

## Raspberry Pi Activity 2: My Binary Addiction...Reloaded

In this activity, you will re-implement the one-bit binary adder that was the subject of Raspberry Pi Activity 1: My Binary Addiction. The difference? You will do it in Python instead of Scratch this time. As in that activity, you will need the following items:

- Raspberry Pi B v2 with power adapter;
- LCD touchscreen with power adapter and HDMI cable;
- Wireless keyboard and mouse with USB dongle;
- USB-powered speakers (optional);
- Wi-Fi USB dongle;
- MicroSD card with NOOBS pre-installed;
- Breadboard (several, actually – work with a partner);
- T-shaped GPIO-to-breadboard interface board with ribbon cable; and
- LEDs, resistors, switches, and jumper wires provided in your kit.

Regarding the electronic components, you will need the following:

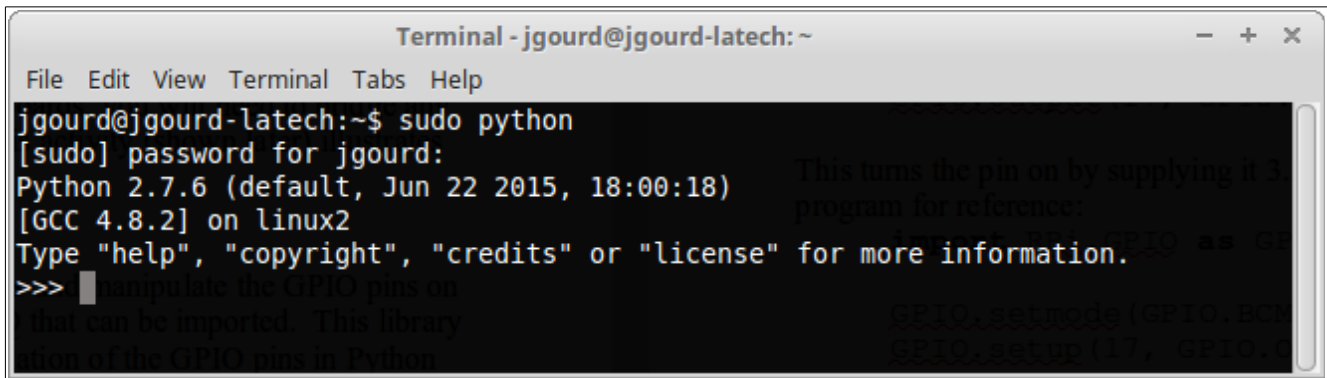
- 2x red LEDs;
- 2x green LEDs;
- 2x yellow LEDs;
- 2x blue LEDs;
- 1x extra LED (work with a partner);
- 2x push-button switches;
- 9x 220 $\Omega$  resistors; and
- 9x jumper wires.

**Like last time, it will greatly help if you work with a partner.** Why, you ask? Well, you require a bit more breadboard space than you have given the tiny one provided in your kit. By working with a partner, you can combine breadboards and get more space! Recall that the breadboards actually snap together when properly oriented. Note that, if you do use two breadboards, you will need to bridge any connections required (e.g., a ground row). The layout diagram for this activity (shown later) illustrates this.

### GPIO in Python

Before we get to the binary adder, 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. It is important to note that manipulation of the GPIO pins in Python requires superuser privileges. Therefore, the Python interpreter or IDLE must be executed with such privileges. The simplest way to do this is through a terminal.

Here's how the Python interpreter would be executed with superuser privileges:



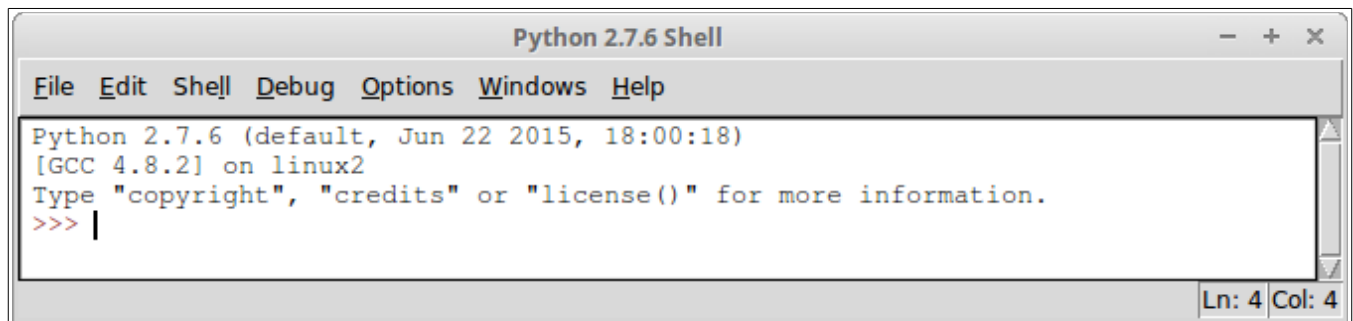
```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ sudo python
[sudo] password for jgourd:
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT)
```

