Com S 227
Fall 2020
Assignment 4
300 points

Due Date: Friday, November 20, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Nov 19)

NO LATE SUBMISSIONS - EVERYTHING MUST BE IN FRIDAY NIGHT

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Canvas.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away! There are likely to be more errors and ambiguities in the spec than in previous assignments, and part of your job is to seek clarification about them.

Check announcements regularly for updates to this document.

Introduction

The purpose of this assignment is to give you some experience

- using interfaces
- reusing code through inheritance ("is-a")
- reusing code through composition ("has-a")

And of course there will be some practice with arrays and lists.

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by your logical organization of classes in this hierarchy and by how effectively you have been able to use inheritance and composition to minimize duplicated code. ***Be sure you have done and understood Lab 8 checkpoint 2!***

Summary of tasks

You will implement the following concrete classes, all of which *directly or indirectly* implement the **`IComponent`** interface:

AndGate
OrGate
NotGate
Multiplexer
HalfAdder
FullAdder
Multiplexer
CompoundComponent
MultiComponent

You'll also implement the two classes implementing the **`IStatefulComponent`** interface:

Register
Counter

In addition to the classes listed above, you will need to implement an abstract class:

AbstractComponent

containing common code for the classes above. *You can also add any additional classes you decide are necessary in order to exploit inheritance to facilitate code reuse.*

That looks like a big list of classes, but you will find that if you do things right there isn't actually a whole lot of code to write!

All your code should be in the **`hw4`** package. The details of what these classes do is discussed further below.

What it's about

The project is a set of classes representing components in a simulation. Each component has an array of *input pins* and an array of *output pins*. Each input or output value is just a single integer 0 or 1 (i.e., one "bit"). Each component has its own rules for determining the outputs based on the inputs. As a simple example, we could define a component with two inputs and one output that computes the output according to the rules:

Input 1	Input 0	Output
1	1	1
1	0	0
0	1	0
0	0	0

You may recognize this rule as the truth table for the boolean "and" operator, where 1 represents "true" and 0 represents "false". We refer to this particular component as an "and gate". Naturally, you could implement the "or" and "not" operations too.

More interesting components can be built up by connecting outputs from one component to inputs of another. A set of such connected components is called a *binary circuit* or *digital circuit*. You could think of these connections as levers, pipes, or wires or in many other ways. (There exist successful implementations using Legos, water pipes, electric relays, Minecraft, and most importantly, the tiny electronic switches called *transistors*. An and-gate can be implemented with three transistors; your laptop's processor contains roughly 1.7 billion transistors.)

Here is another illustration that might be helpful.

Input 1	Input 0	Output 1	Output 0
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

The table above defines a circuit with two inputs and two outputs. Clearly the column labeled "Output 1" is just the "and" operation. The column labeled "Output 0" is similar to "or", except that the first row is zero. This can be expressed as "A or B and not (A and B)". You can picture the thing as a set of connected components as in the illustration below, in which each component is "and", "or" or "not". If you start with inputs of 0 or 1 on the left, you can follow the values through the connections to the outputs on the right, and check that it results in the values from the table above. This leads to an important idea: in order to get the outputs, you have to start with the inputs you want and "propagate" them through the components.

