Raspberry Pi Activity: LED the Way

In this activity, you will implement various circuits, primarily using LEDs, resistors, and push-button switches. The Raspberry Pi will initially be used solely as a power supply; however, as the activity progresses, you will use it to programmatically affect the circuits you create. You will need the following items:

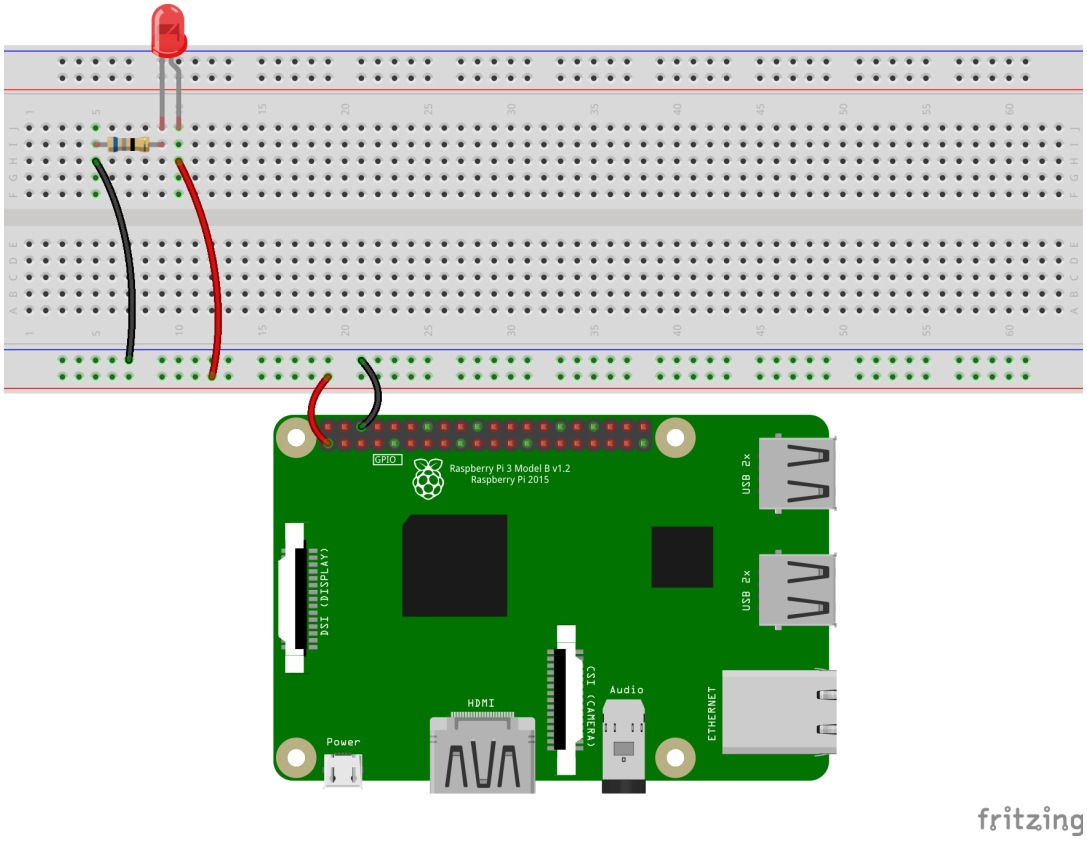
- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- Breadboard;
- GPIO-to-breadboard interface board with ribbon cable; and
- LEDs, resistors, switches, and jumper wires provided in your kit.

GPIO

This is the first activity in which the GPIO (General Purpose Input and Output) pins will be used. Recall that these pins allow various inputs to and outputs from the RPi to be utilized in external circuits (typically implemented on a breadboard).

A simple circuit

The first circuit you will implement is the very simple one shown in an earlier lesson (the topic of which was computer architecture). Here was the layout diagram of that circuit:



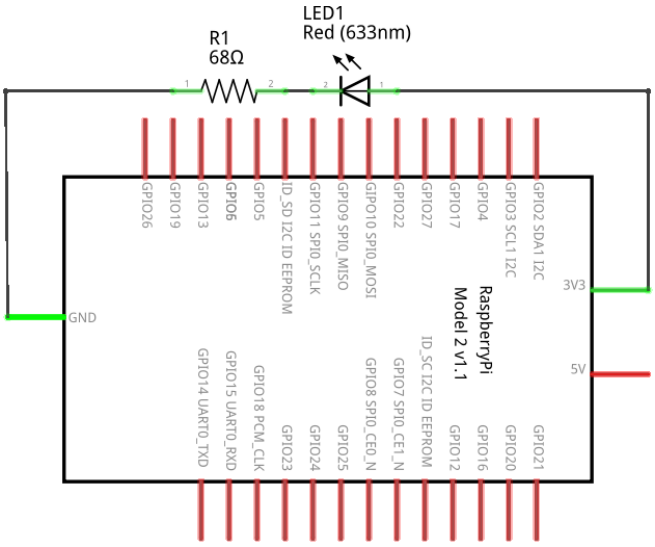
In the diagram, the red wire is connected on one end to a GPIO pin that exposes a 3.3V (3V3) power source. The other end is connected to the bottom row of the breadboard. Recall that this row is internally connected horizontally. Therefore, all holes in that row now have 3.3V.

The black wire is connected on one end to a GPIO pin that exposes ground (GND) or 0V. The other end is connected to the top row in the bottom section of the breadboard. Ground is therefore provided to all holes in that row.

The red LED is inserted so that its legs span across several columns in the center of the breadboard. Recall that these columns are internally connected vertically (however, there is a disconnect across the center gap). Note that the columns are numbered. So for example, the holes in column 20 are internally connected on either side of the center gap (but not to each other across the gap).

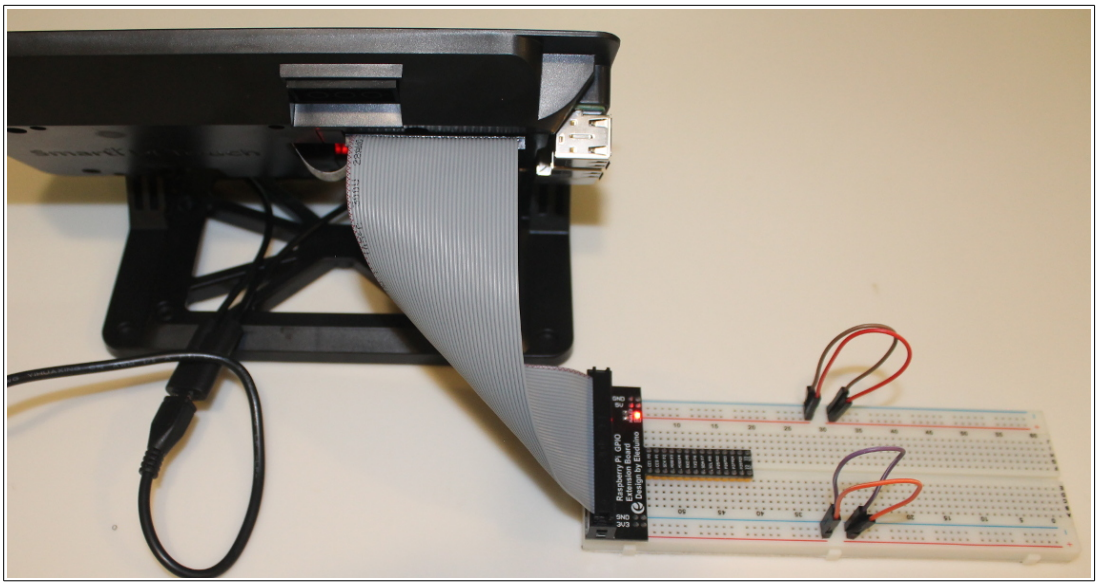
A red wire connected the 3.3V power source from the bottom row to the column that has the positive side of the LED (the long leg – or anode). A 68Ω resistor then connects the negative side of the LED (the short leg – or cathode) to ground. It does so by being placed across several columns (from the negative side of the LED to another column), and then by use of a black wire that brings ground to the other side of the LED.

In all, the circuit requires a red LED, a 68Ω resistor, and some jumper wires (oh, plus the RPi). Here is the circuit diagram:

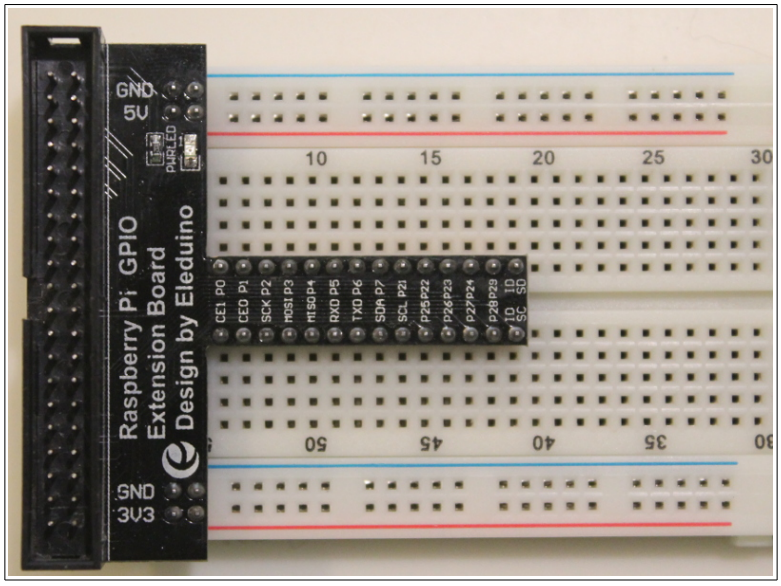


The 68Ω resistor has the colored bands: blue, gray, black. **Note that your kit does not come with 68Ω resistors!** The closest one in your kit is a 220Ω resistor (which will work just fine). It has the colored bands: red, red, brown. For the circuits in this activity, you will use a 220Ω resistor.