And here's how IDLE would be executed with superuser privileges (through the terminal):



```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ sudo idle
[sudo] password for jgourd:
```

The IDLE interface is exactly the same as executing it without superuser privileges; however, manipulation of the GPIO pins is now possible without getting errors:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> |
```

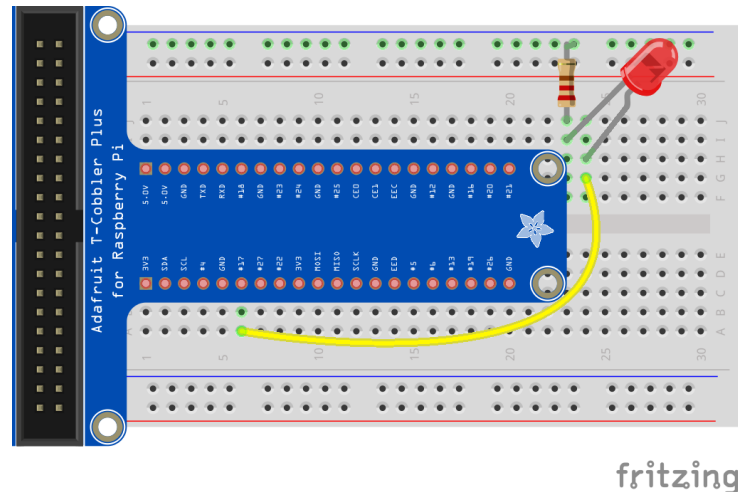
Importing the RPi.GPIO library is as simple as including the following import statement (typically done at the beginning of a Python program):

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

Recall that there are two methods of referring to the GPIO pins on the RPi. We saw this in previous activities. For example, we referred to one pin as both GPIO 25 and also as pin 22. The easiest method of referring to GPIO pins in Python is by using their GPIO *n* designations (e.g., GPIO 25). This is known as the Broadcom (or BCM) method in the RPi.GPIO library. To refer to GPIO pins in this way, we must set the mode or pin layout as follows:

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

To begin, let's light an LED. Connect one to GPIO 17, then to a resistor, then to ground as follows:



To turn the LED on, we must configure or setup GPIO 17 to be an **output** pin as follows:

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

And now 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, etc. We already know how to turn an output pin on and off. 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 (*while (True)* is never false!). **To stop the program, we can press Ctrl+C.** However, when we do so we really need to cleanup and reset the GPIO pins at the end of our program. This is as easy as adding the following statement:

```
GPIO.cleanup()
```

Often, it is standard practice to assign GPIO pin numbers to meaningful variables. For example, we can assign GPIO 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)

GPIO.cleanup()
```

What about getting input from the GPIO pins (e.g., detecting the pressing of a pushbutton switch)? It's also quite simple! The first step is to set the specified pin as an input pin as follows:

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

Note that the pin has been assigned to the variable `button` to make it more meaningful. It makes it easier to maintain our code later on. The previous statements setup pin 25 as an input pin. Of course, we would need to wire it properly. In this case, we could wire a pushbutton switch such that one side is connected to ground and the other to the input pin. Let's call this case 1. In Python, we are not restricted with input pins as we were in Scratch. We can, for example, connect one side of the switch to 3.3V and the other to the input pin. Let's call this case 2. In case 1, the pin should read high by default. Pressing the switch would connect the pin to ground and set it to low. We could then read this state to detect the pushing of the button. In case 2, the pin should read low by default. Pressing the switch would connect the pin to 3.3V and set it to high. Again, we could then read this state to detect the pushing of the button.