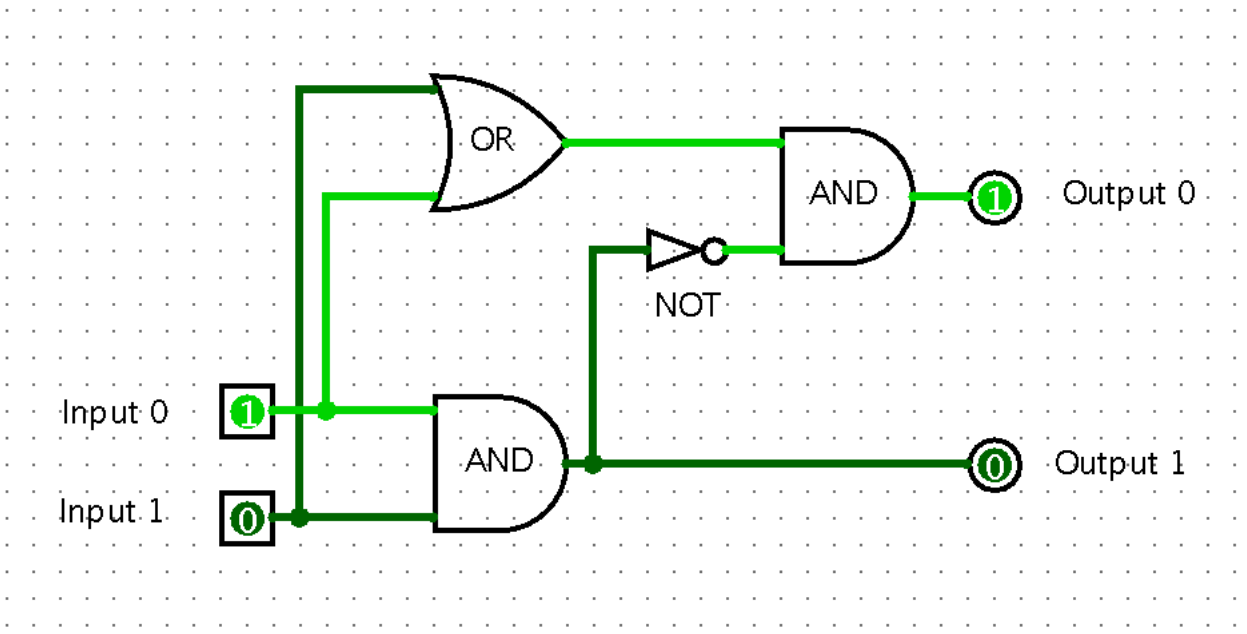


Figure 1 - a "half adder" circuit composed of AND, OR, and NOT gates

The shapes used for the components labeled AND, OR, and NOT in the illustration above are not important, but they are commonly used in drawing circuits like this. Also note that we don't have to implement this particular circuit in this way, and in fact we could write Java code to directly produce the outputs. This is just an example to illustrate how simpler components can be connected together to make more complex components.

More detailed specification

To model the inputs and outputs of the components and the connections between them, we provide the class **`Pin`**. A **`Pin`** contains a *value* (0 or 1) and is either *valid* or *invalid*. A **`Pin`** also has a list of *destinations*, which are other Pins to which it is connected. Whenever the value of an **`Pin`** is set using the **`set()`** method, the destinations connected to it are also set (see the **`set()`** method in the **`Pin`** code). A component is represented by an interface **`IComponent`**. The interface specifies accessors **`inputs()`** and **`outputs()`** that return the arrays of input Pins and output Pins, respectively. The most interesting method of **`IComponent`** is **`propagate()`**. The idea is to model the "flow" of input values to output values. And values of those outputs may change the inputs of other components, and so on. For a given component, when the inputs are valid, a call to **`propagate()`** will set the outputs according to the inputs, based on the desired behavior of the component. As a simple example, here is a possible implementation of the **`propagate()`** method for an "and gate" component:

```
int newValue = 0;
if (inputs()[0].getValue() == 1 && inputs()[1].getValue() == 1)
{
  newValue = 1;
}
outputs()[0].set(newValue);
```

In general to model the behavior of a circuit, we would first *invalidate* all component outputs and inputs except for a small set of "source" components (which could be components with valid externally set values, or special components, described later, that have their own internal state). Then starting with the source components, we repeatedly **`propagate()`** until all components have valid outputs.

There is a sample implementation of **`SampleAndGate`** that you can read as a starting point. Here is a simple usage example of the **`SampleAndGate`**. You can find this code in the **`sample`** package.

```
public static void main(String[] args)
{
  SampleAndGate c = new SampleAndGate();
  Util.setInputs(c, "11");
  c.propagate();
  System.out.println(Util.toString(c.outputs())); // prints "1"
  Util.setInputs(c, "01");
  c.propagate();
  System.out.println(Util.toString(c.outputs())); // prints "0"
  c.invalidateOutputs();
  System.out.println(Util.toString(c.outputs())); // prints "-"
}
```

Here we are making use of a class **`Util`** that you can find in the **`api`** package. It includes some convenient static methods for setting and printing inputs and outputs. There are two things to note that may not be obvious:

- When we specify inputs or print outputs as a string of 0's and 1's, or arrays of 0's and 1's, we *always* start counting from the *rightmost* character in the string. Thus in **`setInputs(c, "01")`**, the '1' in the string becomes the input with index 0, and the '0' becomes the input with index 1.

- If a pin is invalid, the **`Util.toString`** method displays it as a dash character '-'.

