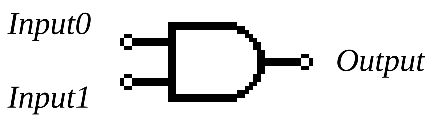
INB/INN 374 JavaBeans Assignment

Due: Friday 5th September 2014
Worth: 20%
Teams: 1 or 2 students per team

In this assignment you will develop components for a CAD (Computer Assisted Design) tool for designing and testing digital circuits. Digital circuits consist of a set of connected digital components. Each digital component takes zero or move inputs and produces zero or more outputs. The input and output values are either true or false. A typical digital component is an AND gate which takes two inputs and produces one output which is true if and only if both input signals are true:



Digital circuits can be implemented electronically (where voltages are used to represent values of true and false) and are the basis for modern electronic computers. This physical hardware world also uses the idea of components with simpler hardware components being used to constructor more complex hardware components.

Please read the submission instructions at the end of this document before you start coding.

Part A – OutputTerminal (1 mark)

Note: This assignment must be developed using the NetBeans IDE (http://netbeans.org/downloads/)

A Java class library called **`ProvidedCircuitComponents.jar`** has been provided to you along with the assignment specification on blackboard. You will need to download it and use it to help complete your assignment.

Create a new Java Class Library project called **`DigitalCircuits`**. The Project Location should be a folder on your local hard drive, creating it on a network drive will cause problems. To make your libraries easier to manage, tick the box that says "Use Dedicated Folder for Storing Libraries, and set the Libraries Folder to **`..\SharedLibraries`**.
Right click on Libraries tab within the Projects window, select **`Add JAR/Folder ...`** and browse to where you have saved the **`ProvidedCircuitComponents.jar`** file. This library contains the definition for an interface called **`Terminal`** which is used to connect together digital components by allowing a true/false value to be passed from one component to the next:

```
package Digital;

public interface Terminal
{
    boolean getValue();
    void addPropertyChangeListener(java.beans.PropertyChangeListener l);
    void removePropertyChangeListener(java.beans.PropertyChangeListener l);
}
```

As you can see, the **`Terminal`** interface defines a Java Bean property called **`Value`** of type **`boolean`**.

Your first task is to create a class which provides a concrete implementation of this interface. Add a new Java Package called **`Digital`** to the Source Packages for your project and add a new Java Class called **`OutputTerminal`** that implements the **`Terminal`** interface shown above. In addition to the methods in the **`Terminal`** interface, your **`OutputTerminal`** class should implement a *setter* method for the **`Value`** property. The **`Value`** property is intended to be a *bound* property which means that a Property Change event is meant to be fired whenever the value of the property changes (see http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html). Note: the **`getValue()`** method should be public, but the field that you use to hold the value of that property should be private. Fields should never be public.

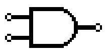
OutputTerminal (Unit Testing) (1 mark)

Once you think you have OutputTerminal implemented correctly, right click on **`OutputTerminal.java`** in the Project window and select **`Tools`**, **`Create JUnit Tests`**. Replace the auto-generated test by the code in the **`OutputTerminalTest.java`** file provided on blackboard. Run the test and debug until all unit tests pass. Run these unit tests again any time you make any changes.

Part B – AND Gate (Visual Appearance) (1 mark)

Your next task is to implement a Java class to represent an AND gate. This Java class will have a visual interface, however we will not need to design it using the NetBeans WYSIWYG editors so, simply create a plain Java class named **ANDGate** and then change its definition so that it ***extends*** **javax.swing.JPanel**.