Since both ways 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, we use what is called a pull-up resistor (which *pulls* the state of the switch *up* to 3.3V). To connect the pin to ground, we use a pull-down resistor (which *pulls* the state of the switch *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_UP)
# for case 2
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

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.setup(in_pins, GPIO.IN)
```

This sets GPIO 17 and 18 as output pins and GPIO 22 and 27 as input 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.setup(out_pins, GPIO.HIGH)
```

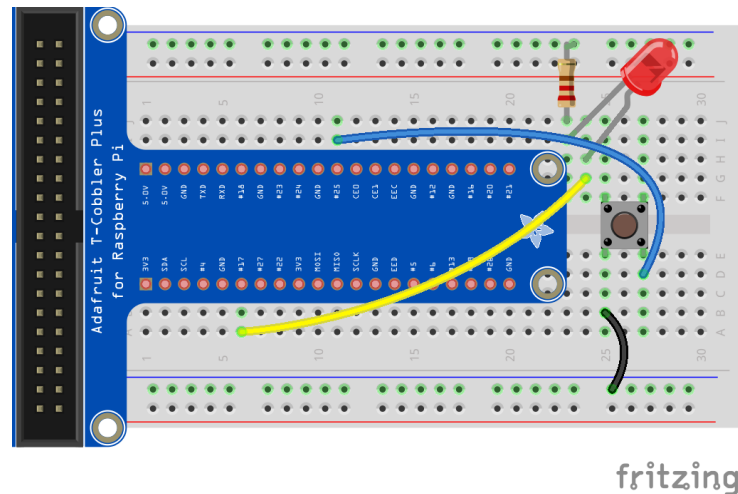
Setting GPIO 17 high and GPIO 18 low can be done in one statement as follows:

```
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):  
    ...
```

Let's see how this can work with a real circuit. Here's one that you've implemented before:



The pushbutton switch is connected to GPIO 25. The other side is connected to ground. In order to properly read the state of the input pin, we will need to configure it with a pull-up resistor so that the default value for the input pin is high:

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

From here, we can check the input pin's state (forever):

```
while (True):  
    if (GPIO.input(button) == GPIO.HIGH):  
        print "Switch is not pressed."  
    else:  
        print "Switch is pressed!"
```

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 uses 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.

Here's a program that blinks an LED once every second (i.e., on for 0.5 second, off for 0.5 second) by default. If the switch is pressed, it blinks the LED faster, once every 0.5 second (i.e., on for 0.25 second, off for 0.25 second):

```
import RPi.GPIO as GPIO  
from time import sleep  
  
# set the LED and switch pin numbers  
led = 17
```

```

button = 25

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the LED and switch pins
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# we'll discuss this later, but the try-except construct allows
# us to detect when Ctrl+C is pressed so that we can reset the
# GPIO pins
try:
    # blink the LED forever
    while (True):
        # the delay is 0.5s if the switch is not pressed
        if (GPIO.input(button) == GPIO.HIGH):
            delay = 0.5
        # otherwise, it's 0.25s
        else:
            delay = 0.25

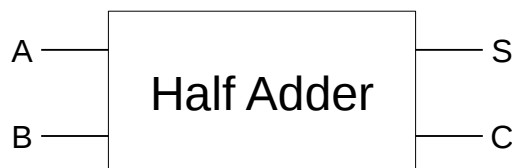
        # blink the LED
        GPIO.output(led, GPIO.HIGH)
        sleep(delay)
        GPIO.output(led, GPIO.LOW)
        sleep(delay)
# detect Ctrl+C
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()

```

You probably noticed the **try-except** construct. A comment notes that it will be discussed later (and it will!). For now, it's enough to know that such a construct is used to group statements that may cause an exception (i.e., some sort of abnormal event during runtime). In the case of the program above, the abnormal event is the user pressing Ctrl+C (which breaks out of the program). We can detect this and execute statements subsequently to do things like cleaning up and/or resetting the GPIO pins.