Compound components

It's convenient to define a type of component that is a *container* for other components that are connected together in various ways. An example is the circuit known as a "half adder" shown in Figure 1. For this purpose we specify a type **`CompoundComponent`**. Basically it is an implementation of **`IComponent`** that contains a list of related components. You would most likely make it an extension of your **`AbstractComponent`** class. The most interesting part is that you need to override **`propagate()`** so that it will eventually propagate the input values to the outputs, regardless of the internal connections between the contained components. There are many ways to do this. The simplest (though not the most efficient) is to just to loop through the components multiple times. Whenever you find a subcomponent with valid inputs, call **`propagate()`** on it. The loop ends when the **`CompoundComponent`** itself has valid outputs. There is an illustration of this process on the last few pages of this document.

Here is one way to implement the half adder as shown in Figure 1.

```
public class HalfAdder extends CompoundComponent {

  public HalfAdder() {
    super(2, 2);

    // create the contained components and add them to the list
    IComponent andGate = new AndGate();
    IComponent andGate2 = new AndGate();
    IComponent orGate = new OrGate();
    IComponent notGate = new NotGate();
    addComponent(andGate);
    addComponent(andGate2);
    addComponent(orGate);
    addComponent(notGate);

    // wire inputs
    inputs()[0].connectTo(andGate.inputs()[0]);
    inputs()[1].connectTo(andGate.inputs()[1]);
    inputs()[0].connectTo(orGate.inputs()[0]);
    inputs()[1].connectTo(orGate.inputs()[1]);

    // wiring to compute (A or B) and (not (A and B))
    orGate.outputs()[0].connectTo(andGate2.inputs()[0]);
    andGate.outputs()[0].connectTo(notGate.inputs()[0]);
    notGate.outputs()[0].connectTo(andGate2.inputs()[1]);

    // wire outputs:
    // output[0] is the "sum", which is the output of second and-gate
    andGate2.outputs()[0].connectTo(outputs()[0]);

    // output[1] is the "carry", output of first and-gate
    andGate.outputs()[0].connectTo(outputs()[1]);
  }
}
```

More details about the required subtypes of IComponent

`AndGate`

- implements IComponent
- has a no-argument constructor; inputs and outputs are initially invalid
- has two inputs and one output whose value is the logical "and" of the inputs

`OrGate`

- implements IComponent
- has a no-argument constructor; inputs and outputs are initially invalid
- has two inputs and one output whose value is the logical "or" of the inputs

`NotGate`

- implements IComponent
- has a no-argument constructor; inputs and outputs are initially invalid
- has one input and one output whose value is the logical "not" of the input

`HalfAdder`

- implements IComponent
- has a no-argument constructor; inputs and outputs are initially invalid
- has two inputs and two outputs according to the table on page 3
- *Note:* One possible implementation of a half adder comes from wiring together the gates shown in Figure 1. You don't have to do it this way, but it would be a good way to test your CompoundComponent (below).

`FullAdder`

- implements IComponent
- has a no-argument constructor; inputs and outputs are initially invalid
- has three inputs and two outputs according to the table below:

Input 2	Input 1	Input 0	Output 1	Output 0
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

Note: One possible implementation of a full adder comes from wiring together two half adders along with an or-gate. (Again, there is no requirement for you to do it this way, but it would be a good test of your **`CompoundComponent`** class.)