Since the power source for the circuit will come from the RPi, we need a way to connect the GPIO pins to the breadboard. One way, as in the layout diagram above, is to connect wires to the GPIO pins and the breadboard. The problem is that the wires in your kit aren't particularly well suited for this. Your kit does, however, include a GPIO interface board that can extend the GPIO pins to the breadboard using a ribbon cable:



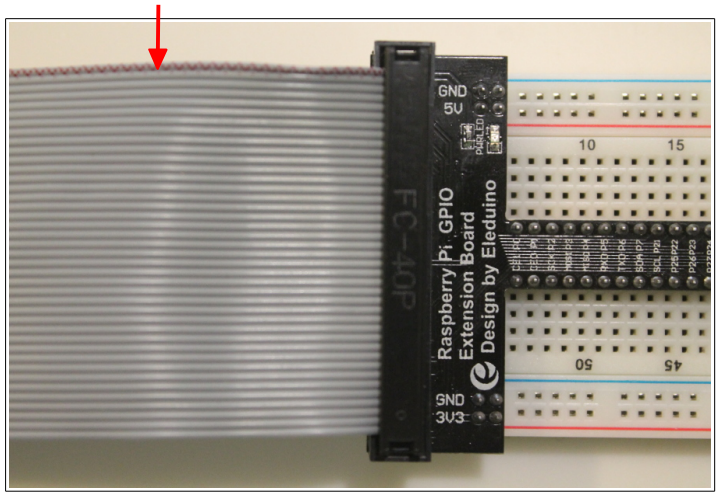
The GPIO interface board extends the GPIO pins to the central holes on the breadboard. First, place the GPIO interface on the breadboard as shown below:



Note that your kit may come with a different GPIO interface board. Perhaps it's a different color (e.g., green vs. black) or has a different pin layout. This activity includes directions for the various GPIO interface boards that may be part of your kit.

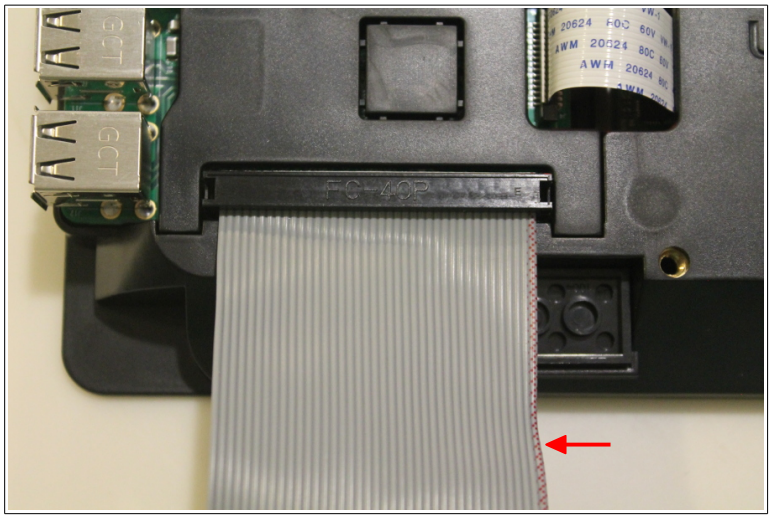
Note the orientation of the breadboard: the positive rails are on the bottom of each section at the top and bottom of the breadboard. Also, the entire GPIO interface rests on the breadboard (i.e., the left part of the GPIO interface in the image above does not hang over the edge of the breadboard), and the central pins straddle the gap in the center of the breadboard. When pressing the interface board into the breadboard, make sure to put even downward pressure entirely across it to prevent it from breaking.

Next, connect the ribbon cable to the GPIO interface as show below:

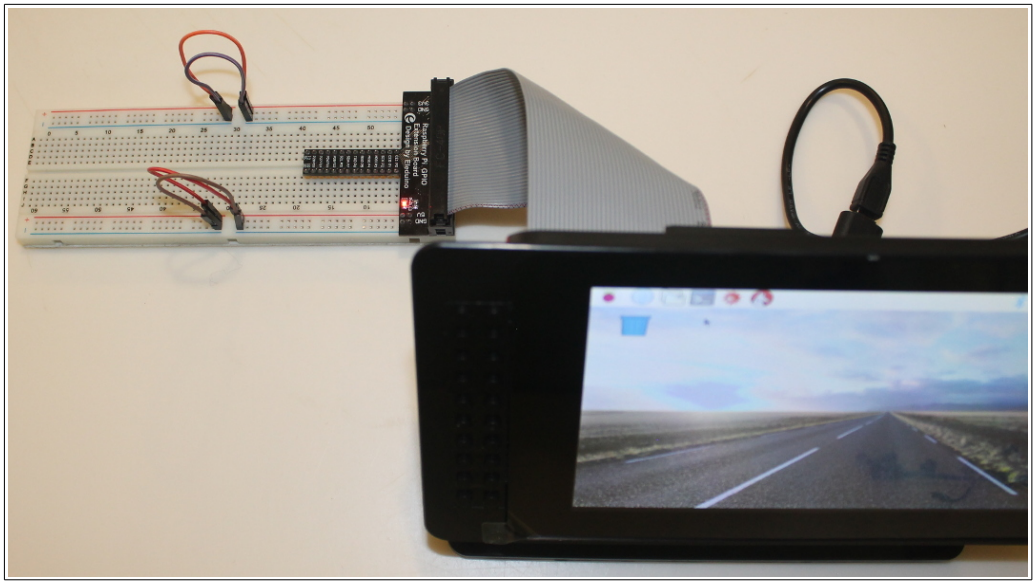


Notice how the **red edge** of the ribbon cable (as noted by the arrows) is aligned with the top of the breadboard. There's also a tab on the hard plastic end of the ribbon cable that prevents it from being inserted incorrectly into the GPIO interface.

Next, connect the other end of the ribbon cable to the GPIO pins on the RPi that are exposed at the rear of the stand:



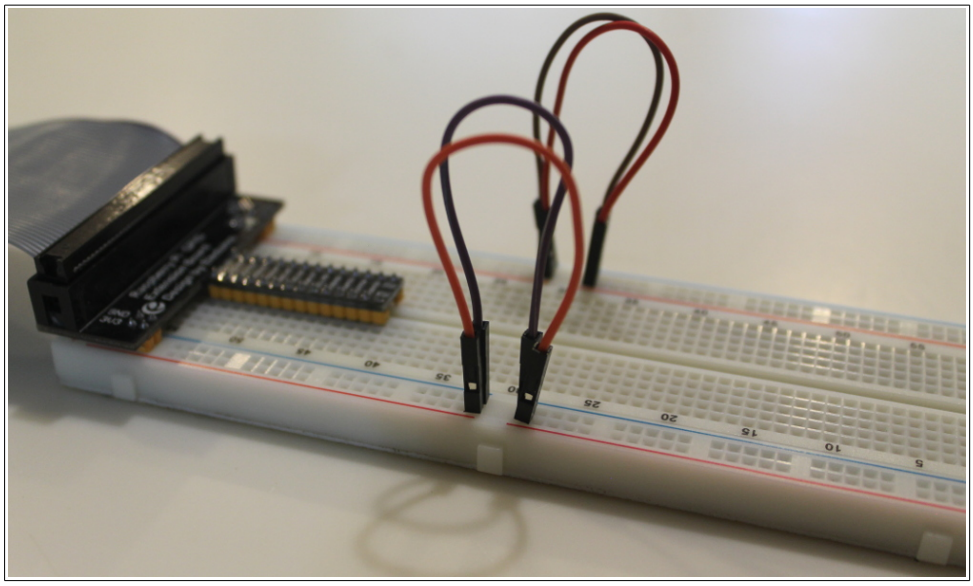
Again, notice the orientation of the ribbon cable! The best way to lay everything out is shown below:



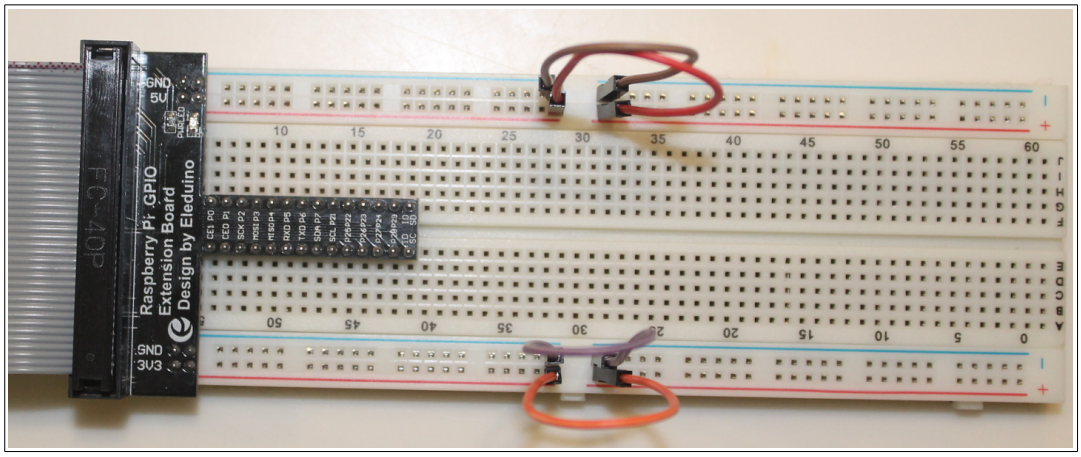
If you've connected everything correctly, a little red LED on the GPIO interface should be on.