### The adder...reloaded

Recall the single-bit half adder shown in the previous activity:



It takes two single-bit inputs, A and B, and produces two outputs, S (the sum bit) and C (the carry bit). The half adder has the following truth table:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Most general purpose programming languages allow bitwise operations (discussed in a recent lesson). That is, they can take Boolean inputs (like A and B) and implement the logic of primitive gates (e.g., *and* and *or*). Recall that Scratch doesn't support bitwise operations. Because of this, we had to create a rather elaborate script made up of many if-statements in order to implement the half adder. Python, however, supports bitwise operators! This allows us to significantly reduce the amount of code required to implement the half adder:

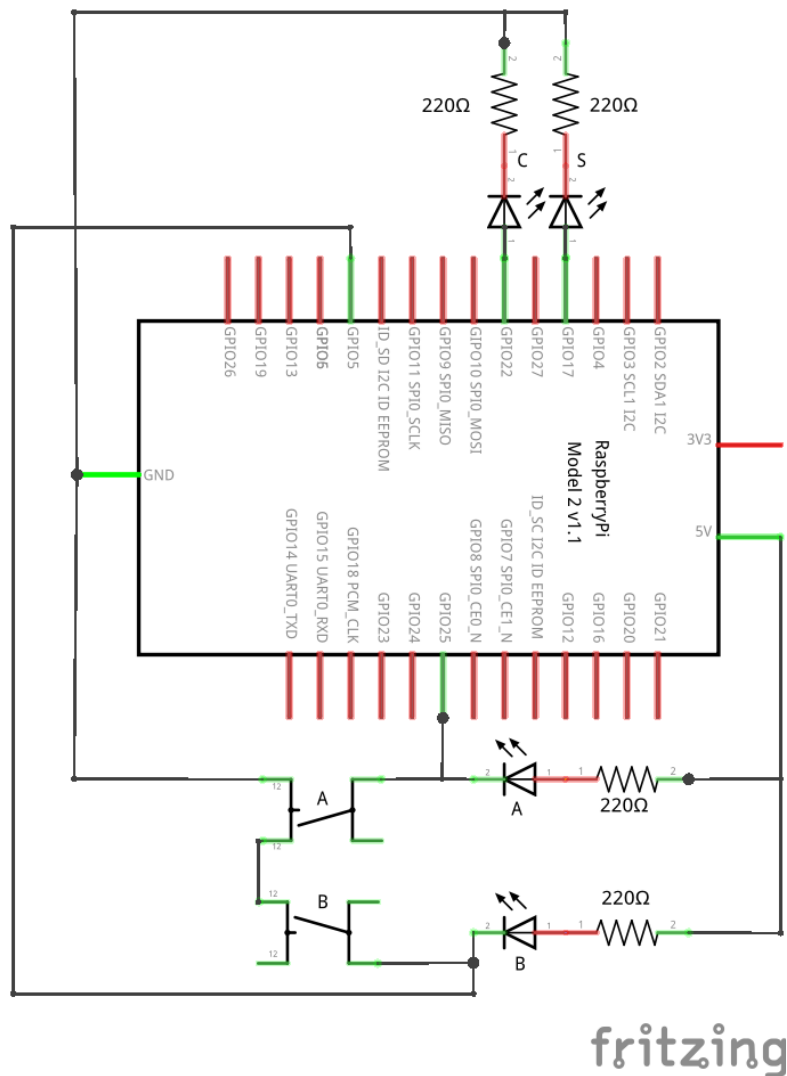
```
S = A ^ B # ^ is the xor bitwise operator
C = A & B # & is the and bitwise operator
```

That's it! Just two statements. S is assigned the result of A *xor* B, and C is assigned the result of A *and* B. Although we structured our half adder such that the *xor* functionality was built using the three primitive gates (*and*, *or*, and *not*), Python has the *xor* bitwise operator that we can use directly! This is quite useful.