To give the AND gate its physical appearance we will load and paint a suitable image file. Firstly create a new Java Package **Digital.images** and inside that folder add the following image (provided as "**AND.gif**" in Zip file provided with assignment specification):



To load this image file at runtime we use the following code:

```
java.net.URL url = getClass().getResource("images/AND.gif");
image = new javax.swing.ImageIcon(url).getImage();
```

This is best done just once in the AND gate constructor and stored in a private field. Make sure your constructor is public. In addition to loading the image, we also want to resize the current component to match the size of the image:

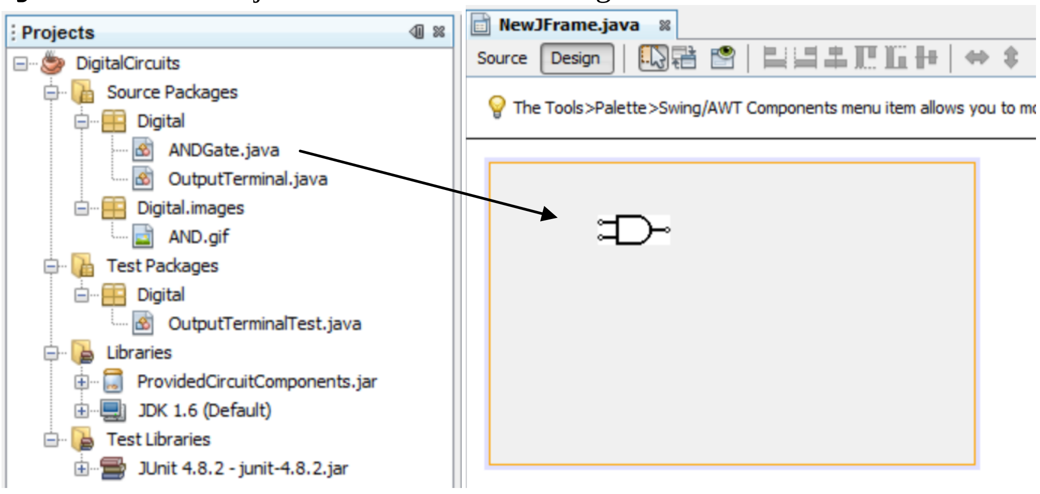
```
this.setSize(image.getWidth(null), image.getHeight(null));
```

To paint the image to the screen we override the **paintComponent** method of our base class:

```
@Override
public void paintComponent(java.awt.Graphics g)
{
    g.drawImage(image, 0, 0, null);
}
```

Make sure your code compiles successfully before continuing.

You can now test the visual properties of this class by creating a **JFrame** form and dragging **ANDGate.java** from the Projects window onto the design surface:



AND Gate (Circuit Logic) (2 marks)

All gate components have a single output terminal represented by an object of type **OutputTerminal**. The **OutputTerminal** object associated with the AND gate will be created when the AND gate object is constructed:

```
private OutputTerminal output = new OutputTerminal();
```

The AND gate will expose this value via a read only public property called **Output** of type **Terminal** (i.e. a *getter* method is required, but there is no need for a *setter* method as the output terminal associated with this AND gate should never change).

The AND gate also has *getter* and *setter* methods for properties `Input0` and `Input1` of type **Terminal**. These *setter* methods are used to connect the AND gate to the **Terminals** of other digital circuit components in order to receive input from those other components.

All digital components (including AND gates) should implement the **PropertyChangeListener** interface:
http://docs.oracle.com/javase/7/docs/api/java/beans/PropertyChangeListener.html

When a digital component receives such a property change event it should call a method to **recompute** its output(s). To compute the new value of the output(s) we will generally need each of the input values. If any of the inputs are not connected to anything (i.e. they have the value *null*) then we will treat them as having the value **false**. Otherwise, we will retrieve the **Value** property of the **Terminal** that is connected to that input. When we recompute our output value we set the **Value** property of our **OutputTerminal**; the value that we set it to will depend on our inputs. In the case of an AND gate, the output will be set to true if and only if both of the input values is true.

The *setter* method for our two input properties actually performs a number of roles:
1. Firstly, it stores the newly provided **Terminal** in a private field for that input property.
2. Next it registers the AND gate (i.e. the ***this*** object) as a property change listener of the provided **Terminal** object. So, whenever, any of our inputs change, we will be automatically notified.
3. Finally, the *setter* method also calls the method to recompute our outputs (as changing one of the input sources may change the currently computed output value).