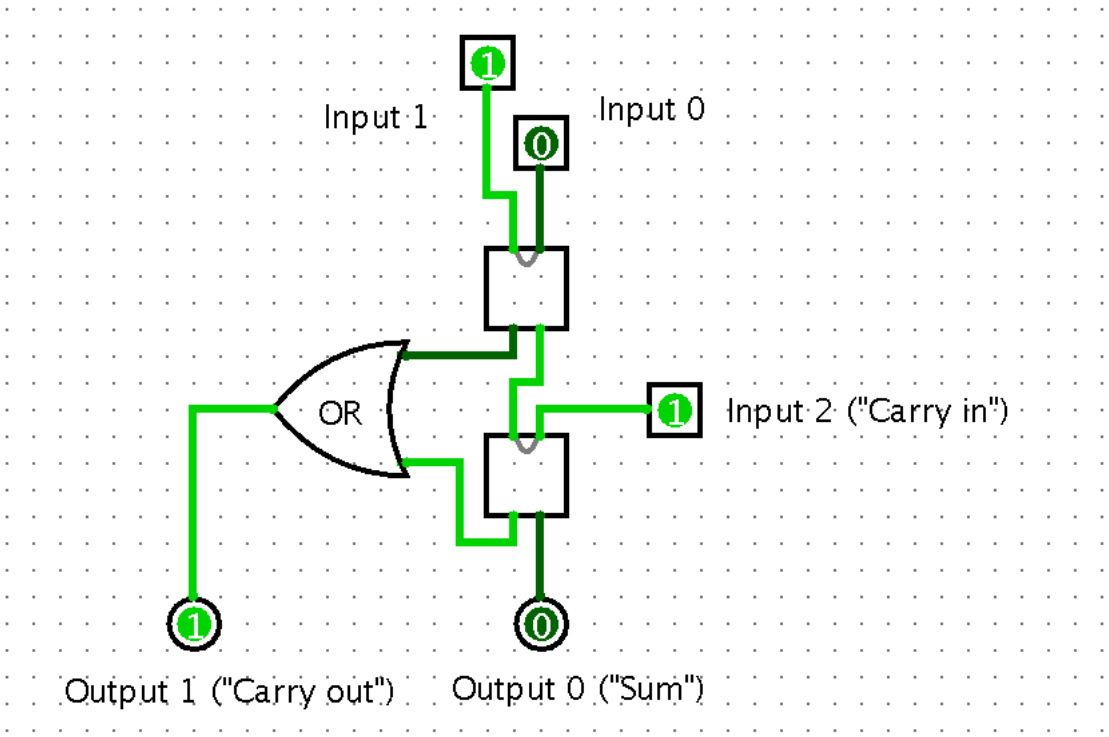


Figure 2 - Implementing a full adder with two half adders

`Multiplexer`

- implements IComponent
- has a constructor with one argument k >= 1; inputs and outputs are initially invalid
- has a total of $2^k + k$ inputs and one output
- the output is equal to one of the inputs, selected from the first 2^k inputs based on the value of the last k inputs, interpreted as a binary number.

  For example, if k is 3 then there are eleven inputs. The last three determine which of the first 8 inputs will be propagated to the output. If the last three inputs are 0, 1, 1, then the output is the value of the input at index 6, since "110" represents the number 6 in binary. In practice, if `arr` is the array of input Pins, you could call `Util.toIntValue(arr, 8, 10)` to convert inputs 8 through 10 into an integer value.

  If you are completely unfamiliar with the binary representation of numbers, you could see Chapter 12 (page 94) of this book from Com S 127: http://web.cs.iastate.edu/~smkautz/cs127f15/notes/Eels_v0.1.pdf)

`CompoundComponent`

- implements IComponent
- has a constructor with two arguments, the number of inputs and the number of outputs; inputs and outputs are initially invalid

- has a method **`public void addComponent(IComponent c)`** that adds component c to this **`CompoundComponent`**'s list of components
- has a method **`public ArrayList<IComponent> getComponents()`** that returns a reference to the list of components
- *Note*: See the preceding section ("Compound components"). This class is not expected to work with stateful components (which would require a more sophisticated implementation of **`propagate()`**).

`MultiComponent`

- implements IComponent
- has one constructor **`public MultiComponent(IComponent[] components)`**; inputs and outputs are initially invalid
- the constructor argument is an n-element array of identical components with m inputs and 1 output; the MultiComponent has n * m inputs and n outputs.
- for each i < n, inputs i * m up to (i + 1) * m are connected to the i-th component in the array, and the output of that component is connected to output i.
- *Tip*: is a **`MultiComponent`** a special kind of **`CompoundComponent`**?

`AbstractComponent`

- implements IComponent
- There are no specific requirements. Design this class as you see fit to minimize duplicated code among the concrete types above. Note that since the instance variables have to be **`private`** (See the "Special style requirement" below) you will almost certainly need a **`protected`** constructor with (at least) parameters for the number of inputs and the number of outputs.

Stateful components

In addition to the examples above, you'll implement two classes implementing the subinterface **`IStatefulComponent`**: **`Register`** and **`Counter`**.

The examples such as the and-gate and half adder described above are "stateless" in the sense that the outputs are always determined by propagating the inputs. A *stateful* component has an internal state that determines the outputs. The inputs may affect the internal state, but generally do not propagate directly to the outputs. The state can be modified by three methods, as specified in the interface **`IStatefulComponent:`**

```
public void tick()
public void setEnabled(boolean enabled)
public void clear()
```

One example is a *register*. A register's state consists of n bits, for some n. It has n inputs and n outputs. The outputs are always valid and equal to the state. It can be enabled or disabled. If enabled, the **`tick()`** method causes the input bits to be copied to the state (provided that the inputs are valid). The **`clear()`** method causes the state to become all zeros (whether enabled or not).