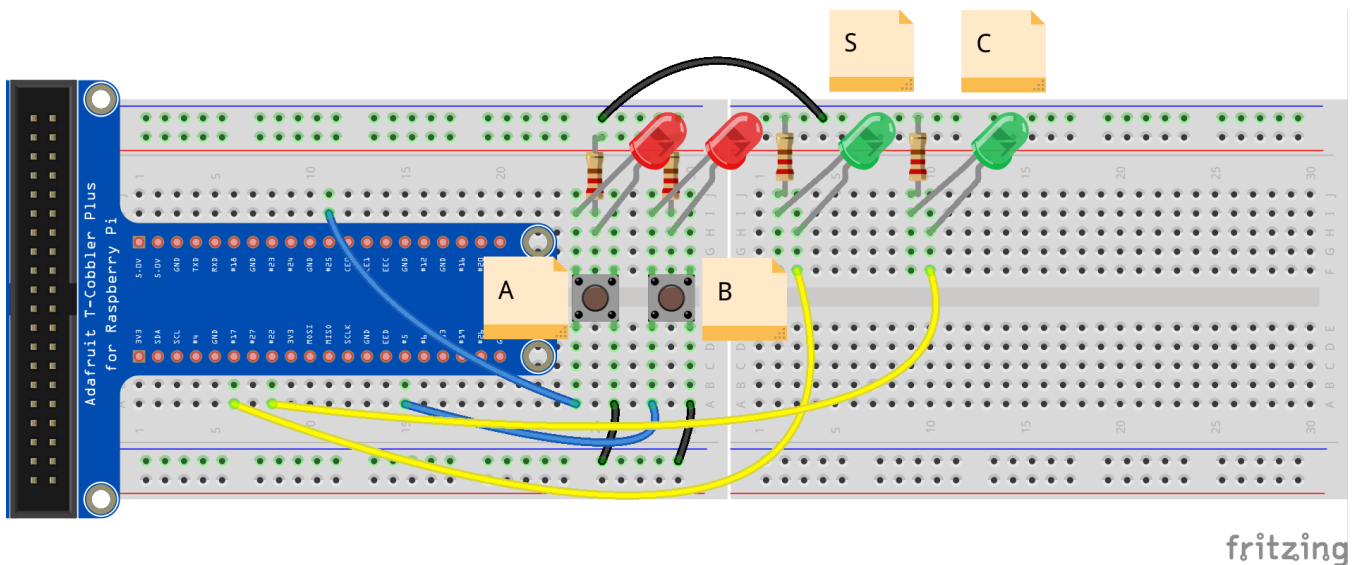
## The circuit...reloaded

As in the previous activity, you will implement the following circuit:



Note that the LEDs labeled C and S (the outputs) are green LEDs, and the LEDs labeled A and B (the inputs) are red LEDs. Again, the input LEDs are also connected to the push-button switches. Since the switches are connected to ground (i.e., they complete the circuit both to the input pins, GPIO 25 and GPIO 5, and to the input LEDs when closed), then the LEDs must be connected such that the cathode (negative side) is matched with the switch (i.e., the positive side is connected to +5V). This is illustrated in the circuit above. Pay close attention to polarity (i.e., where the negative and positive terminals of electronic components are) and wiring. Recall that the position of the resistor (either on the negative or positive side of the LED) doesn't matter. In the circuit above, the resistors are placed on the positive side of the LEDs.

This circuit can be topologically laid out on a breadboard in a number of ways. Here's one way to do so (again, it will not be possible unless you join with a partner and combine breadboards):



Recall that the labels (A, B, S, and C) are strictly informative (i.e., they serve no function other than to provide situational awareness). It should be clear that the left push-button switch represents the bit input A, the right push-button switch represents the bit input B, the green LED on the left represents the bit output S, and the green LED on the right represents the bit output C. The red LEDs are wired to the push-button switches and provide feedback of the state of A and B (i.e., the left LED corresponds to the left push-button switch, and vice versa).

Input A (the push-button switch on the left) is connected to GPIO 25. Input B (the push-button switch on the right) is connected to GPIO 5. Output S (the green LED on the left) is connected to GPIO 17. Output C (the green LED on the right) is connected to GPIO 22.

Again, note that the red LEDs are wired a bit differently than you may be used to! The resistors connect +5V to the anode (positive side) of the LED. The cathode (negative side) of the LED is connected to one pair of terminals on the push-button switch. The other pair is connected to ground. When the switch is closed (and the pairs are internally joined), current can flow through the resistor, the LED, the switch, and to ground. The switch is also connected to an input pin so that its state can be read programmatically.