The GPIO interface allows circuits to be connected to the RPi's GPIO pins. It also exposes +5V, +3.3V, and GND. Viewed as before (where the GPIO interface is to the left of the breadboard), 5V and GND are on the top rails, and 3.3V and GND are on the bottom rails.

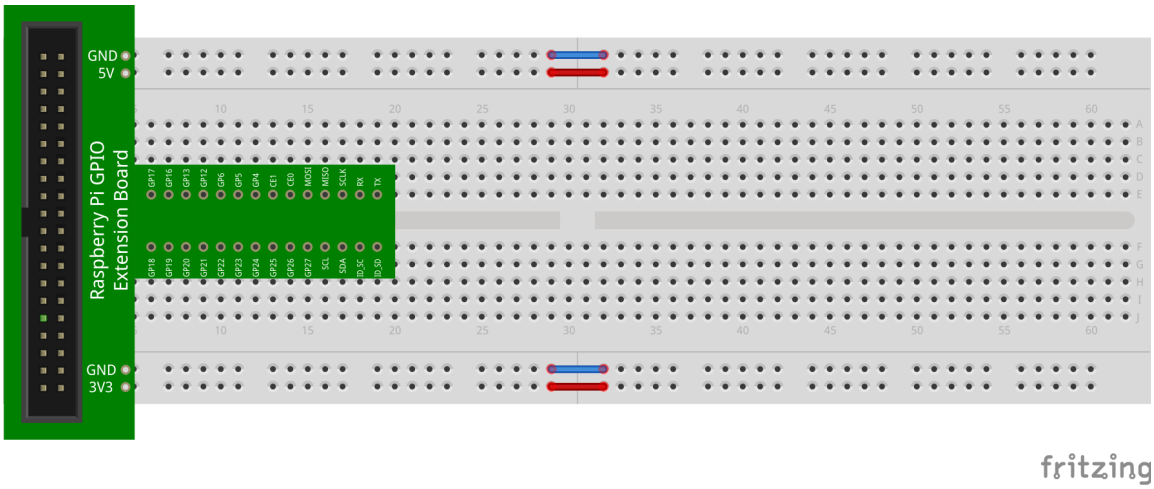
A word of caution, however! Your breadboard may not be internally connected across the entire top and bottom rails. If not, you will need to bridge the left and right halves of the rails as shown below to ensure that power and GND are exposed across the entire length of the rails:



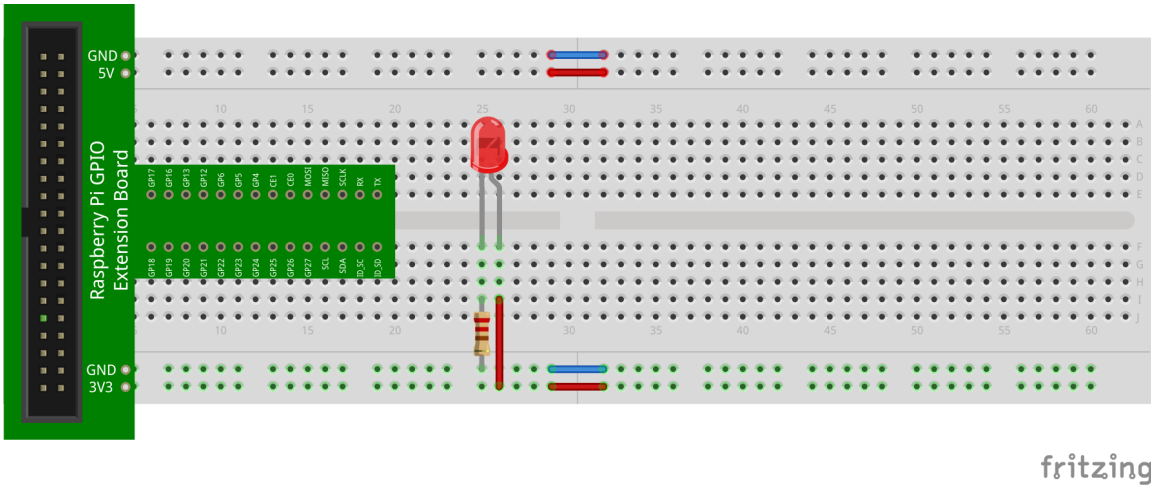
This must be done for both 5V and GND in the top rails, and 3.3V and GND in the bottom rails:



Here's a layout digram of this:



We can now create a layout diagram of the circuit shown earlier a bit to make use of the interface board:



Note that it is assumed that a ribbon cable connects the interface board to the GPIO pins on the RPi (i.e., it is not shown in the layout diagrams).

Create the circuit shown in the layout diagram above **without connecting the power adapter to the RPi yet**. Make sure that:

- The LED straddles two columns (i.e., does not have both of its legs in the same column of holes);
- You connect a wire from a 3.3V pin on the bottom rail to the positive side (long leg) of the LED; and
- You place a 220Ω resistor from the negative side (short leg) of the LED to ground (note that the GPIO interface board brings ground to **both top rows** of the top and bottom rails).

To summarize, use a red (or similar) wire to connect a 3.3V power source from the interface board to the positive side of the LED. Use a 220Ω resistor to connected the negative side of the LED to GND. When you are certain that your circuit is correct, plug in the RPi. If everything is wired correctly, the LED should light.

Be careful! If you accidentally short 3.3V (or 5V for that matter) directly to ground (i.e., connect 3.3V or 5V directly to ground), your RPi will reset (and you may damage it in the process)!

And perhaps even more importantly, directly connecting 5V to 3.3V will "brick" your RPi. "Brick" is a term meaning to cause an electronic device to become completely unable to function (as in permanently).

Calculations

The resistor used in the circuit is a 220Ω resistor, the source voltage is 3.3V, and the LED has a forward voltage (i.e., voltage drop) of 2V (which we know from reading its data sheet). Using Ohm's Law, we can calculate the current that will flow through the circuit as follows:

$$\begin{aligned} V &= I * R \\ (3.3V - 2V) &= I * 220\,\Omega \\ 1.3V &= I * 220\,\Omega \\ 0.00591\,A &= I = 5.91\,mA \end{aligned}$$

The current through the LED will be 5.91mA. This is much less than the recommended 20mA. Note that the brightness of the LED is directly related to how much current flows through it. The more current, the brighter it will be. Of course, there is a limit (as per the data sheet).

We can calculate the power dissipated by the resistor as follows:

$$\begin{aligned} P &= V * I \\ P &= (3.3V - 2V) * 0.00591\,A \\ P &= 1.3V * 0.00591\,A \\ P &= 0.0077W = 7.7\,mW \end{aligned}$$

The resistor in your kit is a 1/4W (250mW) resistor. It is more than enough. The power dissipated by the LED can be calculated similarly:

$$\begin{aligned} P &= V & * \; I \\ P &= 2V & * \; 0.00591\,A \\ P &= 0.0118\,W & = \; 11.8\,mW \end{aligned}$$

The LED in your kit has a forward current limit of 120mW. Again, it is more than enough.

Increasing the voltage

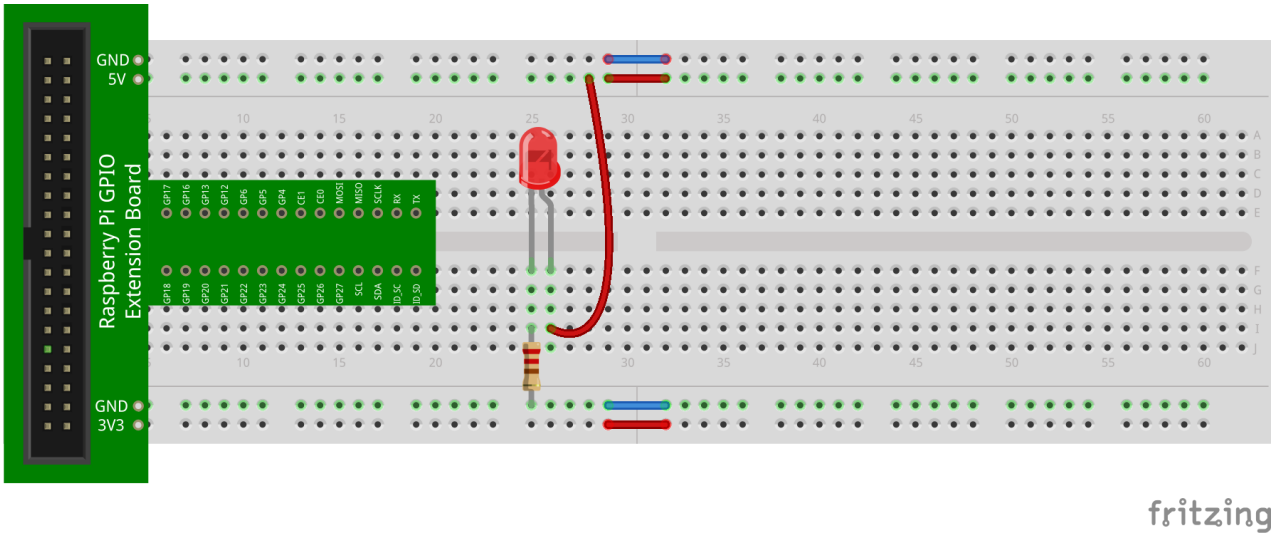
To make the LED a bit brighter, we cannot reduce the resistance in the circuit since there aren't any lesser-valued resistors in the kit. We can, however, change the source voltage! The RPi also has a 5V power source. Suppose you were to, instead, use the 5V power source from the RPi. This would provide more voltage to the breadboard and across the circuit. According to Ohm's Law, if we increase the voltage and keep the resistance in the circuit at 220Ω, the current has to increase. In the space below, calculate the current that would flow through the circuit with a 5V power source and a 220Ω resistor:

Now, calculate the power dissipated by the resistor for the 5V power source:

And finally, calculate the power dissipated by the LED for the 5V power source:

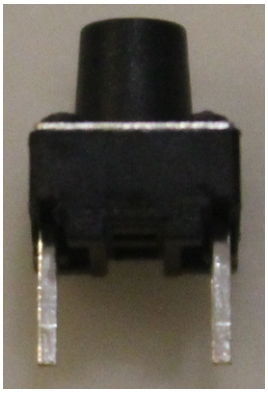
So with the 220Ω resistor and a 5V power source, we increase the current – which should make the LED appear a bit brighter when lit.