Here is a simple usage example of a register:

```
Register reg = new Register(3);
Util.setInputs(reg, "011");
reg.setEnabled(true);
reg.tick();
System.out.println(Util.toString(reg.outputs())); // 011
reg.setEnabled(false);
Util.setInputs(reg, "100");
reg.tick();
System.out.println(Util.toString(reg.outputs())); // still 011
reg.setEnabled(true);
reg.tick();
System.out.println(Util.toString(reg.outputs())); // 100
reg.clear();
System.out.println(Util.toString(reg.outputs())); // 000
```

Note that it might be useful for a **`Register`** to extend your **`AbstractComponent`** class *and* implement the **`IStatefulComponent`** interface. You could use a declaration such as

```
public class Register extends AbstractComponent implements IStatefulComponent
```

Another example of a stateful component is a *counter*. A counter's state consists of n bits, for some n. It has no inputs and n outputs. The outputs are always valid and equal to the state. It can be enabled or disabled. The state is normally thought of as an integer in binary notation. If the counter is enabled, the **`tick()`** method causes the state to increase by 1. Initially the counter is at zero and the **`clear()`** method resets it to zero (whether enabled or not). The value "wraps around" to zero when the maximum value is reached. (For example, for a counter with two bits, successive calls to **`tick()`** would cause the state to cycle through the values "00", "01", "10", "11", "00", "01", and so on.)

A third example is an *externally set value*. We could think of this as a value determined by flipping a bunch of switches or reading from an input device. The **`tick()`** and **`setEnabled()`** methods have no effect.

There is an implementation of an externally set value in the **`api`** package called **`ExternalValue`**. You'll provide implementations of classes **`Register`** and **`Counter`** as described above. Each of

these should have a constructor with one argument n, the number of internal bits. A newly constructed Counter or Register should be disabled by default.

IPinListener and Probe

These are two classes that are already implemented that you might find useful for experimentation and testing. The idea is to be able to automatically update some observable output (the console or some kind of UI) whenever the value of a component's Pins changes. The IPinListener interface has one method: void **`update(IComponent c)`**. A Pin maintains a list of IPinListener objects, and whenever the Pin's state changes, it invokes the **`update()`** method for all listeners. The argument passed to that method is always the parent component for that Pin. A Probe is a simple implementation of the IPinListener interface whose update() method just prints the current state of the parent component's outputs. There is a method addListener() in the Util class that connects a listener to all Pins of a given component's outputs.

As an example, here is how the test of the sample and-gate could be implemented using an ExternalValue to set inputs and a Probe to display the outputs:

```
SampleAndGate c = new SampleAndGate();
ExternalValue ex = new ExternalValue(2);
Util.connect(ex, c);
Util.addListener(c, new Probe("Test SampleAndGate"));
ex.setValues("11");
ex.propagate();
c.propagate();
ex.setValues("01");
ex.propagate();
c.propagate();
c.invalidateOutputs();
```

Note, you are not required to use **`IListener`** or **`Probe`** for any part of your code, they are just there in case they are useful for testing or experimentation.

Importing the sample code

1. Download the zip file to a location outside your workspace. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

Alternate procedure: If for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Unzip the zip file containing the sample code.
2. In Windows File Explorer or OS X Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Package Explorer, navigate to the src folder of the new project.
5. Drag the **`api`** and **`example`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

Testing and the SpecChecker

As always, you should try to work incrementally and write simple tests for your code as you develop it.

Since test code that you write is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests.

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you. It will attempt to package up all the classes in your hw4 directory. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** (or possibly **`protected`**) as well.

Style and documentation

You may not use **`public`**, **`protected`** or package-private instance variables. Normally, instance variables in a superclass should be initialized by an appropriately defined superclass constructor. You can create additional **`protected`** getter/setter methods if you really need them.

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
    - **When a class implements or overrides a method that is already documented in the supertype (interface or class) you normally do not need to provide additional Javadoc,** unless you are significantly changing the behavior from the description in the supertype. You should include the **`@Override`** annotation to make it clear that the method was specified in the supertype.