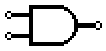
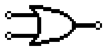
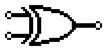
AND Gate (Unit Testing) (1 mark)

Create a JUnit Test for ANDGate.java and drop in the code from the **ANDGateTest.java** source file provided on blackboard. Run the test and debug until all tests pass. Run these tests again any time you make any changes.

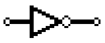
<u>Part C – Other Gates</u> (4 marks)

Now that you have successfully implemented an AND gate, repeat that process for the following other logic gates:

Binary Gates

Name	Icon	`input0 = T` `input1 = T`	`input0 = T` `input1 = F`	`input0 = F` `input1 = T`	`input0 = F` `input1 = F`	Java operator
AND		`True`	`False`	`False`	`False`	`&&`
OR		`True`	`True`	`True`	`False`	`||`
XOR		`False`	`True`	`True`	`False`	`^`

Unary Gates

Name	Icon	`input = T`	`input = F`	Java operator
NOT		`False`	`True`	!
Pin	O	`True`	`False`	no op

You will notice that these implementations are mostly the same, so you should ***refactor*** your code by creating abstract base classes in order to avoid duplicating fields and code. Each of the individual Gate class implementations should only include code that is unique to them, i.e. the name of the image file and the method for recomputing outputs. For example, your refactored **`ANDGate`** class should be as simple as:

```
package Digital;

public class ANDGate extends BinaryGate
{
    public ANDGate()
    {
        super("images/AND.gif");
    }

    @Override
    protected boolean Compute(boolean a, boolean b)
    {
        return a && b;
    }
}
```

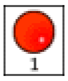
Do not attempt to refactor your code until your **`ANDGate`** passes all tests. Make a copy of your original unadulterated **`ANDGate`** before your start trying to refactor it, as your tutor will likely ask to see that version first if you approach them for help.

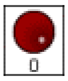
Test and debug as required.

<u>Part D – Input and Output Components</u> (2 marks)

Next we need components that allow us to generate raw input and display the outputs.

First we'll start with output by implementing a new digital **`LED`** component used to display output states. An **`LED`** component has one input and no output terminals. The user interface of the LED is similar to a gate, but there are two images, one to be shown when the input is true and one to be shown when the input is false:





Both images should be loaded (and stored in private fields) when the LED object is constructed and the image displayed should change when the input changes. You will need to call **`repaint()`** whenever a different image needs to be displayed. Calling **`repaint`** will trigger a call to **`paintComponent`** (**`paintComponent`** should never be called directly).

Next we support input by implementing a new digital **`Switch`** component. The user interface of the **`Switch`** class will be constructed by creating a **`JPanel Form`** and using the NetBeans visual designer to construct its user interface. Drag and drop a **`JToggleButton`** (from the Palette of Swing Controls) onto the design surface. In the property window clear the **`text`** property, set the **`margins`** to 0, set the **`rolloverEnabled`** property to **`false`** and set the **`icon`** and **`selectedIcon`** properties to the following provided images respectively:

Resize the Switch control to exactly match the size of the toggle button. Select the **`JToggleButton`** in the designer and use the Events tab in the property window to automatically add an event handler for the **`StateChanged`** event. This event should trigger the Switch to recompute its output. The output value is based on the **`isSelected`** property of the **`JToggleButton`**.

Part E – Testing Gates (2 marks)

The purpose of the JavaBeans component framework is to enable tools like NetBeans to configure and compose third party components without having to write Java source code. This section will demonstrate how this can be done ...

Create a new Java Application Project called **`PartE`** and replace the main class by a new **`JFrame Form`**. Right click inside the Palette window and open the **`Palette Manager ...`** , **`Add from JAR ... ProvidedCircuitComponents.jar`** and add all available components to the palette.

Drag a **`CircuitBoard`** from the Palette Window to the JFrame design window. To avoid having the CircuitBoard obscured by the alignment guidelines provided by the NetBeans Designer, right click on the CircuitBoard and **`Set Layout`** to **`Null Layout`**. Now drag **`Switch.java`** from the Projects window onto the Design window. Drag LED.java from the Projects window onto the Design window.