Alter your circuit as in the figure below. The only difference is that the positive side of the LED should now be connected to 5V instead of 3.3V:



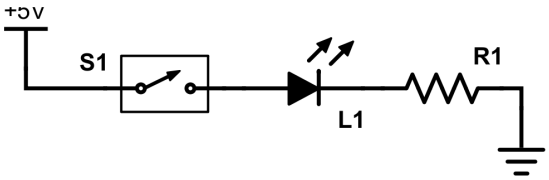
Adding a push-button switch

Let's add a push-button switch to the above circuit. The switch will control the flow of electricity to the circuit. If the button is pushed, current will flow and the LED will light. A push-button switch is a *tactile* switch that usually has two to four legs. The switch in your kit has two legs:

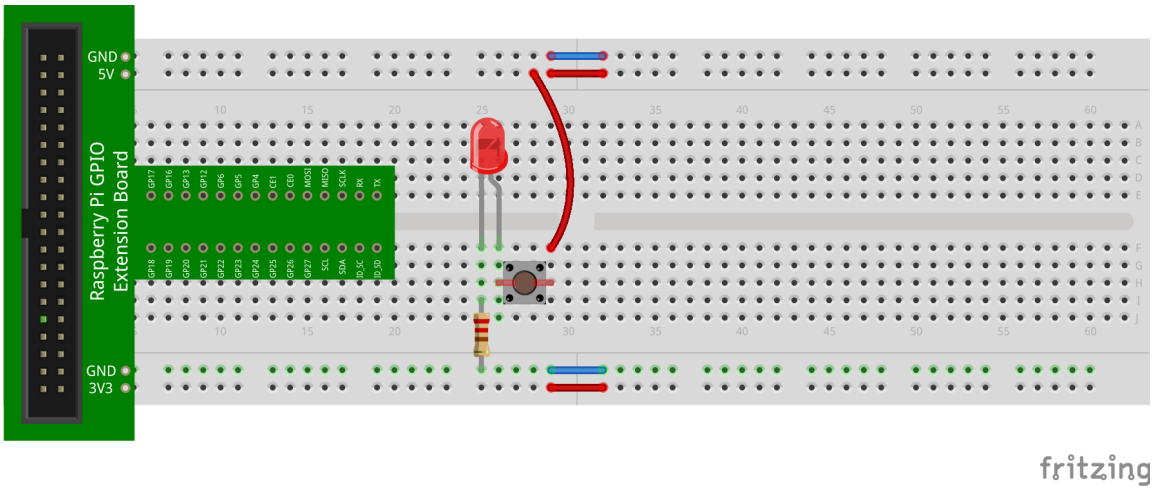


This type of switch is typically positioned across several columns in the central part of the breadboard. That is, the pins should be in separate columns. Power is connected to one pin. The part of the circuit to be powered when the switch is pushed is connected to the other pin.

Modify your circuit by adding a switch as indicated below:



Here's a layout of this circuit:



Note that one pin of the switch is connected to +5V, and the other to the positive side (long lead) of the LED.

An interesting experiment is to see the difference in LED brightness when connected to 5V vs. 3.3V. We can quickly calculate the amount of current flowing through the LED in both cases, with a constant resistance of 220 Ohms (also assuming that the voltage drop across the LED is 2V). First, with 5V:

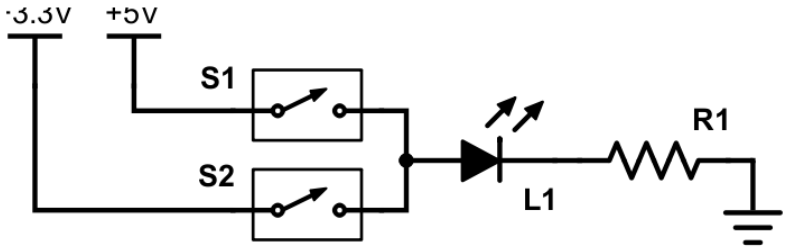
$$V = IR$$
$$5V - 2V = I(220\ \text{Ohms})$$
$$3V = I(220\ \text{Ohms})$$
$$I = 3V / 220\ \text{Ohms}$$
$$I = 0.014\ A = 14\ mA$$

Now, with 3.3V:

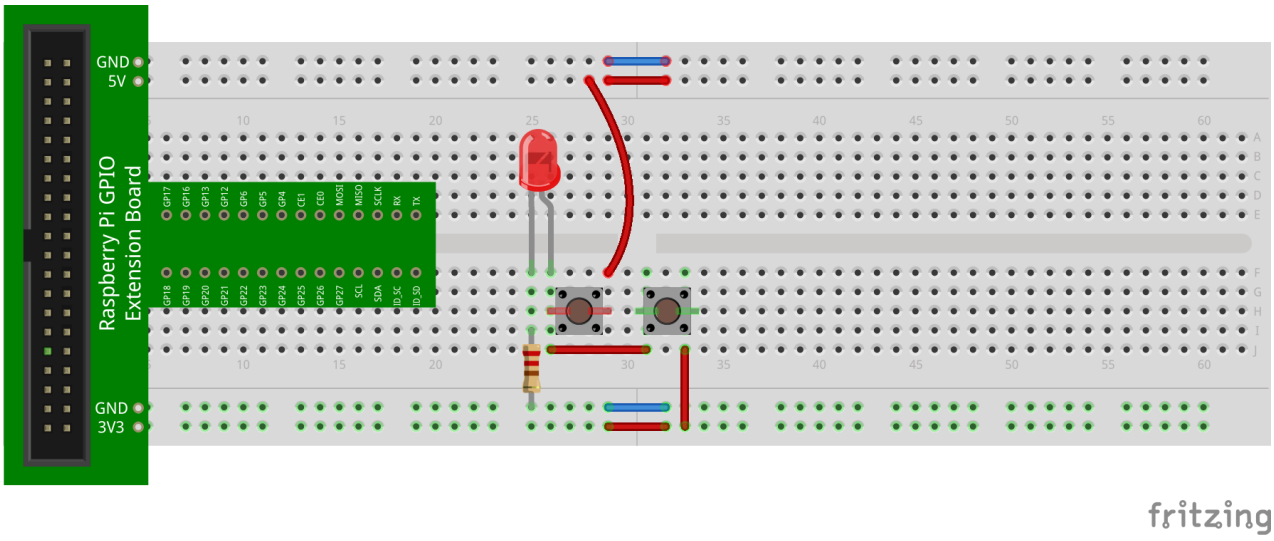
$$
\begin{aligned}
V &= IR \\
3.3V - 2V &= I(220\ \text{Ohms}) \\
1.3V &= I(220\ \text{Ohms}) \\
I &= 1.3V / 220\ \text{Ohms} \\
I &= 0.006\ A = 6\ mA
\end{aligned}
$$

The LED's brightness should be noticeably different!

Let's try out another experiment to further illustrate this. Implement the following circuit:



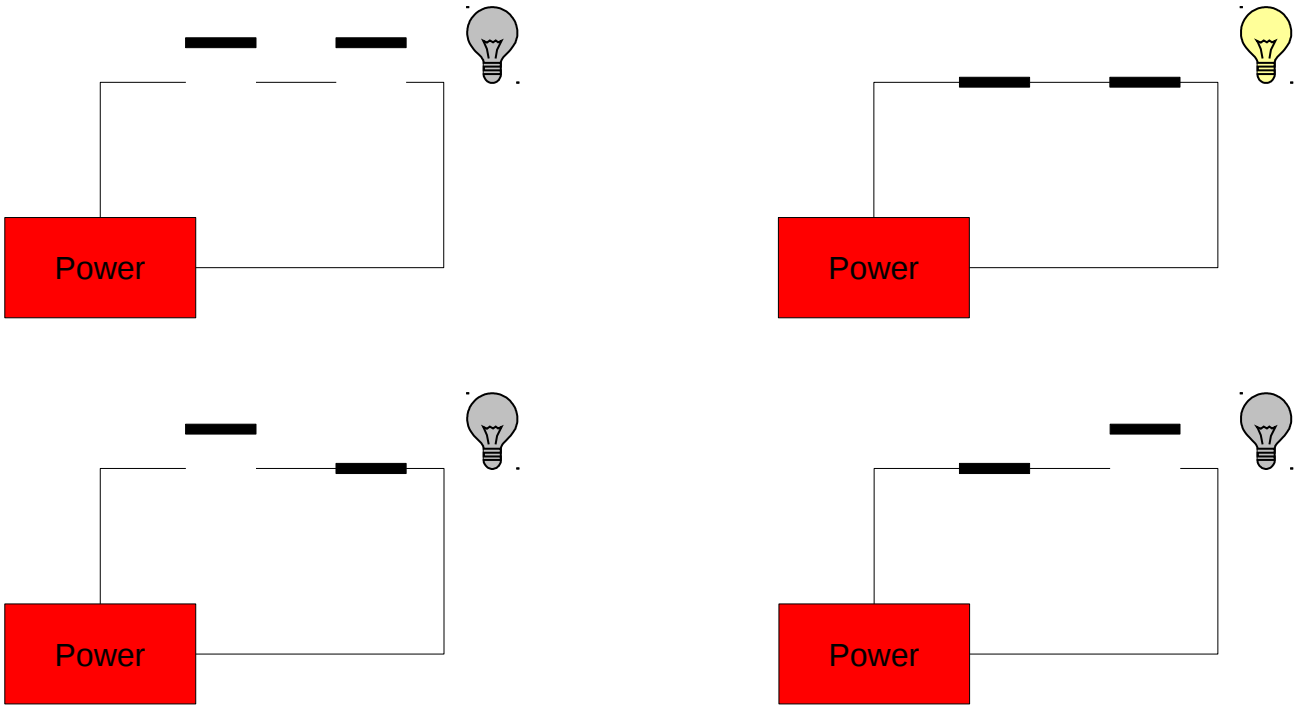
Here's one way to layout this circuit:



To test, experiment by pressing the switches in an alternating fashion.