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`assignment4`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`assignment4`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "tt" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Suggestions for getting started

General advice about using inheritance: A really good way to get started is to forget about inheritance. That is, pick one of the required concrete types, such as **`OrGate`**, declare it to implement **`IComponent`**, and just write the code for all the specified methods (possibly using **`SampleAndGate`** as a guide). Then, choose another required type, such as **`NotGate`**, and write all the code for that one. At this point you'll notice that you had to write some of the same code twice. Now you can implement the class **`AbstractComponent`** to contain the common code.

Be sure you have done and understood Lab 8, checkpoint 2!

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains **at least** the following eleven files:

AbstractComponent.java
AndGate.java
OrGate.java
NotGate.java
Multiplexer.java
HalfAdder.java
FullAdder.java
CompoundComponent.java
MultiComponent.java
Register.java
Counter.java

The specchecker should automatically zip up everything in your hw4 package. ***If you have defined additional files as part of your inheritance hierarchy, check to be sure those additional files are included too.*** Please do not include unrelated or unnecessary files. (E.g. your example code or test code should not be in the hw4 package).

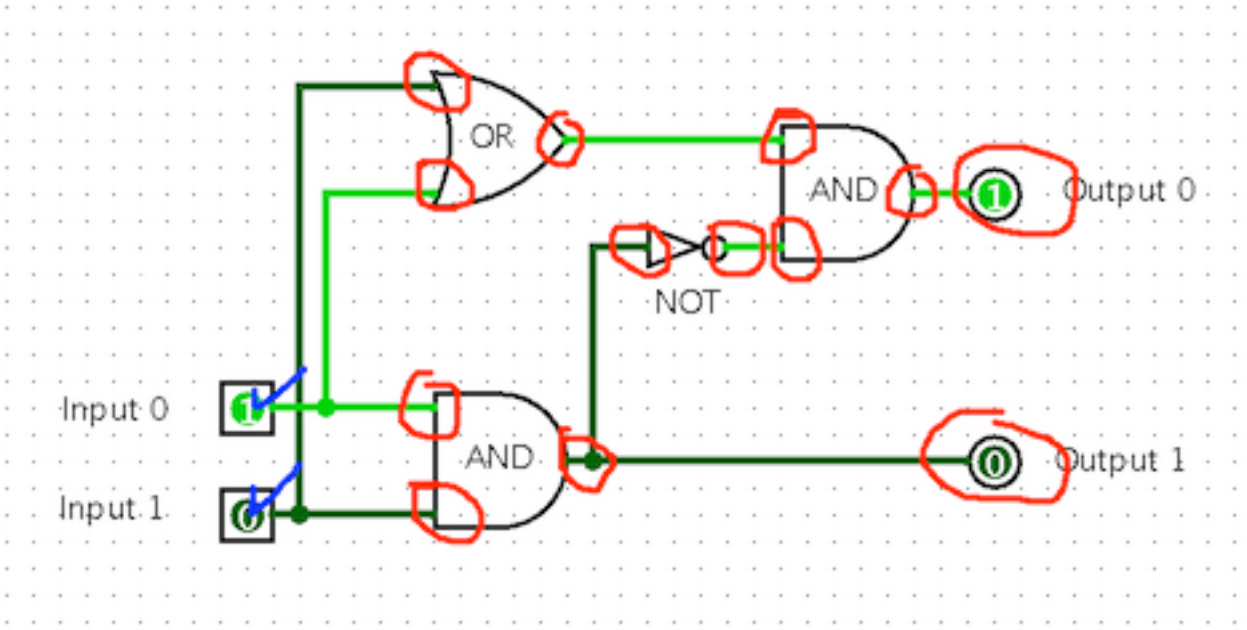
Please LOOK at the archive you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 4 submission link and **verify** that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain the files listed above. Make sure you are turning in **`.java`** files, not the **`.class`** files. You can accomplish this easily by zipping up the **src** directory of your project. (The zip file will include the other **`.java`** files in the project, but that is ok.) The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.