If these components fail to display correctly on the design surface it is because either:

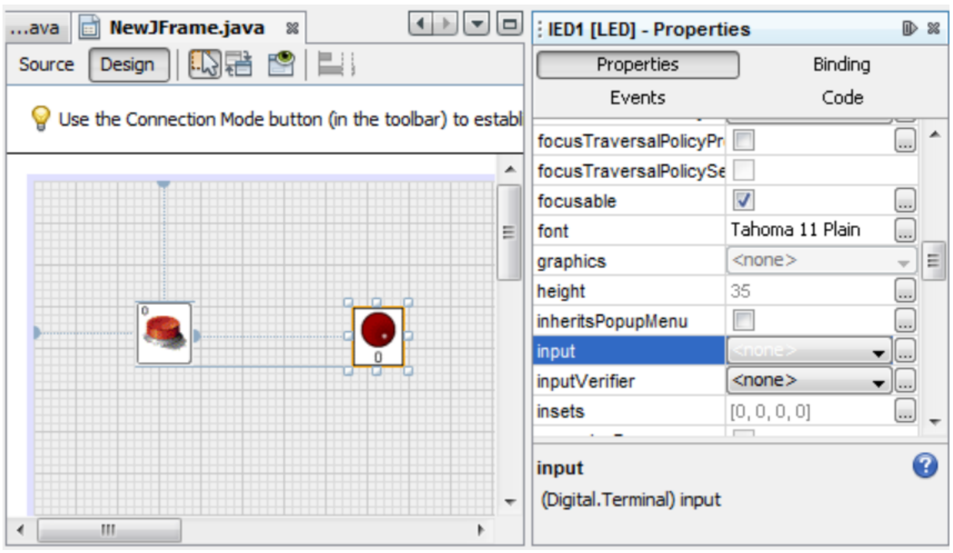
- You have not recompiled since changing your source code.
- Your class or constructor is not public.
- The code in your constructor has thrown an exception due to a programming bug.
- You have changed the source code of components after you have dragged them onto the form. If you find you need to change the implementation of any of the components after you have added them to a form, you may need to completely clear the form, recompile and then add them all again :(

Next, we want to connect the LED to the Switch. If we were doing this directly in Java source code, we would write:

```
led1.setInput(switch1.getOutput());
```

So, we need to do the equivalent of this using the NetBeans Designer and Properties windows ...

Select the LED in the Design window and then in the Properties window click on the ... button to the right of the **`input`** property:

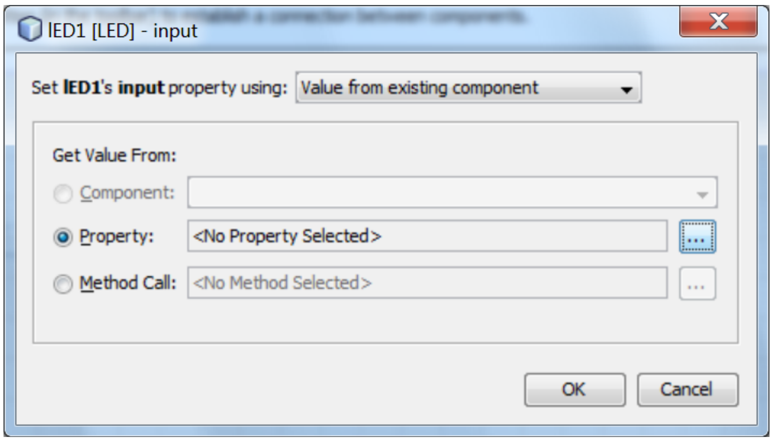


Use the Connection Mode button (in the toolbar) to establ

focusTraversalPolicyPr	☐
focusTraversalPolicySe	☐
focusable	✓
font	Tahoma 11 Plain
graphics	<none>
height	35
inheritsPopupMenu	☐
input	<none>
inputVerifier	<none>
insets	[0, 0, 0, 0]

