Raspberry Pi Activity 1: My Binary Addiction

In this activity, you will implement a one-bit binary adder using LEDs, resistors, and push-button switches. 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 220Ω resistors; and
- 9x jumper wires.

**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! In fact, 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.

**The adder**

Recall the single-bit half adder shown in a previous lesson:

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:

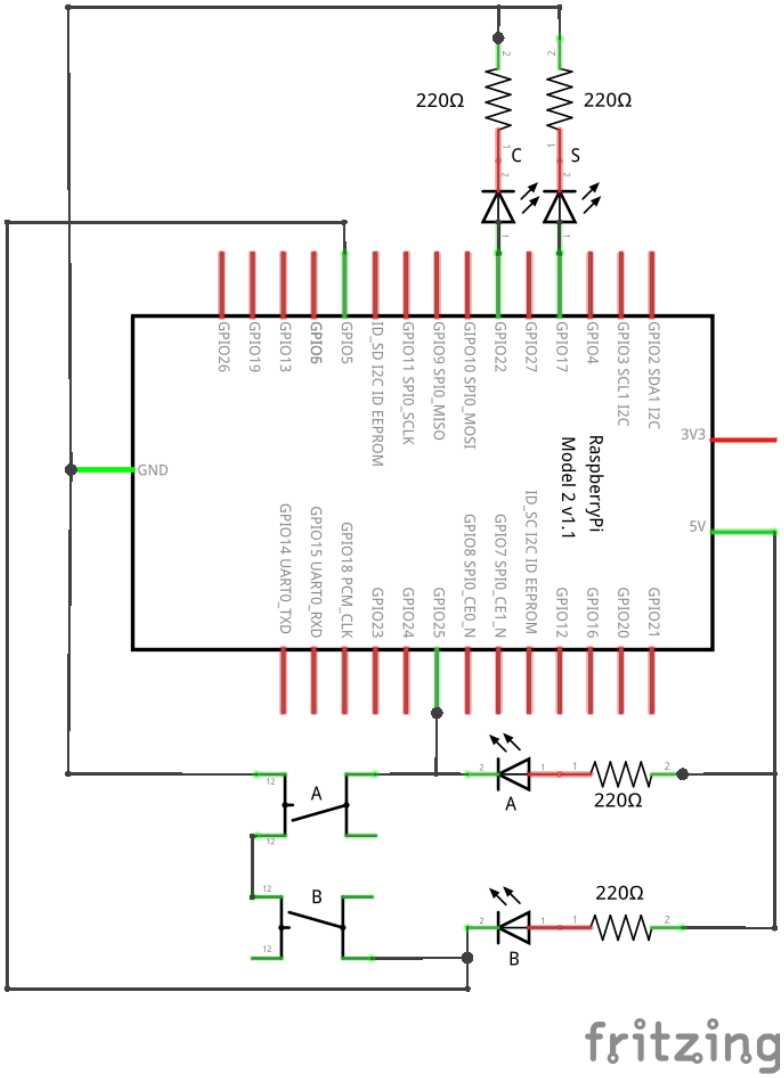
| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Most general purpose programming languages allow what are called bitwise operations. That is, they can take Boolean inputs (like A and B) and implement the logic of primitive gates (e.g., *and* and *or*). Unfortunately, Scratch doesn't support bitwise operations. However, we have seen that if-statements are related to logic gates. For example, we can evaluate if one condition *and* another are true. If and only if both are true will the entire condition be true (and the statements in the true part of the if-statement will be executed). Therefore, one way to implement the truth table for a half adder is as follows:

```
 1: if A is 0 and B is 0
 2: then
 3:     S ← 0
 4:     C ← 0
 5: else
 6:     if A is 1 and B is 1
 7:     then
 8:         S ← 0
 9:         C ← 1
10:     else
11:         S ← 1
12:         C ← 0
13:     end
14: end
```