Example: what propagate needs to do on a compound component

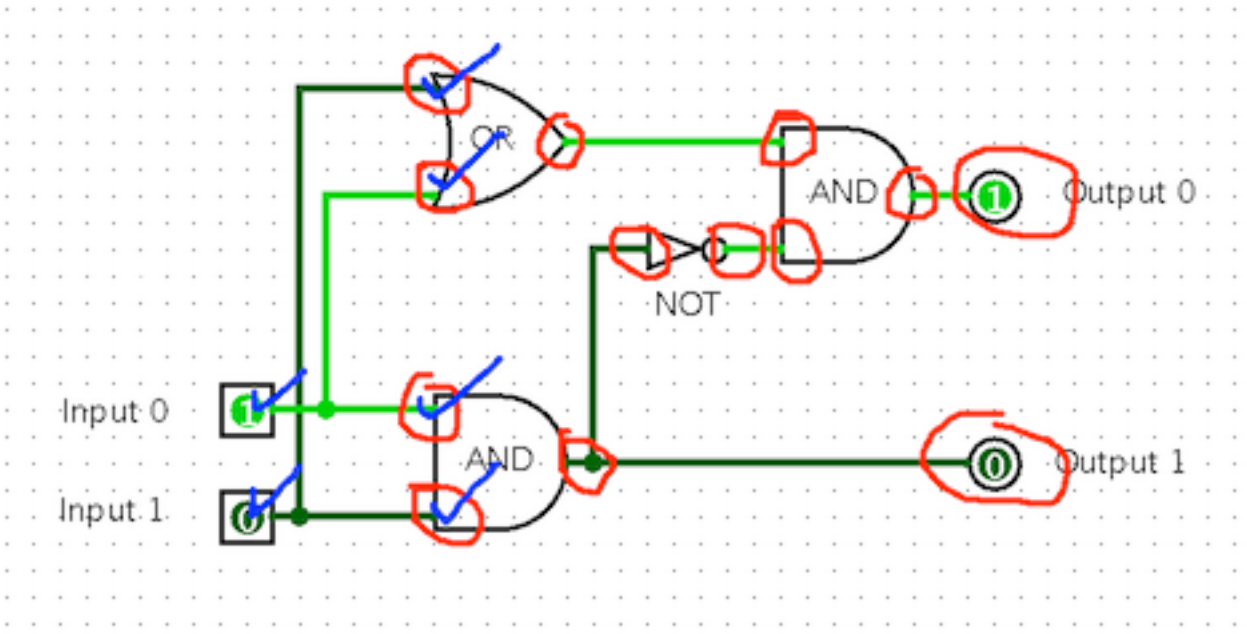
As an example, consider the "half adder" from Figure 1. Assume that the inputs are valid. What needs to happen to propagate those values through all subcomponents to the outputs? We start by invalidating the outputs, as well as all inputs and outputs of the subcomponents. Suppose we mark the valid pins with a blue check, and the invalid ones with a red circle, to get a picture like this.



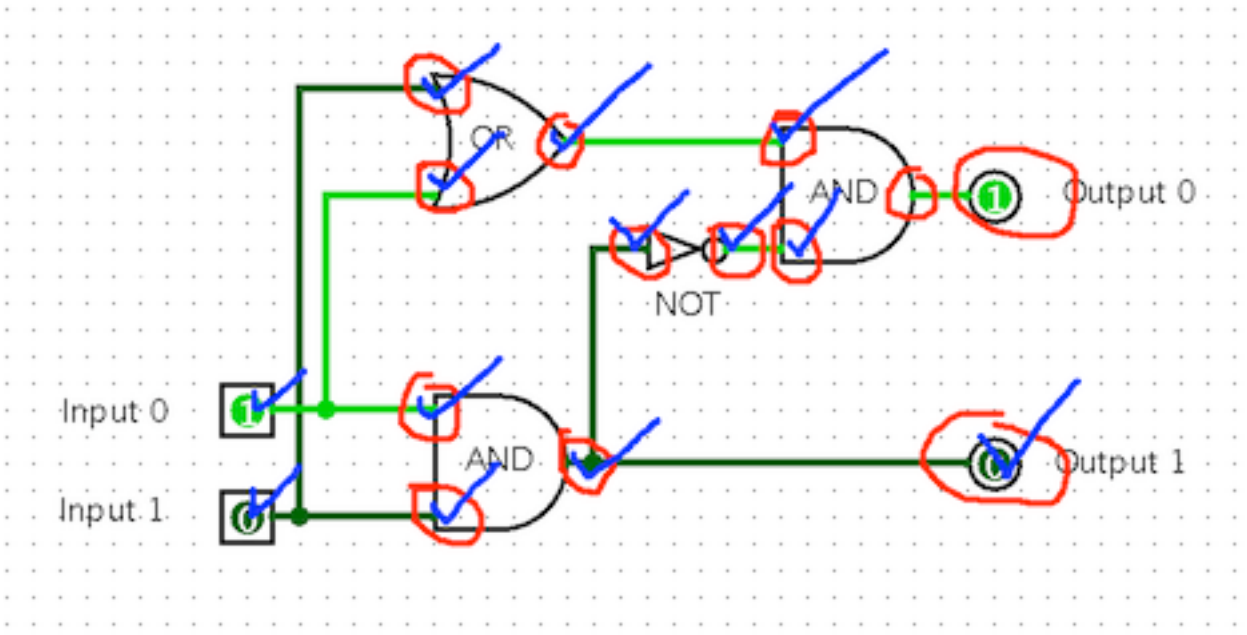
First, the pins directly connected to the main inputs have to become valid. So we first call

```
p.set(p.getValue())
```

