error. If it does, comment out the parts that do. Do not submit code that does not run!

Make sure you've properly commented your code. Commenting in Python should feel similar to commenting in Java. You should include:

- A comment header at the top of the file with your name, the date, the assignment number, etc.
- Docstrings for each class (triple quoted strings on the line right after "class ClassName").
- Docstrings for each method (tripe quoted strings on the line right after the "def" line declaring a function).
- Comments separating out each problem.
- Comments interspersed throughout the code where elaboration is required.

When you're ready to submit, upload your assignment via the online submission mechanism. All of your code should be in a single file, `assign5.py`. You may submit as many times as you'd like up until the deadline. We will only grade the most recent submissions.

Grading

Problem 1		
	`OrGate`	1
	`OrGate2`	0.5
	`XorGate2`	1
	`HalfAdder`	1.5
	`FullAdder`	2
Problem 2		
	addition/subtraction	3
	flags (using gates)	3
Problem 3		
	Proper use of `Adder`/setup	3
	Correct output/formatting	3
	style/comments	4
Total		22