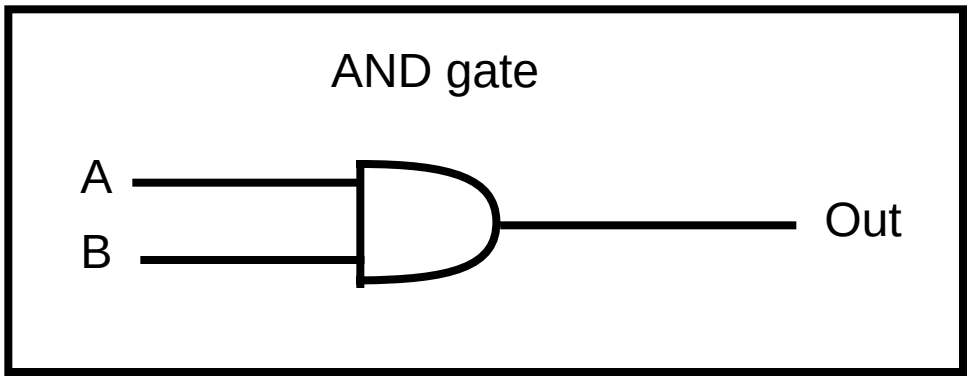
Designing a piece of a larger system

- ***Project:*** Design a circuit simulator
- ***Goal:*** Create a clean interface between the simulator and its client, or calling function

Overview of Circuits

- Circuits are composed of gates
- Inputs have binary values. We can think of them as 1 and 0, or true and false.
- An AND gate returns the logical “and” of its two inputs.
  - In other words, it returns 1 (or true) if and only if both of its inputs are 1 (or true).
  - Here’s the gate:

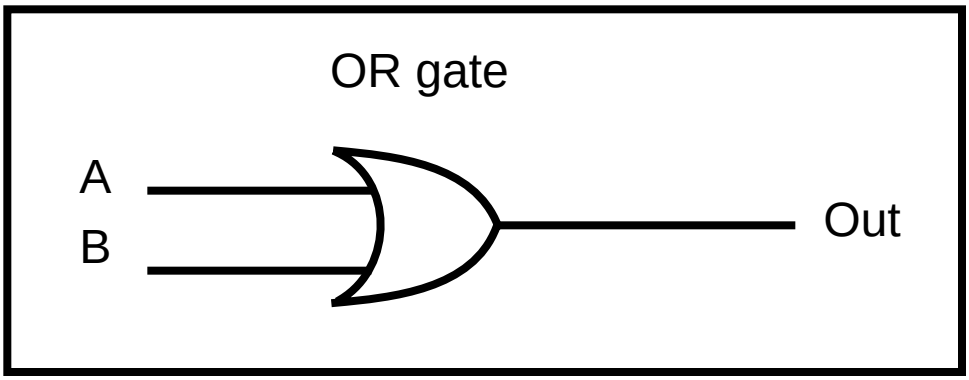


  - This gate implements the following “truth table.” In other words, its output is represented as a function of possible inputs by this table:

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Overview of Circuits (cont.)

- Here's an OR gate, which returns 1 (true) if at least one of its inputs is 1 (true):

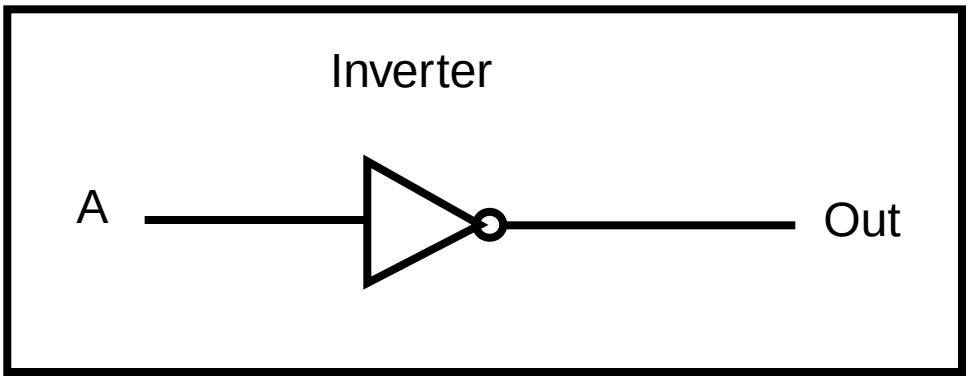


- The function represented by the OR gate can be expressed as follows:

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Overview of Circuits (cont.)

