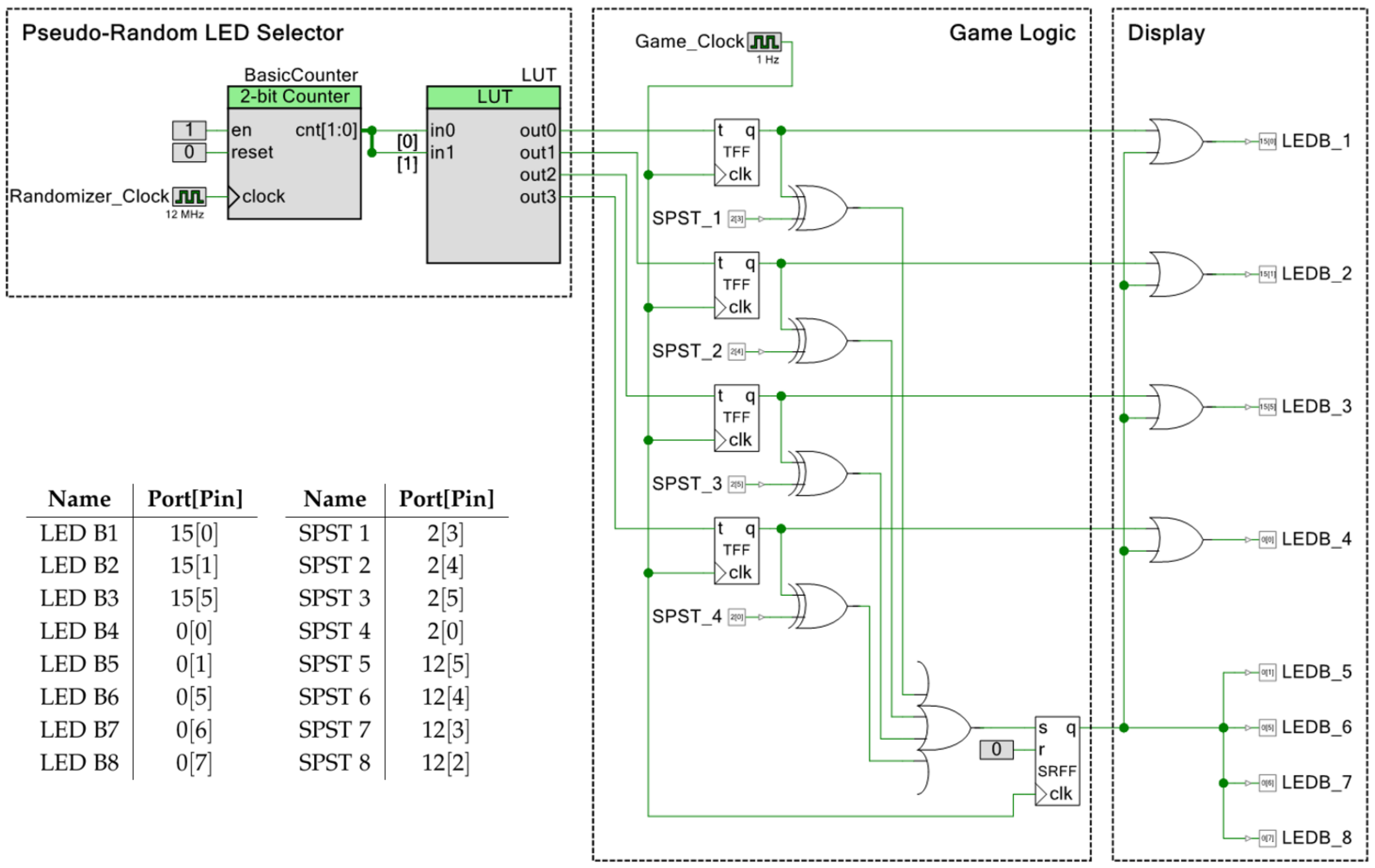
Massachusetts Institute of Technology
Electromechanical Systems Group

Electronics First: PSoC PRACTICE

PSoC Creator Schematic and **Pins:**



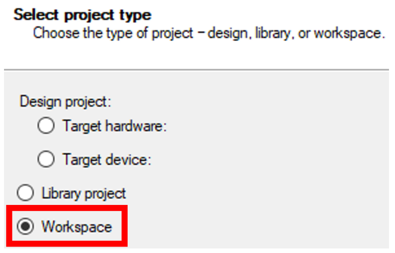
Name	Port[Pin]
LED B1	15[0]
LED B2	15[1]
LED B3	15[5]
LED B4	0[0]
LED B5	0[1]
LED B6	0[5]
LED B7	0[6]
LED B8	0[7]

Name	Port[Pin]
SPST 1	2[3]
SPST 2	2[4]
SPST 3	2[5]
SPST 4	2[0]
SPST 5	12[5]
SPST 6	12[4]
SPST 7	12[3]
SPST 8	12[2]