Recall that the circuits in a previous lesson also included versions with multiple switches (both in parallel and in series). We later related this to logic gates. The rest of this activity will have you experiment with various configurations of multi-switch circuits.

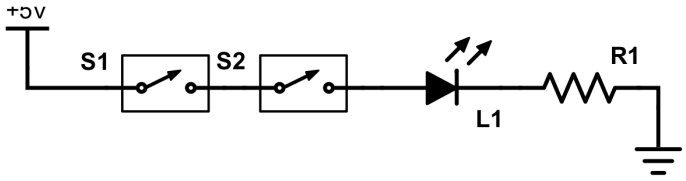
Replicating the *and* gate
Recall the following circuit:



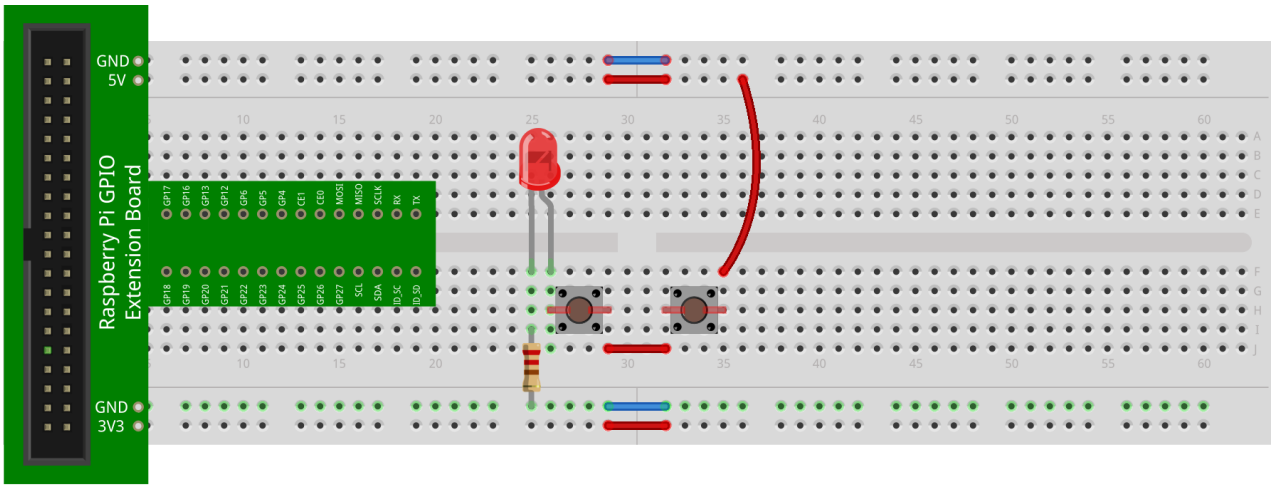
This circuit has two switches in *series*. Placing switches in this configuration in the circuit replicates the functionality of the *and* gate. Fill in the truth table for the *and* gate below:

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	**1**

The output, *Z*, is only 1 (true) when both inputs, *A* and *B*, are 1 (true). In the circuit above, the light bulb is on (1) when both switches are closed (1). To implement this in your LED circuit, modify it as follows:



Here's one way to layout the circuit:

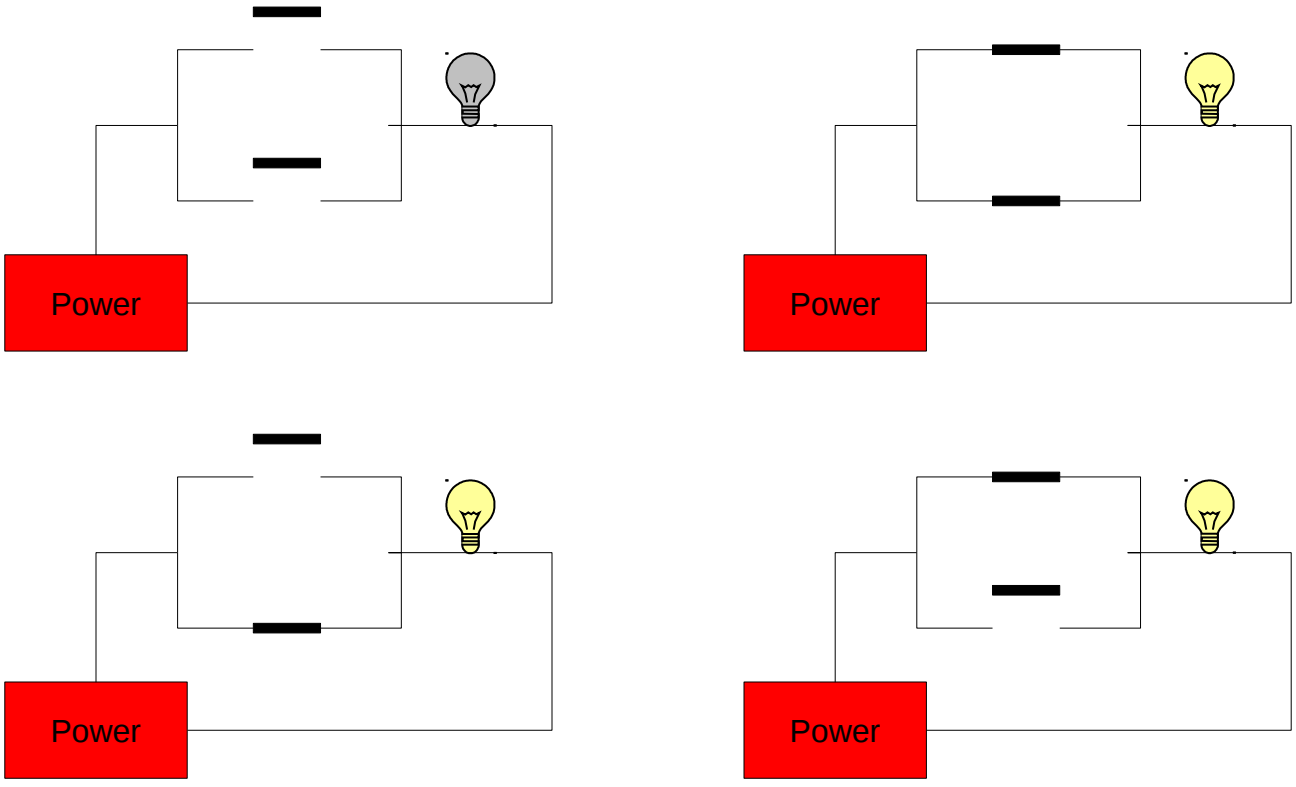


Power is extended to one of the switches (via the long red wire). This switch is then connected to the second switch. That is, this switch's left pin (in the figure above) is connected to the second switch's right pin. The second switch's left pin is connected to the positive side of the LED (they are in the same column). The rest of the circuit (resistor from the negative side of the LED to ground) is the same.

Try the circuit. The LED should only light when *both* switches are closed.

Replicating the *or* gate

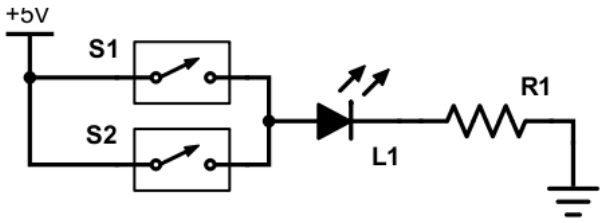
We can also implement the functionality of the *or* gate as follows:



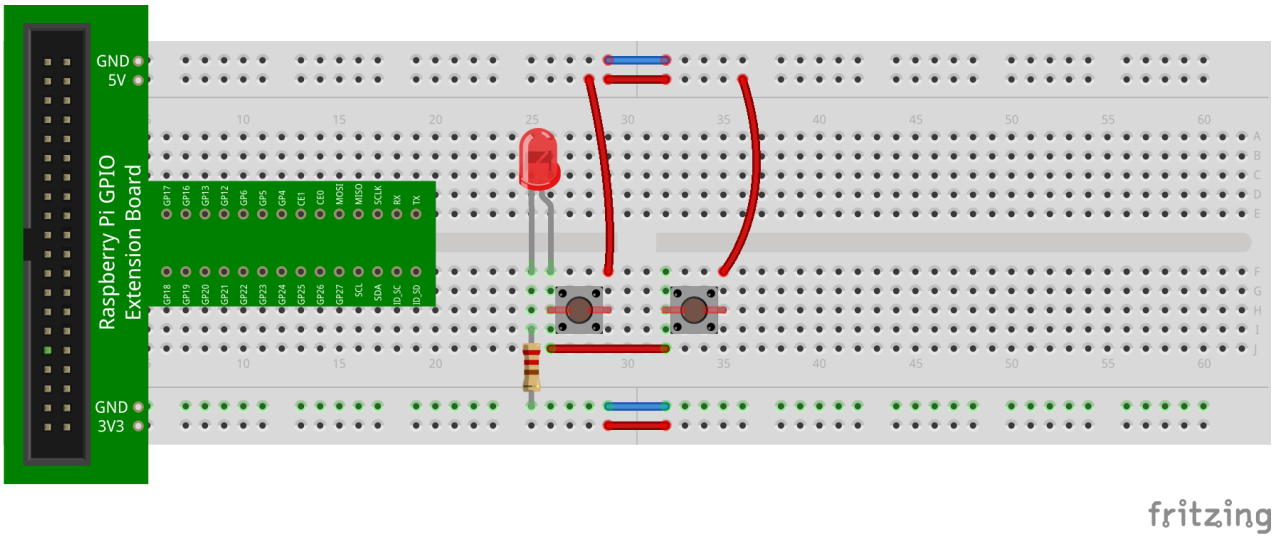
This circuit has two switches in *parallel*. Placing switches in this configuration in the circuit replicates the functionality of the *or* gate. Fill in the truth table for the *or* gate below:

A	B	Z
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

The output, *Z*, is 1 (true) when any input (*A*, *B*, or both *A* and *B*) are 1 (true). In the circuit above, the light bulb is on (1) when either switch (or both) are closed (1). To implement this in your LED circuit, modify it as follows:



Here's one way to layout the circuit:



Power is extended to one pin **of both switches** (via the two long red wires). The second pin **of both switches** is connected to the positive side of the LED. Try the circuit. The LED should light when either (or both) switches are closed.

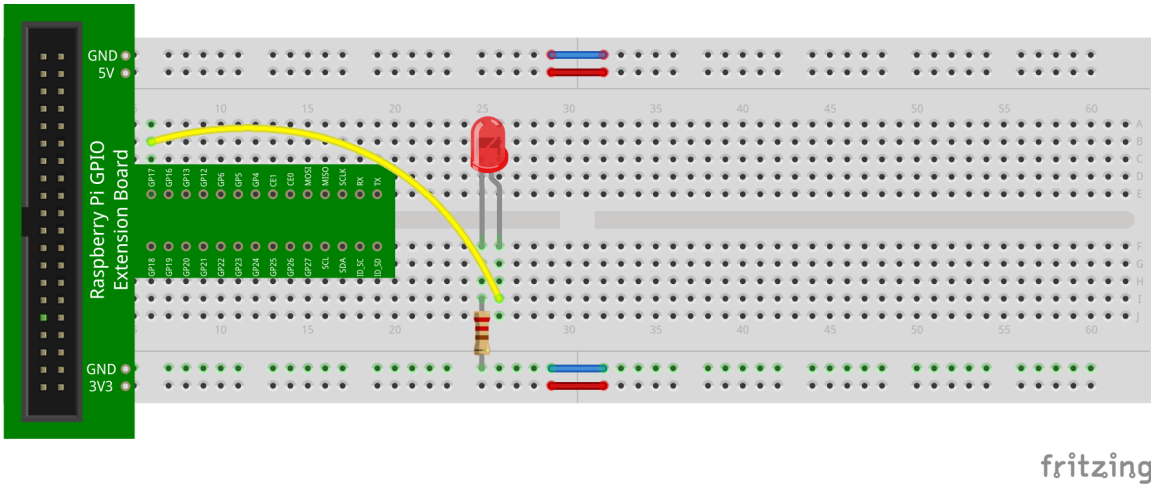
Sensing

The previous circuits in this activity have been entirely external of the RPi. That is, the RPi was only used as a power source (basically, a battery). There are many more GPIO pins than the ones we have

used so far (3.3V, 5V, and ground). In fact, many of the GPIO pins can be used to provide sensor input to the RPi. Others can be used to provide output capabilities from the RPi. For example, we can programmatically detect when a switch is closed and trigger an LED to light.

But before we can get to this, we must first discuss how to access and manipulate the GPIO pins on the RPi in Python. Fortunately, Python has a library called RPi.GPIO that can be imported. This library is installed by default on the RPi. Let's start with a simple example: lighting an LED. Construct the following circuit (which slightly differs from the first part of this activity):

Here's one way to layout this circuit:



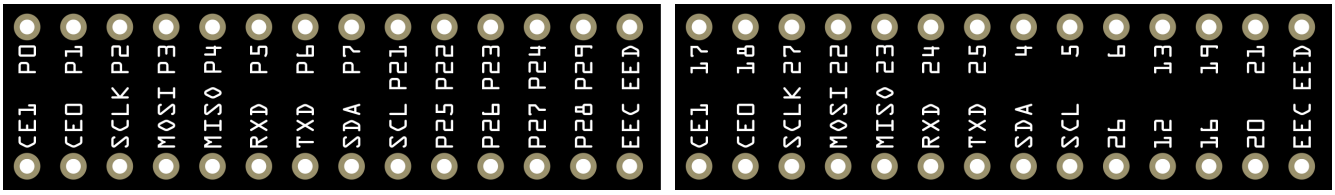
Note that the yellow wire connects the positive side of the LED to a GPIO pin labeled **GP17** on the GPIO interface. If you have the **black** GPIO interface board, the yellow wire will connect to the same physical pin location, but the pin on the GPIO interface board will be labeled **P0** instead of **GP17**. This is a good time to discuss pin numbering schemes. It turns out that there are actually **three** different pin numbering schemes in use with GPIO pins on the RPi: (1) the **physical** pin order on the RPi; (2) the numbering assigned by the manufacturer of the **Broadcom** chip on the RPi; and (3) an older numbering assigned by an early RPi user who developed a library called **wiringPi**. Pins also have a name (e.g., 5V, GND, GPIO.0, etc). Here's a table that cross-references each (and includes names of the pins):

BCM	wPi	Name	Physical		Name	wPi	BCM
		3V3	1	2	5V		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	GND		
4	7	GPIO.7	7	8	TXD	15	14
		GND	9	10	RXD	16	15
17	0	GPIO.0	11	12	GPIO.1	1	18
27	2	GPIO.2	13	14	GND		
22	3	GPIO.3	15	16	GPIO.4	4	23
		3V3	17	18	GPIO.5	5	24
10	12	MOSI	19	20	GND		
9	13	MISO	21	22	GPIO.6	6	25
11	14	SCLK	23	24	CE0	10	8
		GND	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	GND		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	GND		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		GND	39	40	GPIO.29	29	21

Don't worry about understanding the names of the pins for now (although you may have noticed that they somewhat correlate with the wiringPi numbering scheme). The GPIO pin in the layout diagram above that has the yellow wire connecting to the LED is labeled GP17. The labeling on the green GPIO interface in your kit uses Broadcom (BCM in the table above). GP17 is just BCM pin 17 which cross-references to wiringPi (wPi) pin 0 and physical pin 11 on the RPi. You may need this reference chart anytime you write Python programs that make use of the GPIO pins.

Note that Python primarily uses the Broadcom (BCM) pin numbering scheme which, thankfully, matches the GPIO interface board! For those of you that have the **black** GPIO interface board, using this chart can easily provide a crossreference from a BCM pin to a wPi one. In this case, BCM pin 17 (**GP17**) matches wPi pin 0 (**P0**). In a Python program, we simply need to refer to BCM pin 17 to match wPi pin 0.

For reference, here's a comparison of the **black** GPIO interface boards labeled with both pin numbering schemes (wPi on the left, and BCM on the right):



Note that the green GPIO interface board is labeled directly using the BCM pin numbering scheme; therefore, no crossreference is needed.

Importing the RPi.GPIO library is a simple as including the following **import** statement (typically done at the beginning of a Python program):

```
import RPi.GPIO as GPIO
```

To refer to GPIO pins using the Broadcom pin layout, set the mode as follows:

```
GPIO.setmode(GPIO.BCM)
```

To turn the LED on, we must first configure pin 17 (again, using the BCM pin layout) to be an **output** pin as follows:

```
GPIO.setup(17, GPIO.OUT)
```

And finally, to turn on the LED:

```
GPIO.output(17, GPIO.HIGH)
```

This turns the pin on by supplying it 3.3V. And that's all there is to turning on an LED! Here's the full program for reference:

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)          # set the pin mode
GPIO.setup(17, GPIO.OUT)        # setup pin 17 as an output pin
GPIO.output(17, GPIO.HIGH)      # 3.3V to the pin (turn on the LED)
```

Turning the LED off can be done as follows (which turns the pin off by supplying it with 0V):

```
GPIO.output(17, GPIO.LOW)       # 0V to the pin (turn off the LED)
```

Did you know?

Instead of using GPIO.HIGH to supply 3.3V to a pin, you can use 1 or True. For example, the following statements are identical (i.e., they produce the same result):

```
GPIO.output(17, GPIO.HIGH)
GPIO.output(17, 1)
GPIO.output(17, True)
```

Likewise, the following statements are identical and supply 0V to a pin:

```
GPIO.output(17, GPIO.LOW)
GPIO.output(17, 0)
GPIO.output(17, False)
```

Now, let's try to blink the LED. This will mean turning the output pin on, waiting some amount of time, turning the output pin off, waiting some amount of time, and so on. We already know how to turn an output pin high and low. We also know how to repeat a task over and over (we can use a while loop!). But we'll need to introduce a small delay so that we can actually see the LED blink. To do this, we can import the *time* library and make use of its `sleep` function:

```
from time import sleep
```

This will allow us to introduce delays. For example, we can introduce a half second delay as follows:

```
sleep(0.5)
```

To blink an LED with a half second delay in between each state of the LED (on or off) and blink the LED *forever*, we can modify our program as follows:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT)

while (True):
     GPIO.output(17, GPIO.HIGH)
     sleep(0.5)
     GPIO.output(17, GPIO.LOW)
     sleep(0.5)
```