### The code...reloaded

Here's the entire program for the half adder in Python:

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

# set the GPIO pin numbers
inA = 25
inB = 5
outS = 17
outC = 22

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)
```

```

# setup the input and output pins
GPIO.setup(inA, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(inB, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(outS, GPIO.OUT)
GPIO.setup(outC, GPIO.OUT)

# we'll discuss this later, but the try-except construct allows
# us to detect when Ctrl+C is pressed so that we can reset the
# GPIO pins
try:
    # keep going until the user presses Ctrl+C
    while (True):
        # initialize A, B, S, and C
        A = 0
        B = 0

        # set A and B depending on the switches
        if (GPIO.input(inA) == GPIO.LOW):
            A = 1
        if (GPIO.input(inB) == GPIO.LOW):
            B = 1

        # calculate S and C using A and B
        S = A ^ B          # A xor B
        C = A & B          # A and B

        # set the output pins appropriately
        # (to light the LEDs as appropriate)
        GPIO.output(outS, S)
        GPIO.output(outC, C)
# detect Ctrl+C
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()

```

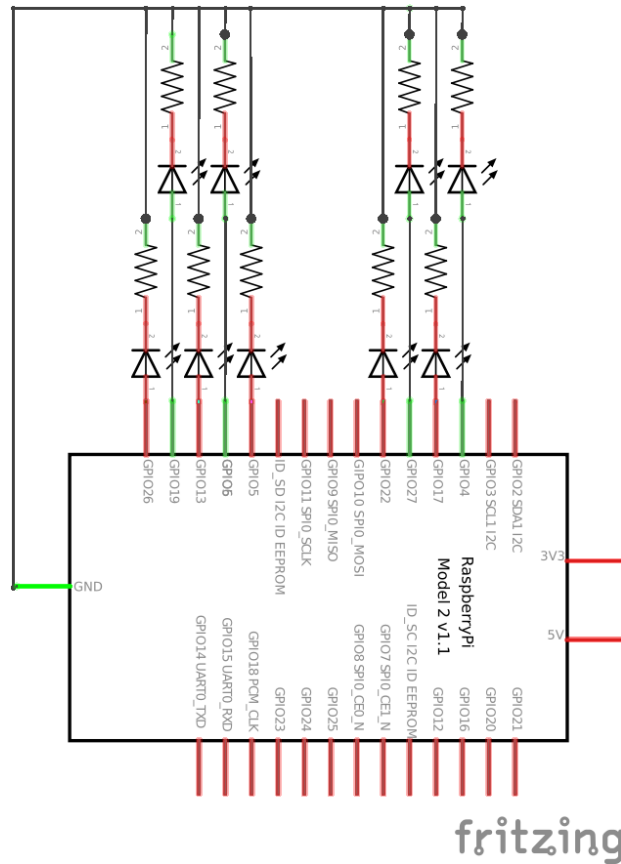
After importing the required libraries, variables that map to GPIO pins are declared (*inA* representing the pin connected to switch A, *inB* representing the pin connected to switch B, *outS* representing the pin connected to LED S, and *outC* representing the pin connected to LED C). Next, the pins are setup (as either input or output pins). Since the switches are wired to ground, the input pins are setup with a pull-up resistor. That is, their default value will be high at 3.3V. When a switch is pressed, this will bring down the pin to 0V. Since the LEDs representing the inputs are also wired to positive voltage, they will light when the switches are pressed.

Next, the program detects the switch states and turns on the LEDs as appropriate. A and B are initialized to 0. If a switch is pressed, its corresponding input (A or B) is changed to 1. The values of S and C are

then calculated (as  $A \text{ xor } B$  for S, and  $A \text{ and } B$  for C). Finally, the LEDs are triggered appropriately depending on the values of S and C.

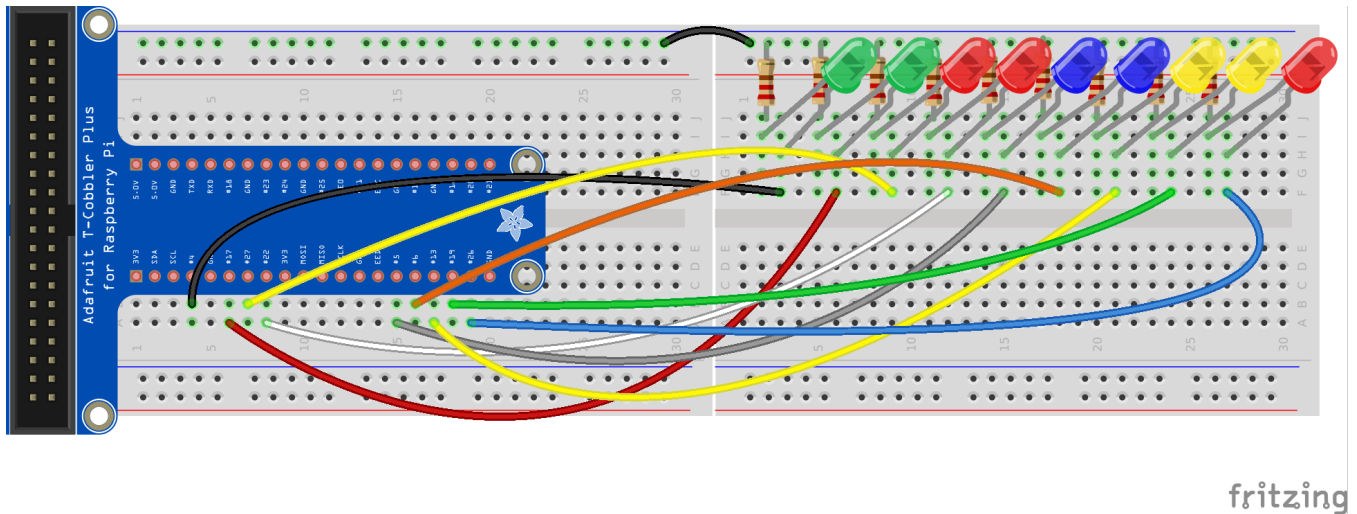
### Extending this a bit...reloaded

Take a look at the following circuit from the last activity:



As in the last activity, there are nine LEDs, all connected to GPIO pins, to a resistor that is connected to ground.

Now take a look at a layout diagram of this circuit:



As in the last activity, it will be much easier if you partner up for this one for additional breadboard space, and also for an extra LED! The LEDs will represent the sum of two 8-bit numbers, with the least significant bit represented by the LED all the way to the right. For example, if the LEDs were to represent the sum of  $94 + 113 = 207$  (see the table below), then the state of the LEDs would be: off, on, on, off, off, on, on, on, on. The overflow bit (on the left) would be 0 (off).

	Binary									Decimal	
	2 <sup>9</sup>	2 <sup>8</sup>	2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	
Carry		0	1	1	1	0	0	0	0		1
1st number			0	1	0	1	1	1	1	0	94
2nd number			0	1	1	1	0	0	0	1	113
Sum		0	1	1	0	0	1	1	1	1	207

Or for example, if the LEDs were to represent the sum of  $150 + 150 = 300$  (see the table below), then the state of the LEDs would be: on, off, off, on, off, on, on, off, off. In this case, the overflow bit would be 1 (on).

	Binary									Decimal	
	2 <sup>9</sup>	2 <sup>8</sup>	2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	
Carry		1	0	0	1	0	1	1	0		1
1st number			1	0	0	1	0	1	1	0	150
2nd number			1	0	0	1	0	1	1	0	150
Sum		1	0	0	1	0	1	1	0	0	300

In the last activity, you did this in Scratch (although you implemented the entire program, you were only asked to generate the code for the full adder). In this activity, you will repeat what you did last time; however, you will implement it in Python! Take a look at the following code (note the header – that you should fill in):