input

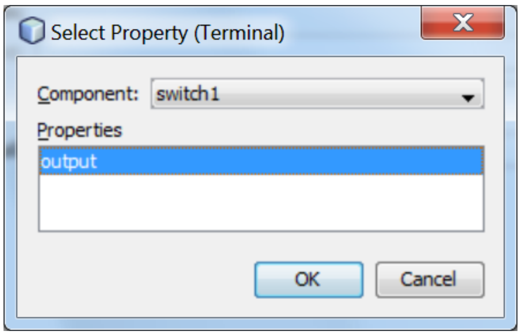
(Digital.Terminal) input


IED1 [LED] - input

Set **lED1**'s **input** property using: Value from existing component

Get Value From:

- Component:
- Property: <No Property Selected>
- Method Call: <No Method Selected>

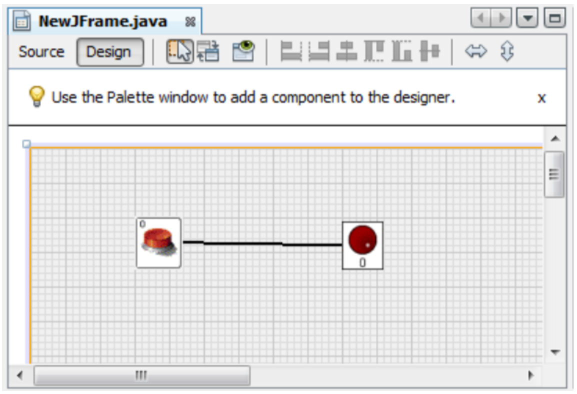

Select Property (Terminal)

Component: switch1

Properties

output

Note that the provided **`CircuitBoard`** background component has been specially designed to display this logical connection visually as a black (or red) line representing a wire in the circuit:



Use the Palette window to add a component to the designer.

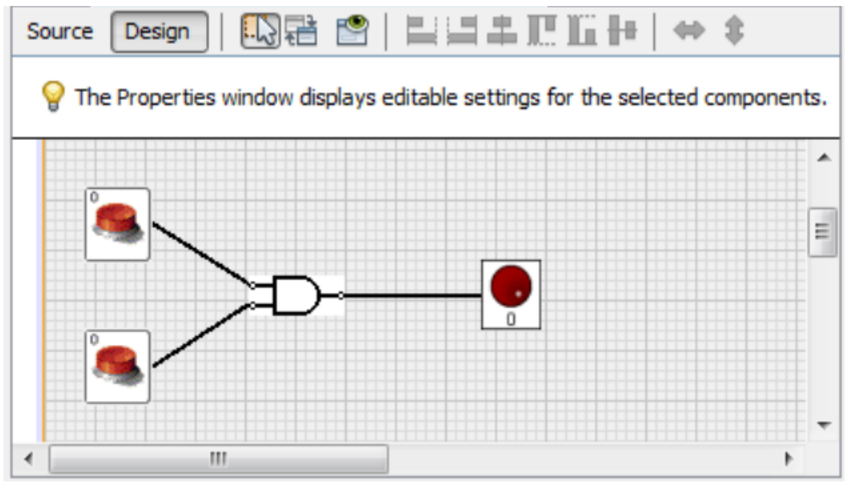
If this line fails to appear it is because either:

- Your input or output properties do not follow the expected naming convention.
- Your input or output properties are not public.
- You have a public field that has the same name as a property.
- If the line appears when you run the program but not in the designer, try closing the design window, recompiling and then reopening.

You can now run this application and test it by clicking on the Switch. The LED should change each time the Switch is clicked.

You may find when you run the application that an extra margin has been added around the toggle button. This is caused by the Nimbus look and feel manager. To avoid this, switch from the Design view to the source view and manually delete the Look and feel setting code in the main method.