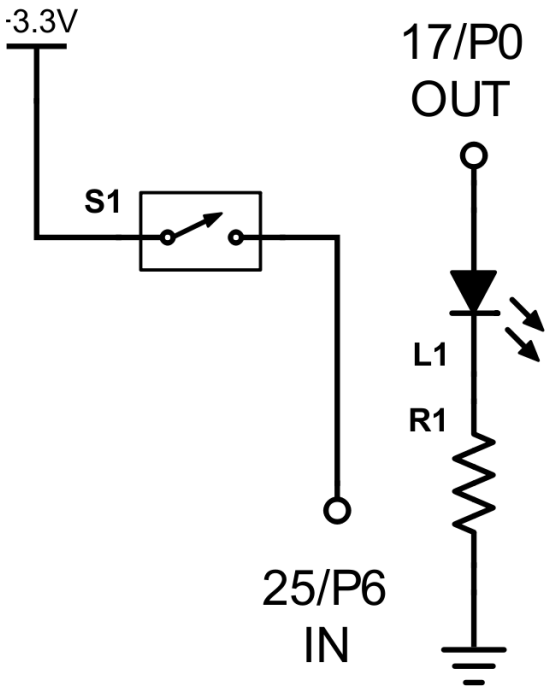
Note that the **while** loop will go on forever (i.e., `while (True)` is never false!). **To stop the program, we can press Ctrl+C**. When doing so, you may notice warnings or errors. This is normal, because our program did not clean up before aborting. In fact, you will most likely also get errors if you try to run the program again (or another program that uses the GPIO pins). These errors are safe to ignore for now. Usually when using GPIO pins, it is recommended to clean them up so that they are reset. We won't worry about this right now.

Often, it is standard practice to assign GPIO pin numbers to meaningful variables. For example, we can assign pin 17 to the variable *led* (since in our program it is used to control an LED). In the end, we can modify our program as follows:

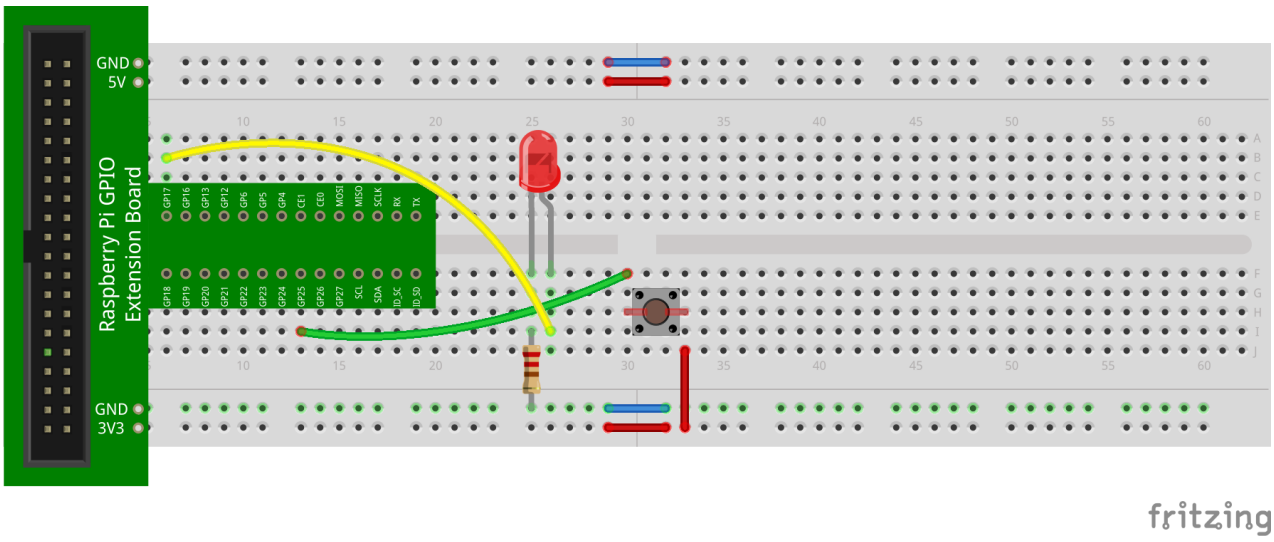
```
import RPi.GPIO as GPIO
from time import sleep

led = 17

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)

while (True):
     GPIO.output(led, GPIO.HIGH)
     sleep(0.5)
     GPIO.output(led, GPIO.LOW)
     sleep(0.5)
```

Adding a switch

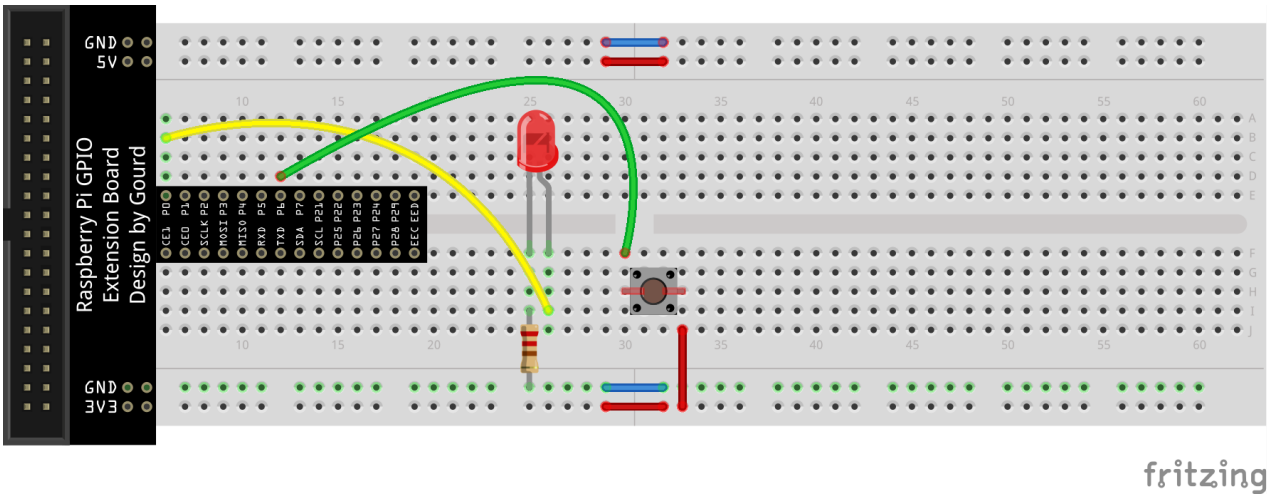
Let's add a switch to the mix. To make this work, we want to programmatically detect the status of the switch (open or closed). If the switch is closed, then the LED should turn on; otherwise, it should remain off. To begin, implement the following circuit:



Here's one way to layout this circuit:



If you have the **black** GPIO interface board, layout the circuit as follows instead:



Again, the yellow wire is connected to GP17 (wPi P0) and the positive side of the LED. The green wire is connected to one side of the switch and to **GP25** (wPi **P6**). The LED portion is unchanged from the last circuit. The only difference is the addition of a switch. One side is connected to +3.3V, and the other is connected to GP25 (wPi P6).

Detecting the state of the switch is not particularly difficult. The pin's state is first setup to be an input pin. While we're at it, we'll set a variable, *button*, to store the number of the pin (like we did with the LED above):

```
button = 25
GPIO.setup(button, GPIO.IN)
```

In Python, input switches can be wired to positive voltage (e.g., +3.3V as in the layout diagram above) or to ground. Let's refer to the case where one pin of the switch is wired to +3.3V and the other to the input pin as **CASE 1**. By default, the input pin should be low. In fact, it should be intentionally **pulled down** to GND to ensure this. If the switch is open, current cannot flow to the input pin. Pressing the switch closes the circuit and allows +3.3V to flow to the input pin, thereby setting it high. The state change can be read to detect the pushing of the button!

We'll define **CASE 2** to be the case where one pin of the switch is wired to GND and the other to the input pin. In this configuration, the input pin is **pulled up** and actually provides current (but it has nowhere to go if the switch is not closed). Pressing the switch closes the circuit and allows current from the input pin to flow to GND. Again, the state change can be read to detect the pushing of the button.