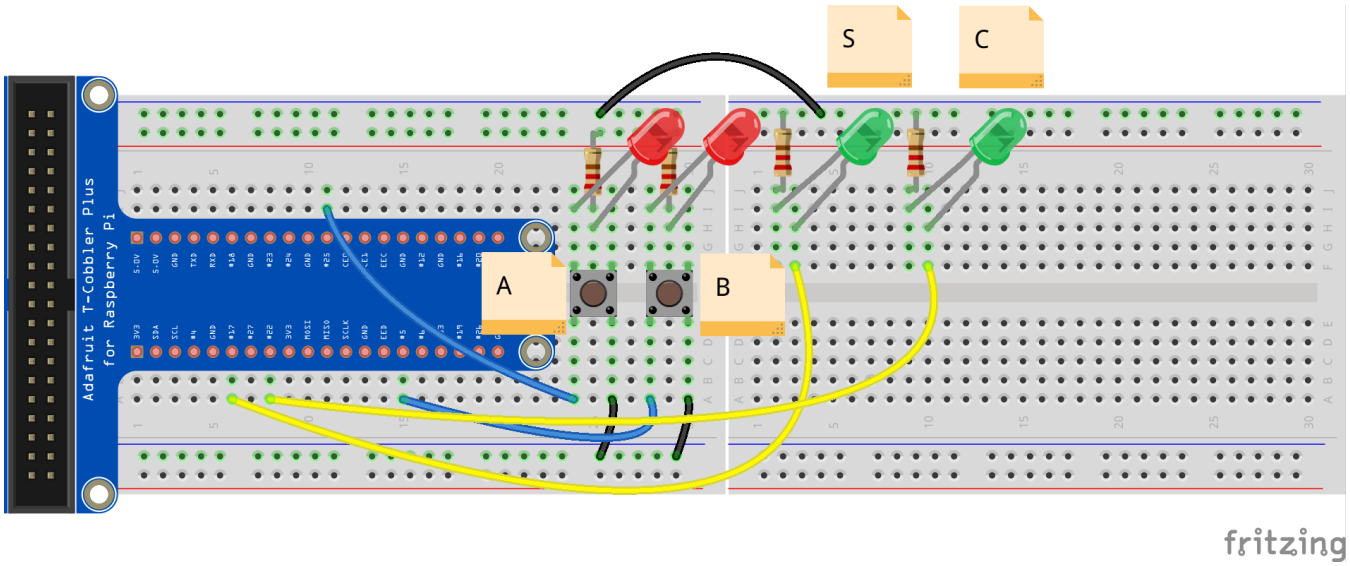
Notice that this basically handles each row of the truth table above. The first if-statement handles the first row of the truth table (when A and B are both 0), and sets S and C to 0. The second if-statement handles the last row of the truth table (when A and B are both 1), and sets S to 0 and C to 1. The last case (the else part of the second if-statement) handles the two middle rows of the truth table, where either A or B is 1 (but not both), and sets S to 1 and C to 0. This is how we will implement a half adder in Scratch.

**The circuit**
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):




Note 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 (pin 22). Input B (the push-button switch on the right) is connected to GPIO 5 (pin 29). Output S (the green LED on the left) is connected to GPIO 17 (pin 11). Output C (the green LED on the right) is connected to GPIO 22 (pin 15).

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.

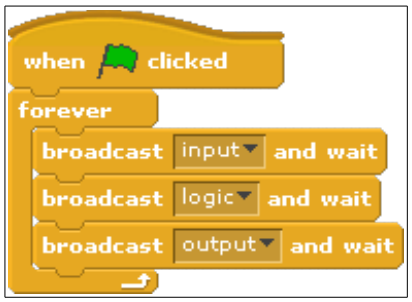
**The code**

So how do we make this all work? Once you have implemented the circuit above using the layout diagram as a guide, launch ScratchGPIO.