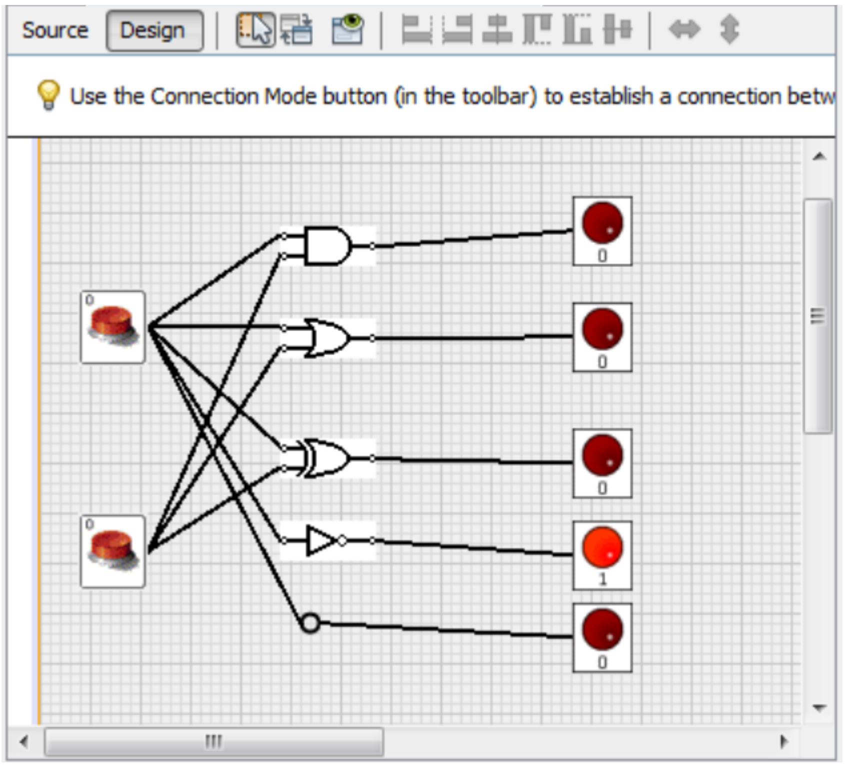
Next, we will test the ANDGate by adding the following connected components to the JFrame form of **`PartE`**:



The Properties window displays editable settings for the selected components.

Note: all connections must be created using the property window, not by writing source code!

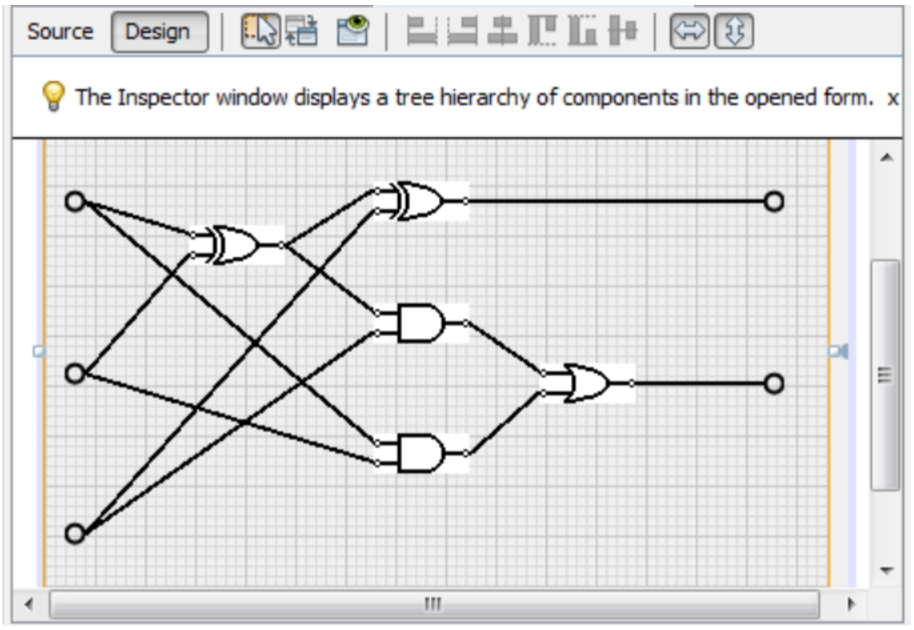
Once that is working correctly, add the remaining gates:



Use the Connection Mode button (in the toolbar) to establish a connection betw

Part F – Bit Adder Circuit (2 marks)

Next we're going to create a circuit to perform ***addition*** of one bit numbers using the gates that we have created so far. Create a new visually designed **`JPanel`** class called **`BitAdderCircuit`**. Drag and drop circuit components onto the designer surface and connect them as follows:

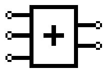


The Inspector window displays a tree hierarchy of components in the opened form.