The Build:

The PSoC microcontroller is a playground for building all sorts of circuits. A single PSoC chip gives us access to a suite of components without needing an extensive collection of integrated circuits or a complex PCB. In the previous lab, we only scratched the surface of the PSoC's capabilities by connecting input and output pins together in Creator. In this lab, you will learn how to use and configure some of the PSoC's more complicated (and very useful) programmable hardware components, such as logic gates, clocks, counters, and flip flops. You will use these components to recreate the egg timer from the COUNTERS lab as well as to build the Switch Game, a "whack-a-mole" clone where a player has to turn on/off DIP switches to match LEDs that the PSoC activates randomly.

THIS LAB CONTAINS A FEW DIFFERENT EXERCISES TO GIVE YOU PRACTICE WORKING WITH PSoC COMPONENTS. You will organize your work inside a single Creator *Workspace* that contains several *Projects*, one for each exercise. Before proceeding to the exercises, open Creator and go to *File* → *New* → *Project...* as you did in the previous lab. When prompted to select the project type, choose "Workpsace." Call it "PSoC Practice," save it to your desired location, and press "Finish" to open Creator into the new (empty) workspace.

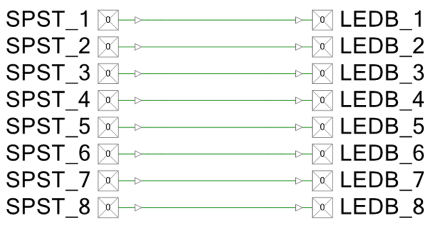

Select project type
Choose the type of project – design, library, or workspace.

Design project:
Target hardware:
Target device:
Library project
[✓ Workspace]

Creating a new workspace.

Exercise 1: Logic Gates

Before you start this exercise, create a new project in the workspace you just made by going to *File* → *Add* → *New Project...*. Call it "Logic Gates". Start by recreating the *SPST* input pins and *LEDB* output pins from the previous lab; see the table on the front page for the pin assignments to use in the `.cydwr` file. **Remember** to configure your *Digital Input Pin* drive modes to **"Resistive pull down".**



Where you left off in the last lab...

Theory of Operation

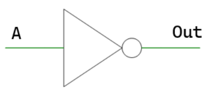
Digital circuits operate at two states, LOW and HIGH. Logic gates, a fundamental component in digital circuits, allow us to compare and perform functions on digital signals. They take an input/set of inputs and produce an output based on them. The rules for the output depend on the type of logic gate used.

We'll explore four of the most basic logic gates first: NOT, AND, OR, and XOR. The function of each gate generally correlates with its name:

- **NOT**: HIGH if the input is LOW, LOW otherwise.
- **AND**: HIGH if both inputs are HIGH, LOW otherwise.
- **OR**: HIGH if at least one input is HIGH, LOW otherwise.
- **XOR**: HIGH if exactly one input is HIGH, LOW otherwise.

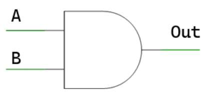
You can see images of each gate below. Each gate has a truth table, which lists the gate's output for each possible combination of inputs (0 being LOW, 1 being HIGH). Based on the descriptions of the gates above, fill in what you would expect the outputs to be.

NOT Gate



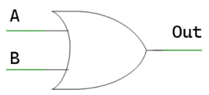
A	Output
0	
1	

AND Gate



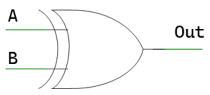
A	B	Output
0	0	
1	0	
0	1	
1	1	

OR Gate

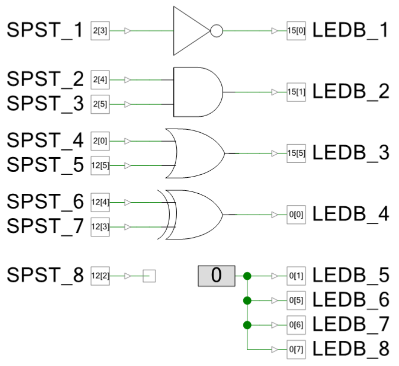


A	B	Output
0	0	
1	0	
0	1	
1	1	

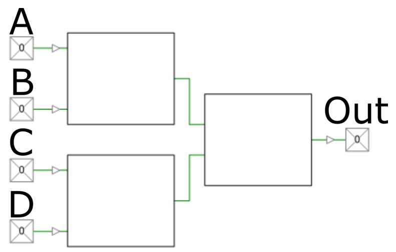
XOR Gate

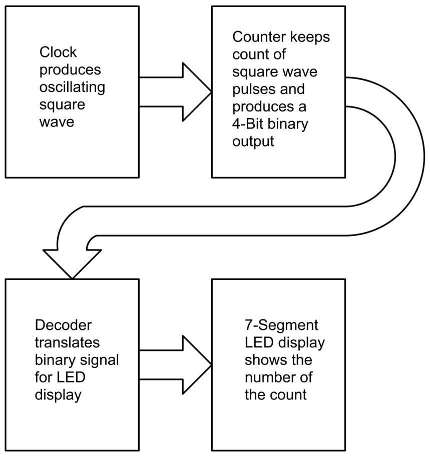


A	B	Output
0	0	
1	0	
0	1	
1	1	



Adding in logic gate components. Unused outputs are connected to *Logic Low*.