The first thing to do is to setup a general algorithm that will run forever:

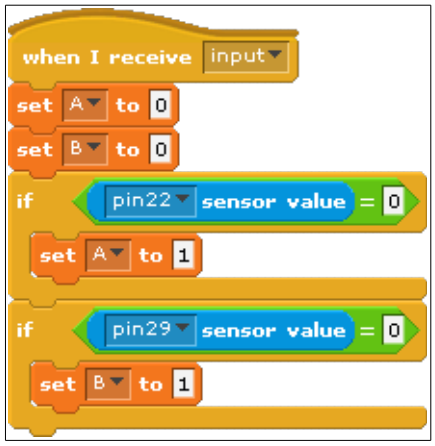
```
1: Check the status of the input switches, A and B
2: Apply the adder logic to determine the outputs, S and C
3: Toggle the LEDs appropriately
```

This can be accomplished with the main part of the script as show below:




Notice how this main part is driving the logic. We aren't specifying exactly how to check the inputs, apply the logic, and set the outputs...yet. We often use this kind of development method, where the main part of the program acts as the driver. Each part represents a potentially complex instruction that is described elsewhere (such as in a subprogram).

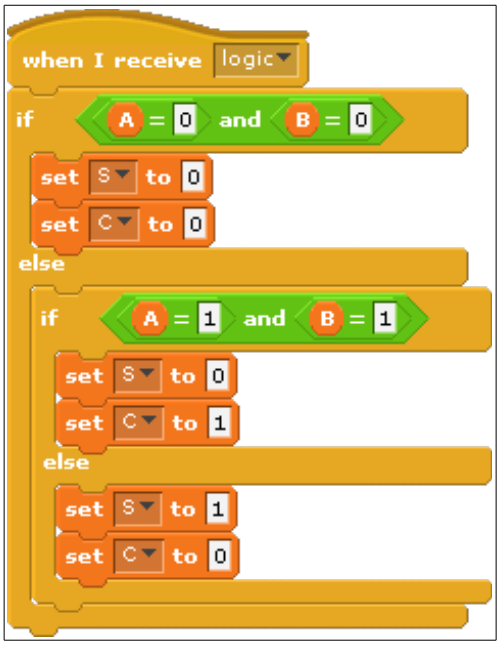
Let's work on the *input* subprogram (or script, in Scratch):




The purpose of the *input* subprogram is to sense the state of the push-button switches and set the input variables, A and B, appropriately. Notice that two variables were created to map to these inputs: A and B. This was done in the variables blocks group. In addition, the output variables, S and C, were also created:

The *input* subprogram initializes both A and B to 0. It then checks the state of the push-button switches on pin 22 (GPIO 25) and pin 29 (GPIO 5). The inputs A and B are appropriately changed depending on the state of the push-button switch (open or closed). Recall that the pin reads a 0 if the switch is closed and a 1 if it is open. That is, the default state of an input pin is high (1). When a switch is closed, the pin reads low (0). Therefore, if the state of an input pin is 0, then the corresponding input variable is set to 1 (which matches the adder logic).

The next subprogram to implement is the application of the adder logic to the outputs, S and C, based on the inputs, A and B:




The *logic* subprogram simply implements the following truth table using a series of if-statements as described earlier:

| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

In the first case, the subprogram checks to see if both A *and* B are 0. If so, it sets the values of S and C appropriately (to 0). This implements the first row of the truth table.

In the second case, it checks to see if both A *and* B are 1. If so, it sets the value of S to 0 and C to 1. This implements the last row of the truth table.

The last case implements the middle two rows (lines two and three) of the truth table. Note that these produce the same output, setting S to 1 and C to 0.

The last subprogram to implement is the application of the adder logic to the output LEDs (so that we can see the result of single-bit addition):


The *output* subprogram applies the values of S and C to the output pins, thereby reflecting them on the green LEDs. If the sum bit (output variable S) is 1, then the LED that represents S (on GPIO 17) is turned on; otherwise, it is turned off. Similarly, if the carry bit (output variable C) is 1, then the LED that represents C (on GPIO 22) is turned on; otherwise, it is turned off.