- Finally, here's an inverter, which simply returns the complement of its single input value:



- The function represented by the inverter can be expressed as follows:

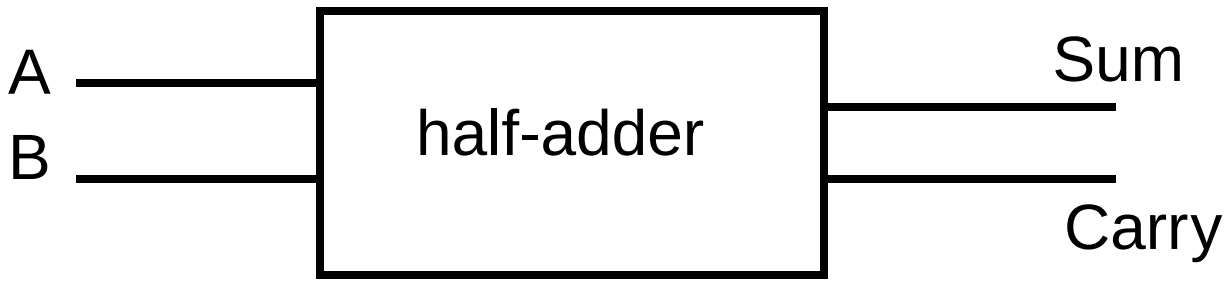
A	Out
0	1
1	0

Combining Gates to form Circuits

- We can combine gates to create more complex functions.
- For example, suppose we want to create a “half-adder” in hardware.
- The goal of the half-adder is to add two numbers in binary.
- For example:

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 0, carry 1
```

- From this example, we can see that we have two inputs (say A and B), and two outputs (the sum and a carry bit)



The Half-Adder Circuit...