```
#####
# Name:
# Date:
# Description:
#####
import RPi.GPIO as GPIO      # bring in GPIO functionality
from random import randint    # allows us to generate random integers

# function that defines the GPIO pins for the nine output LEDs
def setGPIO():
    # define the pins (change these if they are different)
    gpio = [4, 17, 27, 22, 5, 6, 13, 19, 26]
    # set them up as output pins
    for i in gpio:
        GPIO.setup(i, GPIO.OUT)

    return gpio

# function that randomly generates an 8-bit binary number
def setNum():
    # create an empty list to represent the bits
    num = []
    # generate eight random bits
    for i in range(0, 8):
        # append a random bit (0 or 1) to the end of the list
        num.append(randint(0, 1))

    return num

# function that displays the sum (by turning on the appropriate LEDs)
def display():
    for i in range(len(sum)):
        # if the i-th bit is 1, then turn the i-th LED on
        if (sum[i] == 1):
            GPIO.output(gpio[i], GPIO.HIGH)
        # otherwise, turn it off
        else:
            GPIO.output(gpio[i], GPIO.LOW)
```

```

# function that implements a full adder using two half adders
# inputs are Cin, A, and B; outputs are S and Cout
# this is the function that you need to implement
def fullAdder(Cin, A, B):
    #####
    # write your code here!!!!
    #####

    return S, Cout      # yes, we can return more than one value!