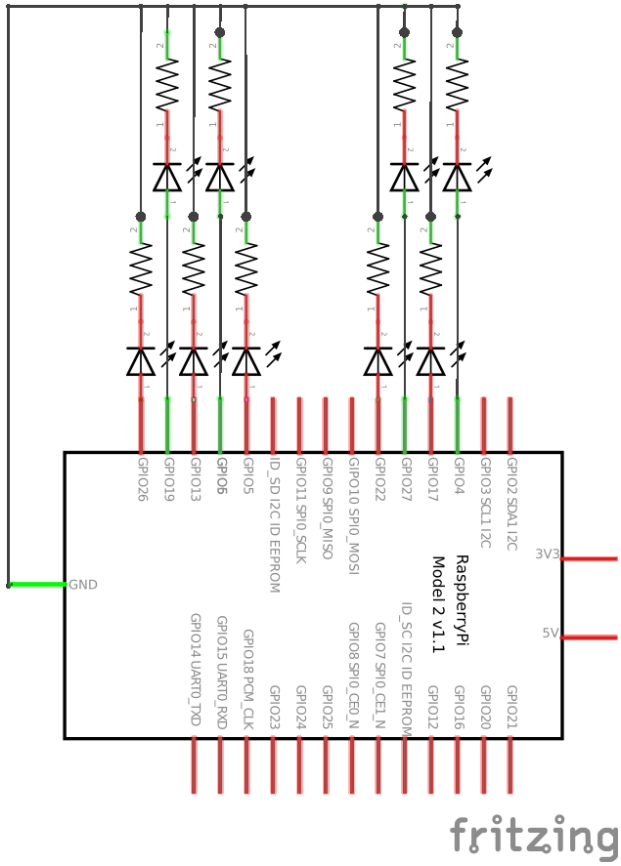
**How it works**

When the green flag is clicked, the main script runs forever. It is the only script that runs when the green flag is clicked (since it is the only script that implements then **when green flag clicked** block).

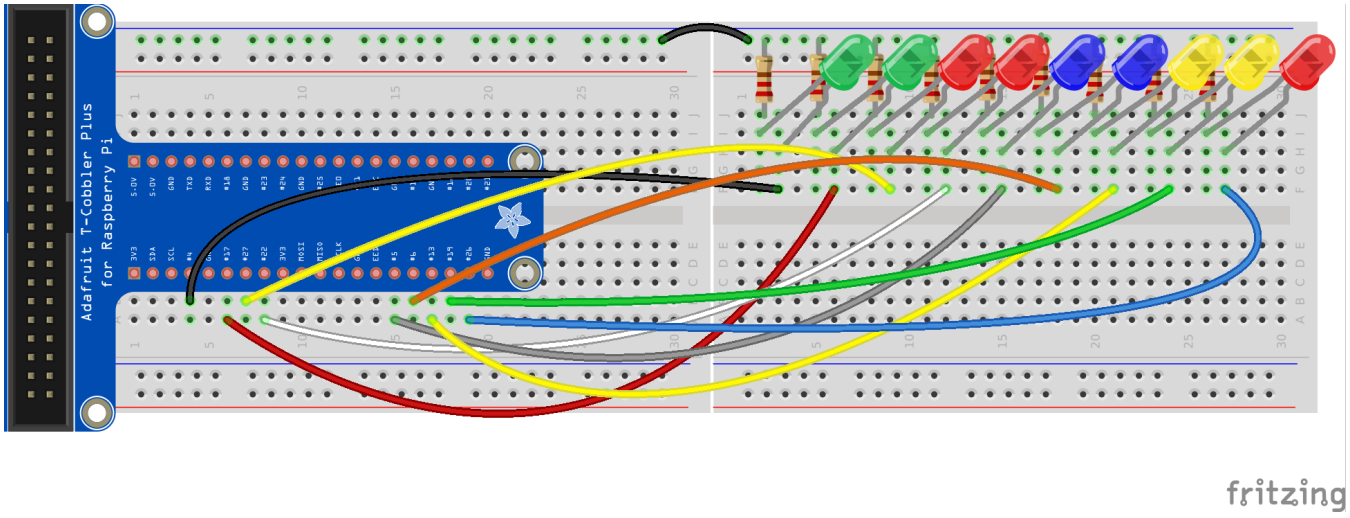
As it runs, it first executes the *input* subprogram (i.e., control is redirected there), which checks the state of the input pins (and therefore the push-button switches). The main script waits until the *input* subprogram completes (due to the **broadcast and wait** block). Once the *input* subprogram is finished, control goes back to the main script again. From there, it executes the *logic* subprogram to apply the adder logic and set the values of the output variables, S and C. When this is complete, control goes back to the main script yet again. Finally, the *output* subprogram is executed, which sets the output pins to match the state of the output variables, thereby turning the green LEDs on or off.