- The half-adder circuit implements the following table:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- We can describe the sum and carry outputs in terms of logical expressions:

  - Carry is 1 if and only if A and B are both 1.
    - If we interpret 1 as true, we can turn this into a logical expression:

      Carry = A AND B

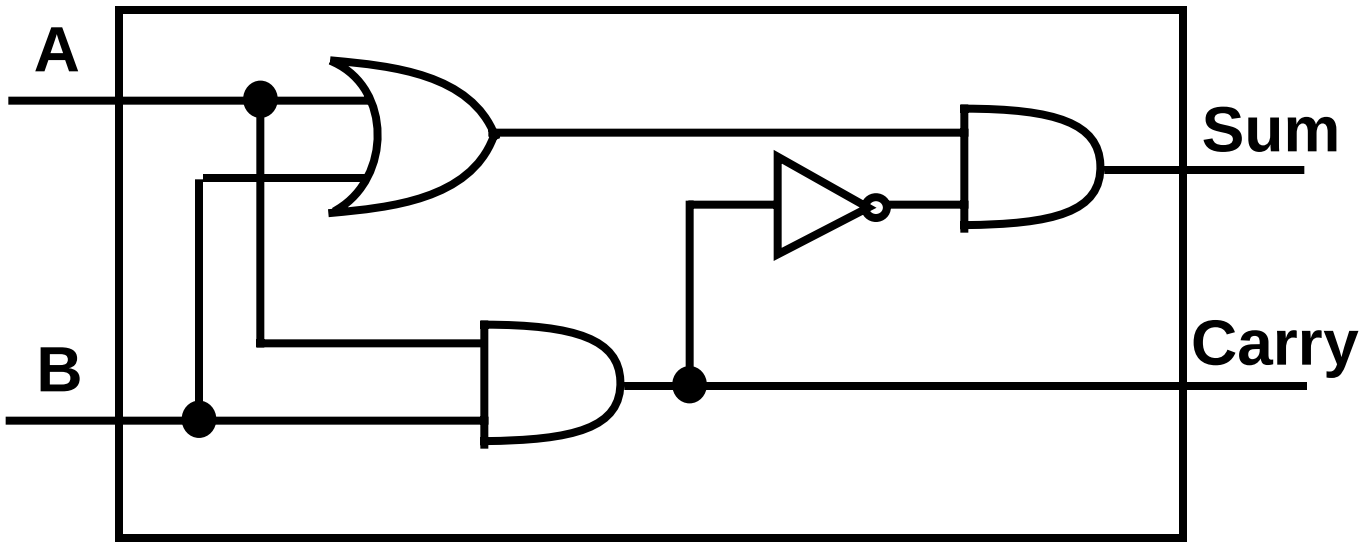
  - Sum is 1 if either A or B is one, and both A and B are not 1
    - A logical expression for this would be:

      Sum = ((A OR B) AND (NOT (A AND B)))

The Half-Adder Circuit...

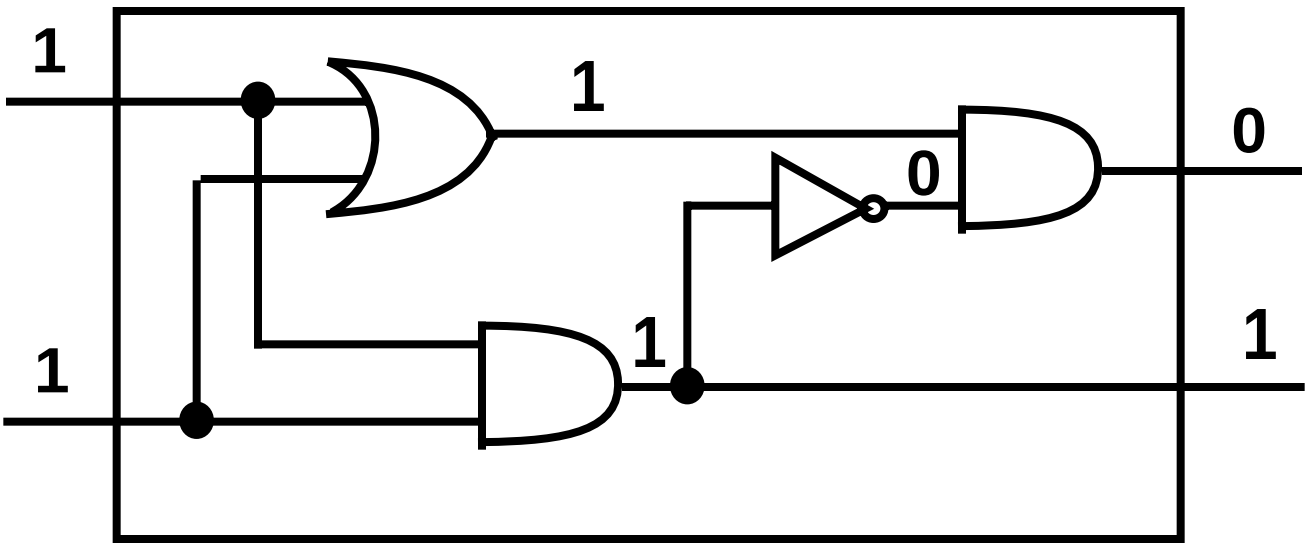
- We can implement these logical expressions directly as sequences of gates to form the half-adder circuit