Since both of these ways of detecting an input are possible in Python, it is standard practice to set an input pin's *default* state (which depends on how it is wired). We do so by either connecting the input pin (internally through our program) to 3.3V or to ground. To connect the pin to 3.3V, the RPi internally uses a **pull-up** resistor (which *pulls* the state of the input pin *up* to 3.3V). To connect the pin to ground, the RPi internally uses a **pull-down** resistor (which *pulls* the state of the input pin *down* to 0V). Specifying a default input pin state can be done as follows:

```
button = 25
# for case 1
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
# for case 2
```

```
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Which you choose doesn't matter in most cases. Just make sure that you connect the other side of the switch as appropriate (e.g., to +3.3V if the input pin has a pull-down resistor and is low by default, or to ground if the input pin has a pull-up resistor and is high by default).

Did you know?

You can set multiple GPIO pins either as input or output in one single statement. The method involves providing a list of the GPIO pins to the setup command as follows:

```
out_pins = [17, 18]
in_pins = [22, 27]
GPIO.setup(out_pins, GPIO.OUT)
GPIO.output(in_pins, GPIO.IN)
```

This sets GPIO 17 and 18 as input pins and GPIO 22 and 27 as output pins. In fact, this can also be used to set all of the output pins in the output pin list (GPIO 17 and 18) as either high or low as follows:

```
GPIO.output(out_pins, GPIO.HIGH)
```

Setting GPIO 17 high and GPIO 18 low can be done in one statement as follows:

```
GPIO.output(out_pins, (GPIO.HIGH, GPIO.LOW))
```

Reading the state of an input pin can be done as follows:

```
if (GPIO.input(button) == GPIO.HIGH):
    ...
```

To begin, let's just display a status message that informs us whether the switch is open or closed. Here's the full Python program:

```
import RPi.GPIO as GPIO
from time import sleep

led = 17
button = 25

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

while (True):
    if (GPIO.input(button) == GPIO.HIGH):
        print "Closed!"
    else:
        print "Open!"
    sleep(1)
```

Try running the program. Notice that when the switch is open, “Open!” appears; otherwise, “Closed!” appears. This happens every second. Why?

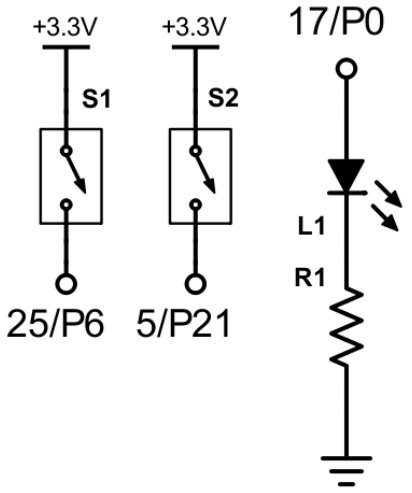
To *connect* the switch to the LED (i.e., to make the switch control the LED), simply change the statements in the while loop as follows (note that the rest of the program remains unchanged). While we're at it, we can sleep a little less each time to allow the circuit to react faster to changes in the switch state:

```
while (True):
    if (GPIO.input(button) == GPIO.HIGH):
        GPIO.output(led, GPIO.HIGH)
    else:
        GPIO.output(led, GPIO.LOW)
    sleep(0.1)
```

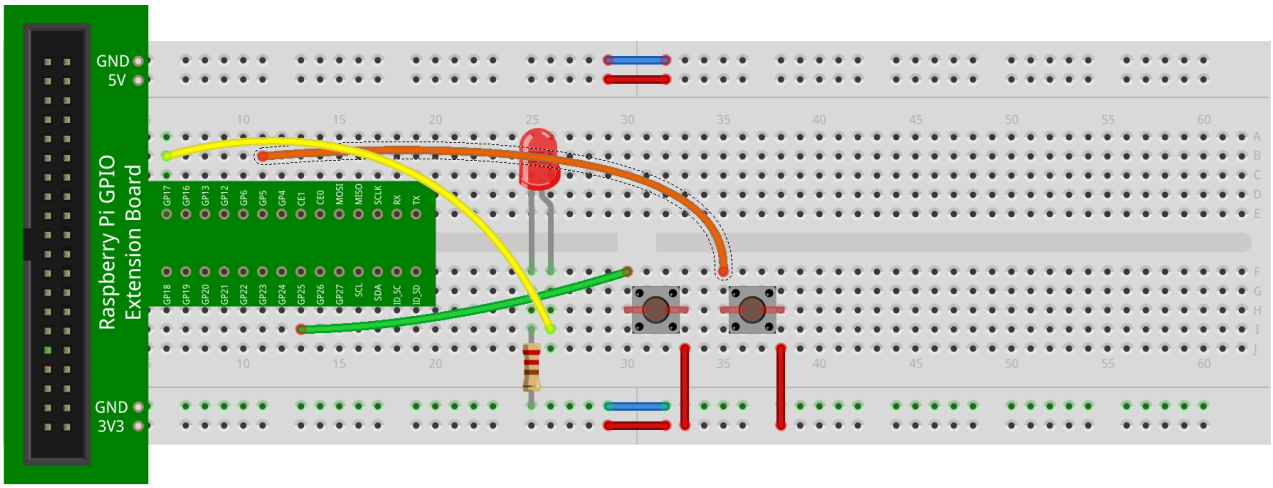
Detecting an input pin in this way is called **polling**. The input pin is repeatedly polled (checked) for its state. As you can see, this repeats forever and can use a lot of CPU processing time. There are better ways to detect changes in input pins that do not keep the CPU so busy; however, this will work for now.

Two switches (to implement *and* and *or*)

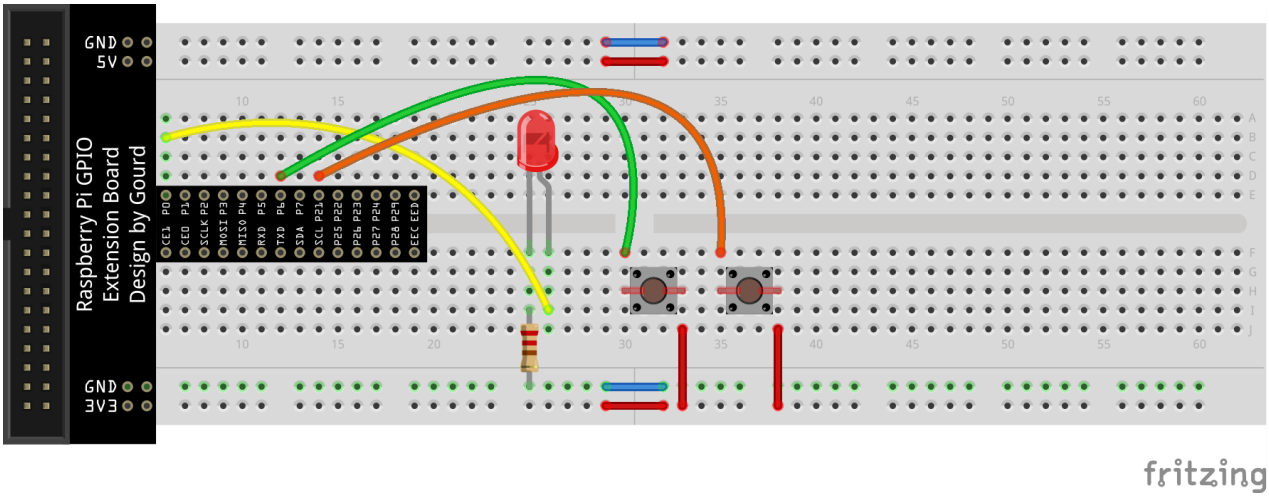
Earlier, to manually implement *and* and *or*, two switches had to be wired either in series or parallel. Programmatically doing so precludes this. The logic can be done in Python! Let's try this by first implementing the following circuit:



Here's one way to layout this circuit:



If you have the **black** GPIO interface board, layout the circuit as follows instead:



The only difference in this circuit is the addition of the second switch. It is connected to +3.3V and to **GP5** (wPi **P21**). To implement the functionality of *and* (i.e., replicating two switches in series), we simply need to turn the LED on when both input pins read high. We can do this by implementing the following Python program:

```
import RPi.GPIO as GPIO
from time import sleep

led = 17
button1 = 25
button2 = 5

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button1, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(button2, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

```
while (True):
      if (GPIO.input(button1) == GPIO.HIGH and GPIO.input(button2) == GPIO.HIGH):
            GPIO.output(led, GPIO.HIGH)
      else:
            GPIO.output(led, GPIO.LOW)
      sleep(0.1)
```

The logic is actually quite clear: the LED is turned on if both button1 (on BCM pin 25/wPi P6) **and** button2 (on BCM pin 5/wPi P21) are high. Both conditions (on the left and right of the **and** operator) must be true in order for the entire if-statement to be true.

Of course, implementing the *or* gate is just as easy. In fact, there is no need to change the circuit! We simply switch the *and* operator with the *or* operator. The rest of the logic is exactly the same:

```
if (GPIO.input(button1) == GPIO.HIGH or GPIO.input(button2) == GPIO.HIGH):
```

Did you know?

When circuits are continuously toggled (such as when an LED is turned on and off, over and over), we can refer to the portion of time that the circuit is on as a duty cycle. Formally, a **duty cycle** is the percentage of one period in which a signal is active. A period is the time it takes for a signal to complete an on-and-off cycle. In a simple LED circuit, a duty cycle of 50% means that the LED turns on and off for the same amount of time (e.g., the LED turns on for one second, off for one second, and so on). A duty cycle of 25% means that the LED turns on 25% of the time (e.g., the LED turns on for 0.25s, off for 0.75s, and so on).

Homework: Blink!

Implement a single LED, single switch circuit and write a Python program that does the following:

(1) The LED should blink continuously such that it is on for 0.5s and off for 0.5s;
(2) When the switch is pressed (closed), it should change the LED's blink rate so that it is on for 0.1s and off for 0.1s (i.e., it should make the LED blink faster); and
(3) When the switch is released (open), the LED should go back to blinking at the original rate of 0.5s on and 0.5s off.

Submit your Python source file (i.e., the one with a .py extension) through the upload facility on the web site.

Raspberry Pi Activity Assignment: LED the Way

Name: _____

Calculations

Suppose that a circuit has a power source voltage of 9V, and an LED with a forward voltage (i.e., voltage drop) of 2.5V that requires 25mA of current for optimum brightness. In the space below, calculate the resistance of the series resistor required. State the formula used as the basis for your answer, identify all units, and show all work!

In the space below, calculate the power dissipated by the resistor. State the formula used as the basis for your answer, identify all units, and show all work!

In the space below, calculate the power dissipated by the LED. State the formula used as the basis for your answer, identify all units, and show all work!

Truth tables
Fill in the truth table for the *and* gate below:

A	B	Z

Fill in the truth table for the *xor* (exclusive *or*) gate below:

A	B	Z

Circuits
Using the figure below, draw the single-switch, single-LED circuit in which the RPi was responsible for detecting the state of the switch (as input) and accordingly affecting the blink rate of the LED (as output):