Control goes back to the main script one final time, and since the broadcasts are in a **forever** block, the entire process begins again (at least until the red stop sign is clicked, which halts the main script).

**Extending this a bit**
Take a look at the following circuit:




This time, 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:




Again, you'll have to 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
| 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 |

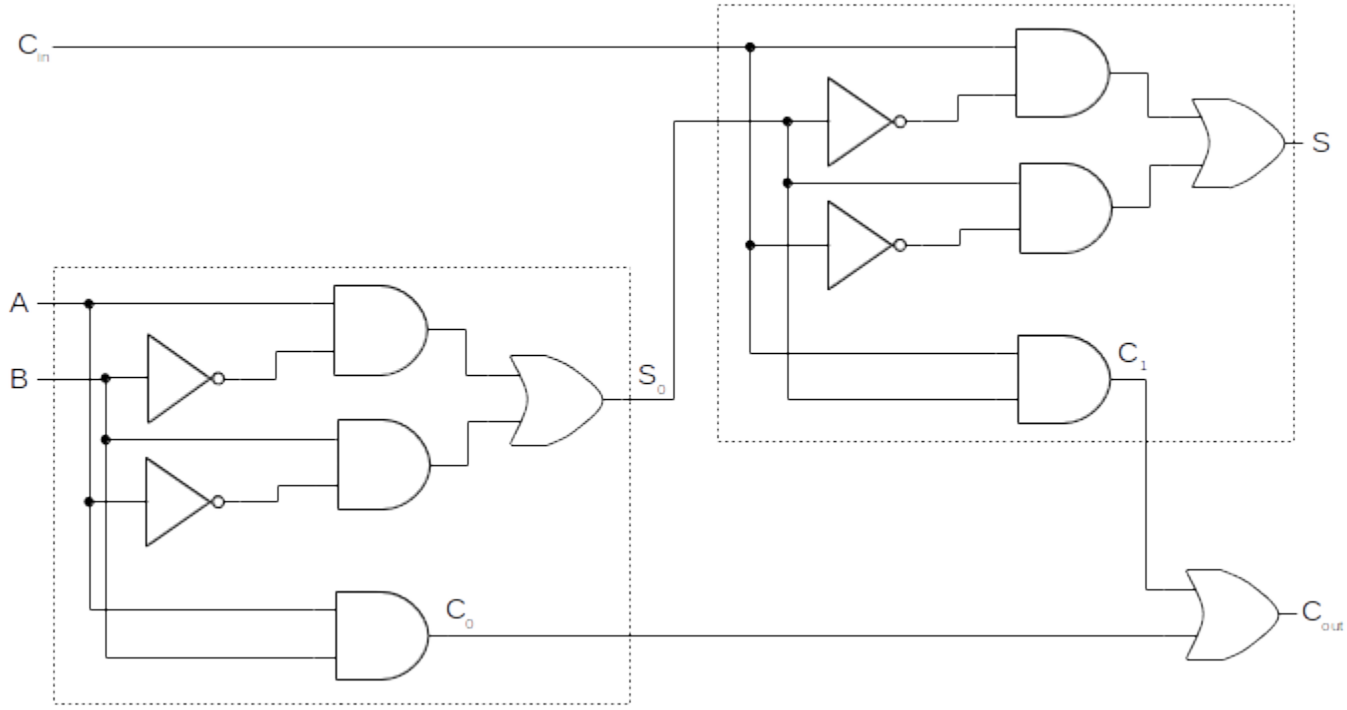
Or for example, if the LEDs were to represent the sum of 150 + 150 = 300 (see the table below), then the state of the LEDs would be: on, off, off, on, off, on, on, off, off. In this case, the overflow bit would be 1 (on).

| | Binary | | | | | | | | | | Decimal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
| 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 |

The basic algorithm at the main subprogram level can be written as follows:


The idea is to first set the GPIO pins (*setGPIO*) to match that of the actual connections (of the LEDs to the RPi). Next, we randomly generate two 8-bit numbers to add together (*setNums*). Then, we *calculate* the sum of the two numbers. And finally, we *display* the sum by turning on the appropriate LEDs.