```
Carry = A AND B
Sum = ((A OR B) AND (NOT (A AND B)))
```



Problem: Design a Circuit Simulator

- Your job is to design a circuit simulator as part of a larger circuit construction program.
- A circuit simulator will compute circuit outputs given a set of connected gates and a set of inputs.
- Example:
  - ***Given the half-adder circuit and inputs A=1 and B=1, your simulator would return sum=0 and carry=1.***



Design Goals

- Because you designing a piece of a larger system, you may encounter many unexpected errors during development and debugging.

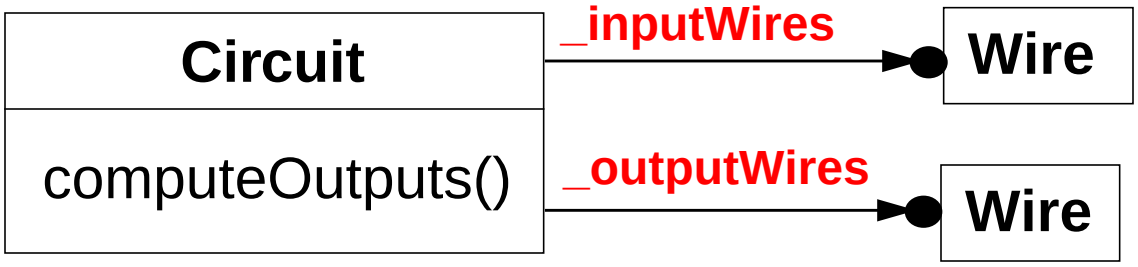
- To minimize problems, think carefully about:

  - ***designing a clean interface to your simulator***

  - ***checking that the simulator is called with legal arguments***

Circuit Object..

- Here is one possible design:

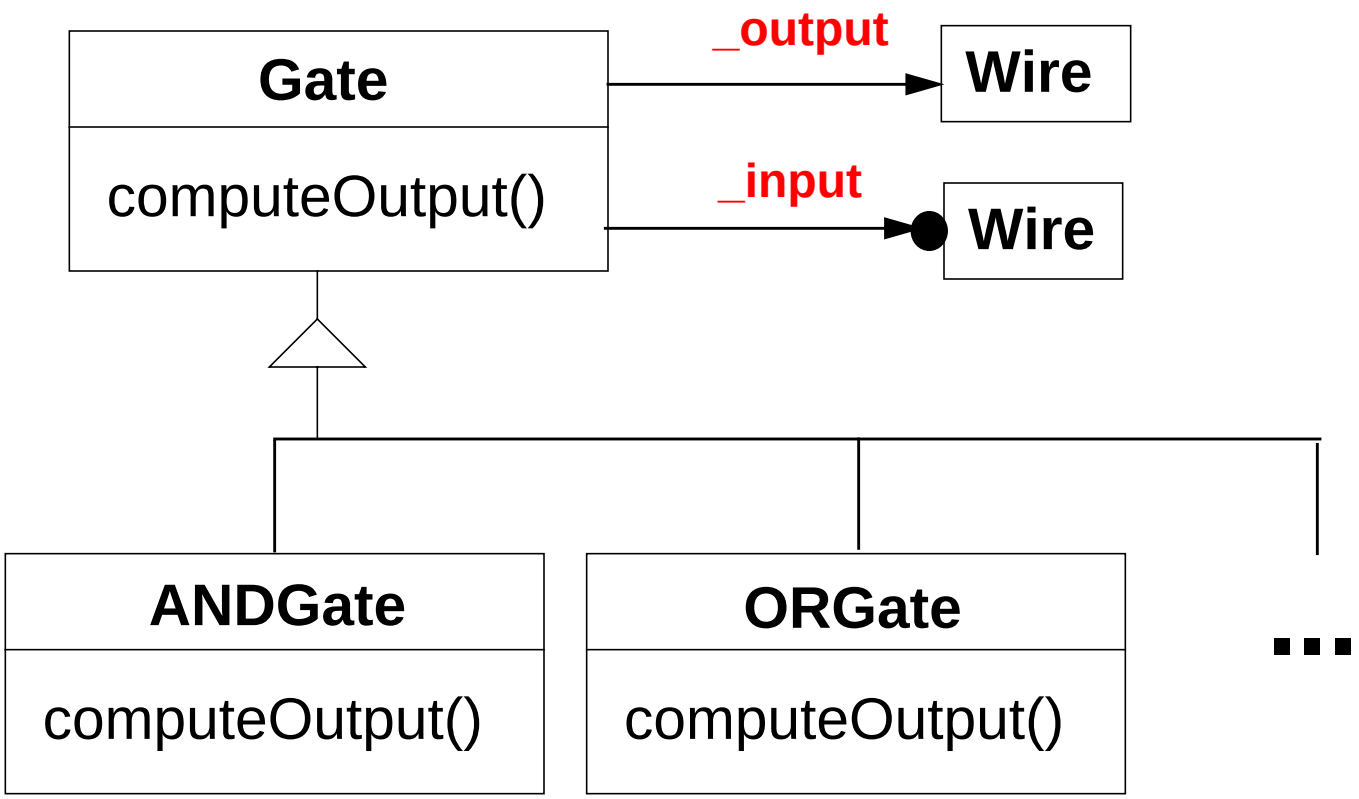


- The simulator is passed a **Circuit** object
- A **Circuit** knows about its **_inputWires** and **_outputWires**.
- These **Wires** provide connections to the rest of the objects that make up the **Circuit**.

Wires

- **Wires** are connections between **Gates**
- A **Wire** knows about the **Gate** it is leaving
- A Wire may enter many **Gates**
- Each **Wire** has a **_value**, 0 or 1, that is computed each time the **Circuit** is simulated.
- It is the responsibility of the circuit simulator to set the **_value** of each of the **_outputWires**

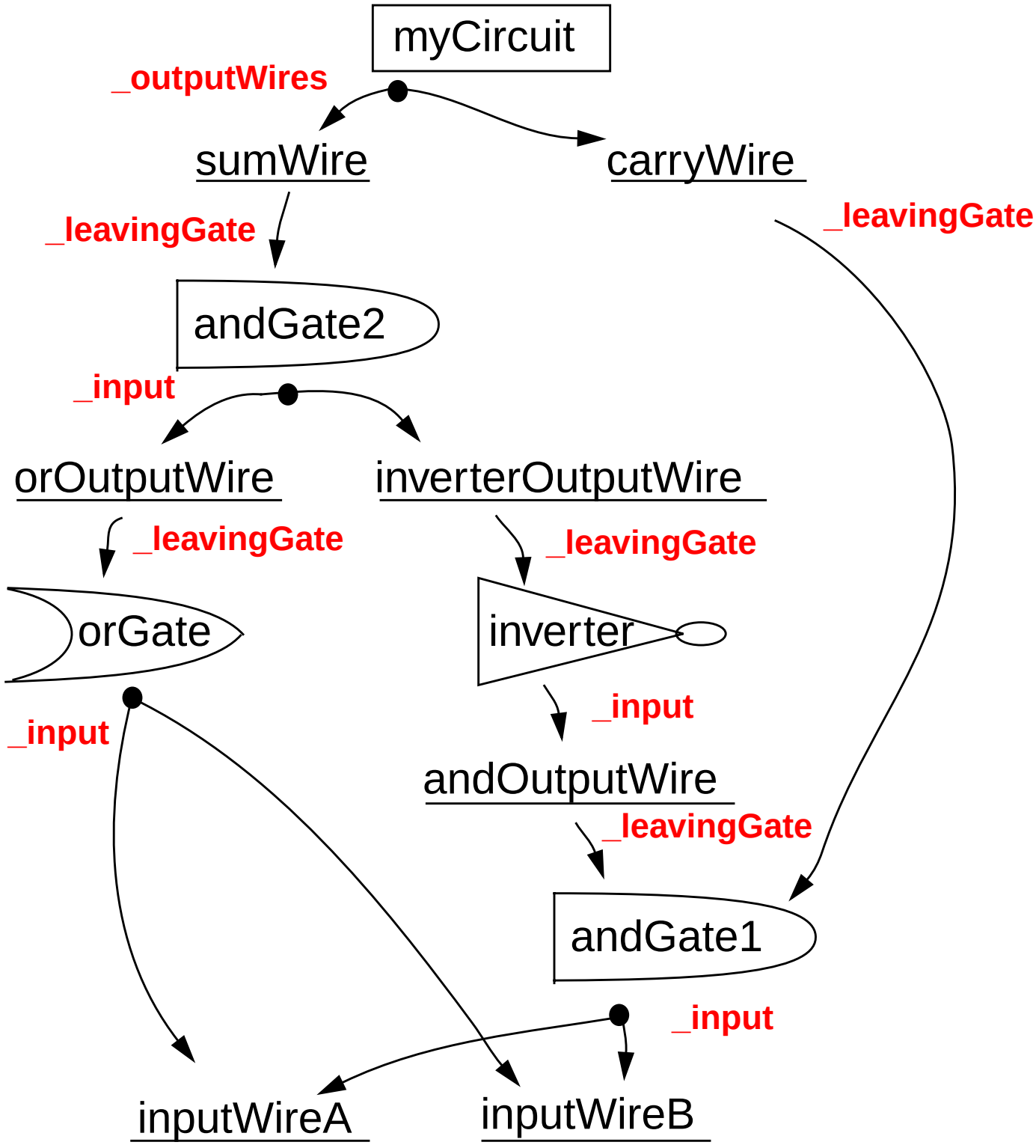
Gate Object..



- A **Gate** knows about its **_output Wire** and a set of **_input Wires.**
- A **Gate** knows how to compute its output value using *computeOutput()*.
- There are many types of **Gates**. In our example, we have **ANDGates**, **ORGates**, and **Inverters**.

Data Structure

- *Here is a portion of the **Circuit** data structure for the half-adder:*



What Gets Passed to the Circuit Simulator?

- We need the **Circuit** object
- To compute the **_values** of the **Circuit** **_outputWires**, we need to know the settings for the **_inputWires**
- One approach would be to require the calling function to set the **_value** variables for the inputs
- Then the Circuit object could simulate itself!

```
// add A and B
inputWireA.setValue(A);
inputWireB.setValue(B);
myCircuit.simulate();
```

Evaluating the Interface

- The calling routine creates a Circuit object that knows how to simulate itself.
  - This is nice, because the calling routine does not have to know about or create a separate simulator object
- The simulate() method assumes that the input values have been set
  - This is a potential trouble spot, because the calling routine could forget to set these values!
  - We need to make sure the Circuit knows how to check that its input values are ok
- The Circuit data structure could be incomplete or flawed
  - We should also add code to check the integrity of a Circuit before we try to simulate it