This circuit is designed to add together two one bit numbers (plus a carry bit). It produces a one bit number as output, plus a carry bit (if the addition overflows). This is basically how computers perform arithmetic – using on/off voltages and simple binary gates.

Part G – Bit Adder Component (2 marks)

To create an *n*-bit adder we would need to chain together *n* of these circuits. Displaying the entire circuit replicated many times is not ideal, so we want to encapsulate the entire bit adder circuit as a single digital component. So, create a new Java class called **`BitAdderComponent`** that is represented like a gate as a single image:



The **`BitAdderComponent`** contains a **`BitAdderCircuit`** (but it is not rendered as such):

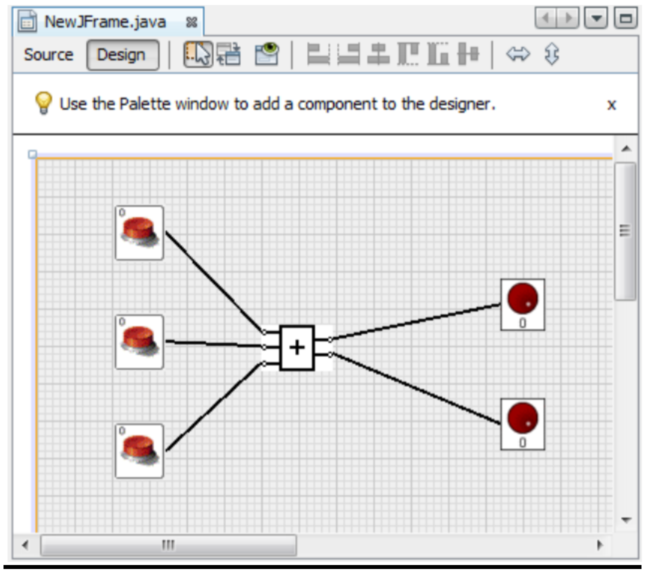
```
private BitAdderCircuit adder = new BitAdderCircuit();
```

In order to allow the BitAdderComponent to access the private Pin fields contained within the BitAdderCircuit, we must first add public getter methods to the BitAdderCircuit. This is most easily done by right clicking within the source code of the BitAdderCircuit and then selecting Insert Code ..., Getter ... and tick each of the Pins to automatically create getter methods named getPin1, getPin2, etc.

The **`BitAdderComponent`** will have three properties named **`Input0`**, **`Input1`** and **`Input2`** (all with both *getters* and *setters*) and two properties named **`Output0`** and **`Output1`** (with just *getters*). The BitAdderComponent does not however, have private fields to store any of those property values. They are implemented instead using the underlying **`adder`** circuit. For example:

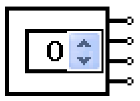
```
    public void setInput0(Terminal input)
    {
        adder.getPin1().setInput(input);
    }

    public Terminal getInput0()
    {
        return adder.getPin1().getInput();
    }

    public Terminal getOutput0() {
        return adder.getPin4().getOutput();
    }
```

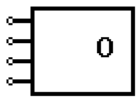
Once implemented, create a new Test circuit in a project called **`PartG`** that consists of a **`BitAdderComponent`** connected to input switches and LED outputs as follows:



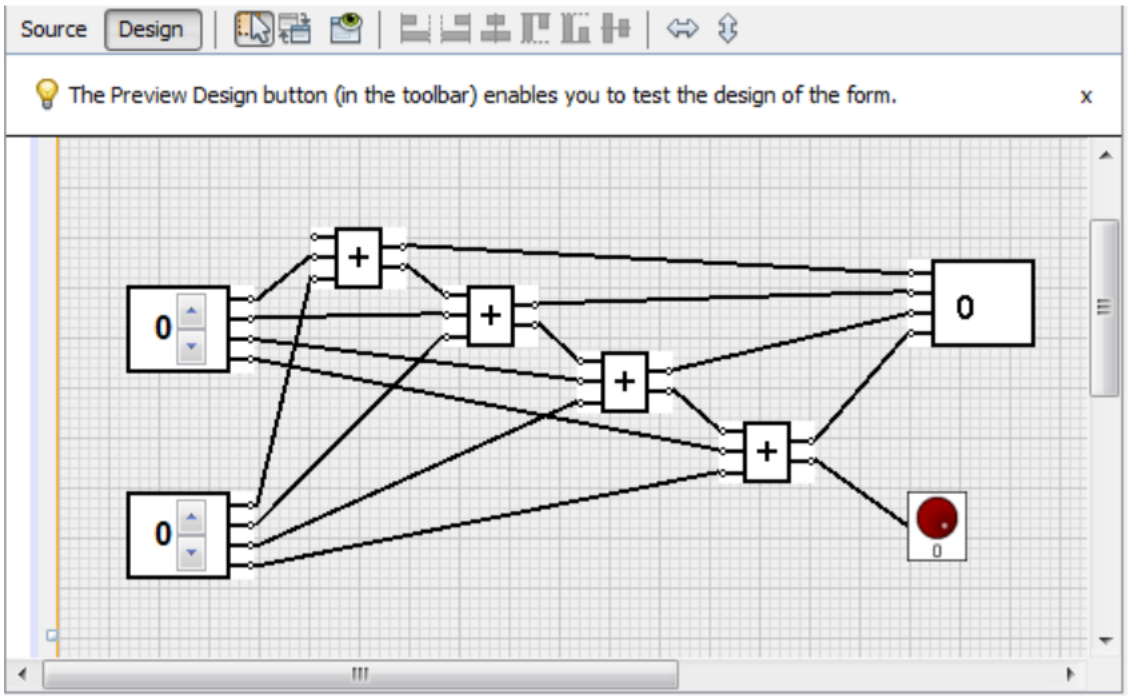
Part H – 4-Bit Adder Circuit (2 marks)

Finally, we will create a project called **`PartH`** and chain together a sequence of bit adders to create a 4-bit adder. We'll also make use of two higher level input and output components that allow 4-bit numbers to be entered and displayed in decimal form. These two components are included in **`ProvidedCircuitComponents.jar`**





Use them and the **`BitAdderComponent`** to create the following 4-bit adder test circuit:



The Preview Design button (in the toolbar) enables you to test the design of the form.

Submission Instructions

Assignments must be submitted via ***Blackboard - Assignment 1*** (listed under Assessment).

You must submit only one file named Assignment1.zip containing exactly 1 ReadMe.txt file up to 4 NetBeans project folders (each including sub folders for build, src, dist, etc) and a SharedLibrary folder. The four project folders should contain:

- DigitalCircuits (Java Class library project)
- Part E test circuit (Java application project)
- Part G test circuit (Java application project)
- Part H test circuit (Java application project)

The DigitalCircuits class library should contain the implementation for all gates, digital components, images and any other classes required to implement them. The test circuits must be in separate projects and should not contain any manually generated source code.

The ReadMe file included in the root directory should list all students names and student numbers and provide be a very short statement of completeness in which you list which parts of the above you have attempted and or completed successfully and if there are any known bugs or deficiencies in what you have submitted. Please be honest and concise. This statement of completeness document should be no more than one paragraph in length.

Extensions will only be granted for extenuating circumstances beyond your control and must be applied for as early as possible and prior to the deadline.

You are strongly encouraged to seek help from the unit coordinator or your tutor if you get stuck with any of the above.