To make this work, you will need to implement a full adder as described in a previous lesson:

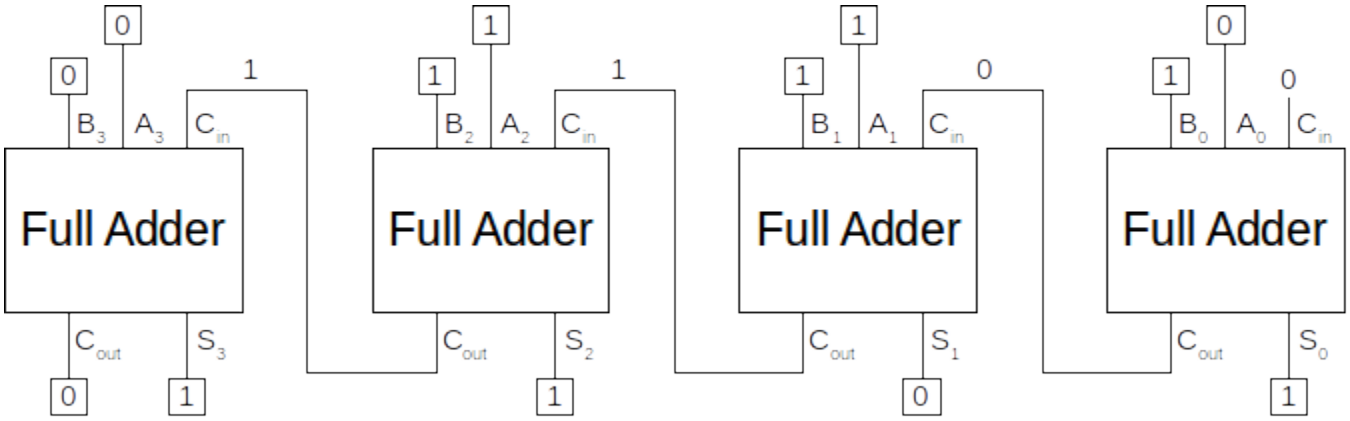



Recall that a full adder is made up of two half adders. One half adder computes the sum and carry of A and B. The sum is then brought into another half adder and added along with the carry in. The sum of this second half adder produces the actual sum of A and B plus the carry in. The carry out of this half adder is combined with the carry out of the first half adder through an *or* gate. The output is the carry out. You can take the script that implements the half adder (created in the first part of this activity) and extend it to a full adder.

The strategy here is to represent the two 8-bit numbers (and the sum) as lists in Scratch:

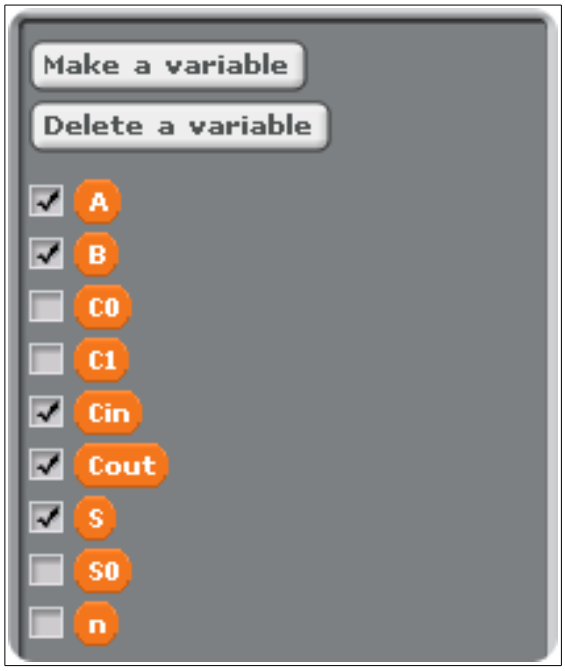

The list *num1* will represent the first number, *num2* will represent the second number, and *sum* will represent the sum of the two. The list *gpio* is used to store the numeric values of the GPIO pins that are connected to the LEDs. This will make more sense later.