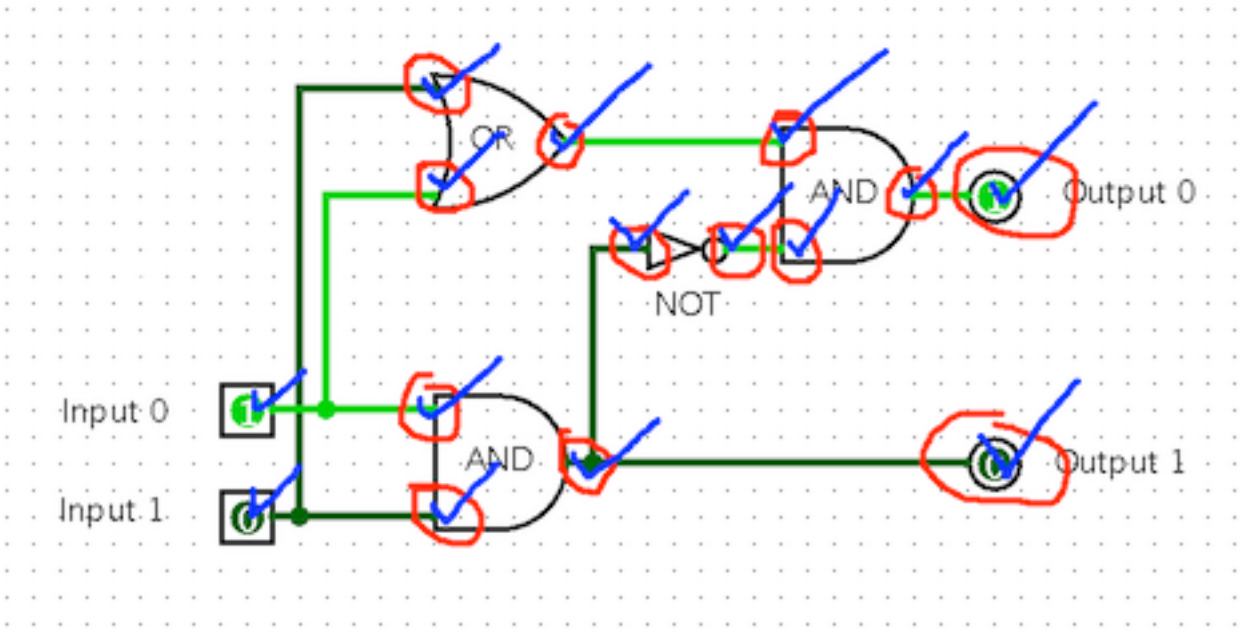
on each input pin, which causes `set()` to be called on each *connected* pin. Now the inputs of the first or-gate and and-gate are valid:



Now, begin iterating over the subcomponents and call **`propagate()`** on each one. If the order of components happens to match that of the sample code on page 6, we'd first call propagate() on the left and-gate. Its output becomes valid, which in turn sets the input of the not-gate as well as output 1. Then, calling propagate() on the second and-gate does nothing, because its inputs aren't all valid. Iterating to the or-gate and the not-gate finally results in this:



Now, we have reached the end of the list of components, but not all outputs are valid, so we iterate again. This time, the second and-gate's inputs *are* valid, so propagate() causes its output to be set, causing the main output 0 to be set.



Since the compound component's outputs are valid now, propagate() is finished.

> *Note:* You might notice some inefficiency built into this strategy, since we may iterate over the entire list of components even when almost all of them already have valid outputs. There are some clever things one could do to improve it, e.g., by keeping track of components that do not need to be visited since they are already known to have valid outputs, and components that do not need to be visited since it is known that none of their inputs is valid yet. Think about this problem again after you learn *breadth-first search* in 228.