# controls/drives the addition of each 8-bit number to produce a sum
def calculate(num1, num2):
    Cout = 0             # the initial Cout is 0
    sum = []             # initialize the sum
    n = len(num1) - 1    # the position of the right-most bit of num1

    # step through each bit, from right-to-left
    while (n >= 0):
        # isolate A and B (the current bits of num1 and num2)
        A = num1[n]
        B = num2[n]
        # set the Cin (as the previous half adder's Cout)
        Cin = Cout

        # call the fullAdder function that takes Cin, A, and B...
        # ...and returns S and Cout
        S, Cout = fullAdder(Cin, A, B)

        # insert the sum bit, S, at the beginning (index 0) of sum
        sum.insert(0, S)

        # go to the next bit position (to the left)
        n -= 1

    # insert the final carry out at the beginning of the sum
    sum.insert(0, Cout)

    return sum

# use the Broadcom pin scheme
GPIO.setmode(GPIO.BCM)

# setup the GPIO pins
gpio = setGPIO()

# get a random num1 and display it to the console
num1 = setNum()
print "      ", num1

```

```

# get a random num2 and display it to the console
num2 = setNum()
print "+      ", num2

# calculate the sum of num1 + num2 and display it to the console
sum = calculate(num1, num2)
print "= ", sum

# turn on the appropriate LEDs to "display" the sum
display()

# wait for user input before cleaning up and resetting the GPIO pins
raw_input("Press ENTER to terminate")
GPIO.cleanup()

```

First, take a look at the bit of code near the bottom (not the function definitions). The program first sets the GPIO pins by calling the function `setGPIO`. This function defines a list that contains the pins corresponding to the nine output LEDs. It then iterates over them (via a for loop) and sets them up as output pins. The list is then returned to the main part of the program (note that the variable `gpio` contains this list).

Next, the first number is generated by calling the function `setNum`. This function iteratively builds a list of eight random bits. Once finished, the list is returned to the main part of the program (note that the variable `num1` contains this list). The same occurs for the second number. Note the use of the `randint` function. It is imported through the *random* library. Its format is `randint(x, y)`, where  $x$  and  $y$  are the lower and upper values specified by the interval  $[x, y]$  to select a random integer from. For example, `randint(5, 44)` selects a random integer from 5 to 44.

Next, the sum is calculated by calling the function `calculate` (passing in the two numbers as parameters). This function is similar to the one in the last activity. It functions as the 8-bit adder that chains together eight full adders. It cycles through the two numbers from right-to-left, each time (i.e., for each bit) calling the `fullAdder` function. This function is provided  $C_{in}$ ,  $A$ , and  $B$  as input parameters. It implements a full adder (made up of two half adders) and calculates (and returns) values for  $S$  and  $C_{out}$ . **Your task, as in the last activity, is to implement the `fullAdder` function.**

Finally, the `display` function is called, which turns on the appropriate LEDs that correspond to the bits that are on (1) in the variable `sum`.

Note the `raw_input` function near the end of the program. You have previously seen the `input` function (which allows user input to be provided and stored to a variable). The function `raw_input` is similar; however, it treats any input as a string. The regular `input` function attempts to evaluate the user input (which, for example, could be an integer). The function `raw_input` works in Python 2.7.x; however, it has been removed in Python 3.x (which only has the `input` function).



If you look closely, you will note that this Python version of the 8-bit adder is structured similarly to the version in the last activity (in Scratch).

### **Homework: Full Adder...Reloaded**

Create the *fullAdder* function that implements a full adder (that is made up of two half adders). Of course, you will also need to implement the program previously covered to test appropriately. **A template is provided on the class web site.**

If you wish, you may modify the source code provided and your circuit to add two 4-bit numbers instead of two 8-bit numbers. Therefore, the output sum will be a 5-bit number (four bits plus the overflow bit). This will allow you to reduce the number of LEDs required to five instead of nine. It may also be possible to layout the circuit on a single breadboard, thereby not requiring you to work with a partner.

**You are to submit your Python source code only (as a .py file) through the upload facility on the web site.**