Since each number is represented as a list, we will iterate through each, one bit at a time, and implement the full adder to produce a sum and carry out for each bit. Recall that the carry out is fed into the carry in for the next bit (to the left). We saw this when chaining full adders together to add two 4-bit numbers together:




In this activity, we are extending this to add two 8-bit numbers. The idea is the same.

You will need to create several variables to make this all work:

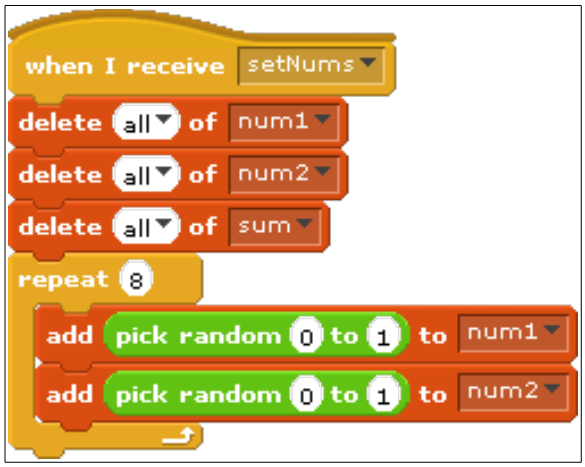



A, B, S, $C_{in}$, and $C_{out}$ are self-explanatory: they are the inputs to and outputs from the full adder. $C_0$, $C_1$, and $S_0$ are the intermediate values produced in the full adder. Recall that the first half adder produces a sum bit ($S_0$) and a carry bit ($C_0$). The second half adder produces a sum bit (S) and a carry bit ($C_1$). the two carry bits ($C_0$ and $C_1$) are combined to produce the carry out.

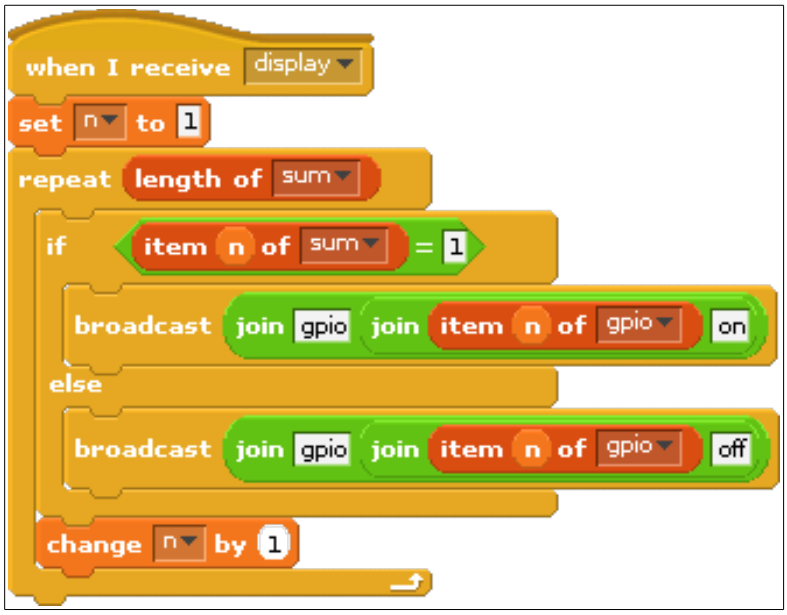
So how do we actually set the GPIO pins and generate two random 8-bit numbers? The following script creates the *gpio* list with the GPIO pins that the LEDs are connected to on the RPi:


Of course, you will need to make sure that your circuit is wired as shown in the layout diagram earlier (i.e., that the LEDs are indeed connected as shown, and in the same order as shown).

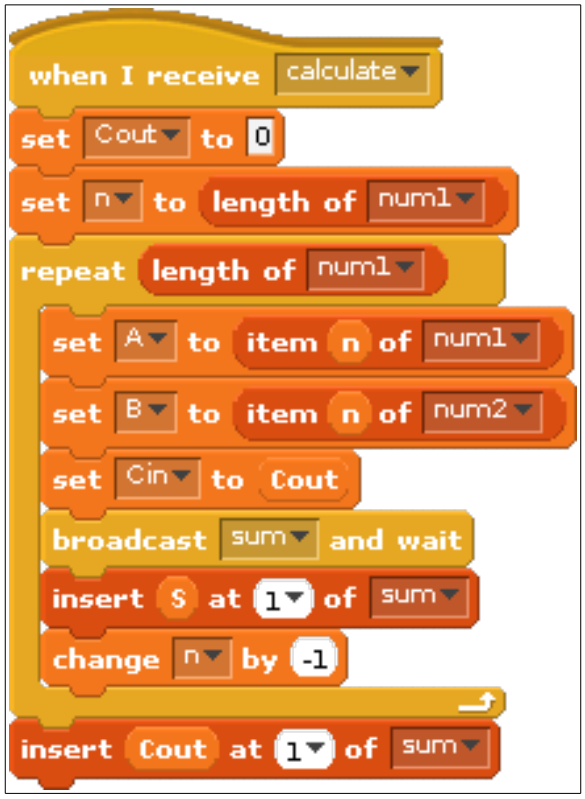
The following script generates two random 8-bit numbers and stores their bits in the lists *num1* and *num2*:




Once the sum has been calculated, the following script sends the appropriate logic (high or low) to the GPIO pins listed in the *gpio* list:




The only thing left to do is to actually implement the logic to add the two 8-bit numbers together. One way to do this is to separate the logic into two subprograms: one subprogram manages the linking of the full adders for each bit. That is, it iterates through the numbers from right-to-left, transfers control to the full adder, and generates the sum, one bit at a time:




The subprogram *calculate* initially sets $C_{out}$ to 0. This will be evident shortly. The variable *n* is then set to the length of any one of the numbers (it doesn't matter which one since they are the same bit length).

The idea is to iterate through each number from right-to-left, summing each bit one at a time. The variable *n* is used to keep track of which bit is currently being worked on. Each time, A is set to the current bit in *num1*, B is set to the current bit in *num2*, and $C_{in}$ is set to whatever $C_{out}$ was calculated to be in the last sum operation. Initially, $C_{out}$ is set to 0; therefore, the first $C_{in}$ is set to 0 (as is required when chaining full adders together). The *calculate* subprogram then calls another subprogram, *sum*, which implements the full adder, taking in $C_{in}$, A, and B, and generating S and $C_{out}$. Once the full adder has handled the sum and carry of the current bit, S is inserted at the front of the list *sum* (i.e., to the left – this list is built from right-to-left, as each full adder works through the bits), and *n* is decremented (which means that the next bit, to the left, will be worked on the next time we go through the *repeat-n* loop). Once all of the bits have been run through the full adder (and the sum has been completely calculated), the overflow bit of the sum (i.e., the left-most bit at the first position in the list *sum*) is set as the final $C_{out}$. This is why the circuit requires nine LEDs.

Again, note that this subprogram makes use of another subprogram called *sum*. The subprogram *sum* implements the full adder. As such, it takes as input three variables: $C_{in}$, A, and B. $C_{in}$ is either 0, if the current bit position is the least significant bit, or the carry out of the previous (to the right) bit otherwise. A is the current bit of *num1*, and B is the current bit of *num2*. A and B are run through a half adder to produce $S_0$ and $C_0$. $C_{in}$ and $S_0$ are run through another half adder to produce S and $C_1$. Finally, $C_0$ and $C_1$ are put through an *or* gate to produce $C_{out}$. S is stored as the current bit of the sum, and $C_{out}$ is fed in to the next bit's $C_{in}$.

The only goal of the *sum* subprogram is to produce valid values for S and $C_{out}$. and that brings us to your homework assignment for this activity.

**Homework: Full Adder**

Create the *sum* subprogram that implements a full adder. Of course, you will also need to implement the subprograms previously covered to test appropriately: *main*, *setGPIO*, *setNums*, *calculate*, and *display*.

If you wish, you may modify the subprograms 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 Scratch v1.4 (not v2.0!) file (with a .sb extension) through the upload facility on the web site.**