Clock produces oscillating square wave
Counter keeps count of square wave pulses and produces a 4-Bit binary output
Decoder translates binary signal for LED display
7-Segment LED display shows the number of the count

The egg timer operation from the COUNTERS lab.

The *Clock* component.

Assembly

We can use the PSoC's logic gate components to test your predictions about how they function. In the last lab, you connected the PSoC to the DIP switches and LEDs on your board. Instead of connecting the switches directly to the LEDs, connect the switches to the inputs of the logic gates, and the gate outputs to the LEDs. You can find the gate components under *Digital* → *Logic* in the Component Catalog. Connect any unused output pins to *Logic Low* '0' components to turn off their corresponding LEDs and prevent Creator errors.

Using the DIP switches, test the various combinations of inputs from the truth tables. Do your observations match your predictions?

Logic gates can be combined to produce more interesting behavior. Consider the following situation: we have 4 inputs (`A`, `B`, `C`, and `D`), and one output. We only want the output to be 1 when just one of `A` and `B` are 1 and at least one of `C` and `D` are 1. Fill in the diagram on the left with the gates that model this behavior, then test out your prediction on the PSoC.

Exercise 2: PSoC Egg Timer

The PSoC, in addition to simple logic gates, offers much more powerful components that we can leverage to build equally powerful circuits. You will use some of these components to rebuild the egg timer from the COUNTERS lab entirely with the PSoC.

The egg timer has three main parts: an oscillator, a counter, and a display decoder. The oscillator generates a square wave; the counter keeps track of how many oscillations have occurred so far and outputs a count of them in binary; and the display decoder takes the binary signals from the counter and translates it into signals that can drive a 7-segment display to show the numbers 0-9. The PSoC has its own components that allow us to implement these three functions. In this exercise, you will use these components to build a timer that counts 0-7 on the LED bar display.

Before you start this exercise, create a new project in the same Creator workspace by going to *File* → *Add* → *New Project...*. Call it "Egg Timer"; put all your work for the next exercise into this project.

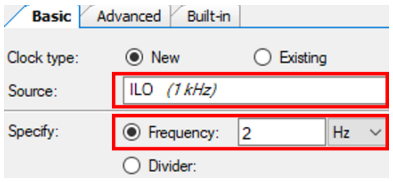
Clock (Component Catalog: *System* → *Clock*)

For the counter's oscillator, we'll use the PSoC's *Clock* component, which generates a square wave at a desired frequency. To change the frequency, go to the clock's configuration menu and change "Frequency" to the desired value.

Test it out yourself: drag in a clock component, set it's frequency to some value of your choosing (try 2Hz to start with). You will also need to change the "Source" for the clock to "ILO", which stands for Internal Low Speed Oscillator. The frequency of *Clock* components in the PSoC is generated by starting with a higher frequency source clock and dividing it down to the desired frequency.

The PSoC has several source clocks to choose from, but since 2Hz is a low frequency, we want to select the ILO as its source, since it is designed for lower frequency oscillation.

With the clock set up, connect it to one of your LED outputs and program the PSoC. You should see the LED begin to flash on and off. Make sure your observations line up with the frequency of your clock!

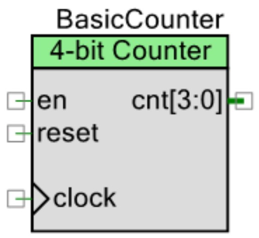


Clock type: [✓ New] Existing
Source: ILO (1 kHz)
Specify: [✓ Frequency:] 2 Hz
Divider:

Changing a clock's source and frequency.

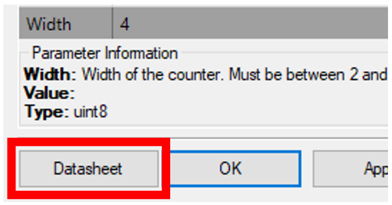
Basic Counter (Component Catalog : *Digital* → *Utility* → *Basic Counter*)

In the Counter lab egg timer, you used a counter IC to count the number of pulses generated by the oscillator. The PSoC has a nearly identical component, the *Basic Counter*. Drag one of these components into your design. The counter has what appears to be 4 "pins", but it's unclear what they do just by looking at the component. This is where the component's *datasheet* will come in handy.



The *Basic Counter* component.

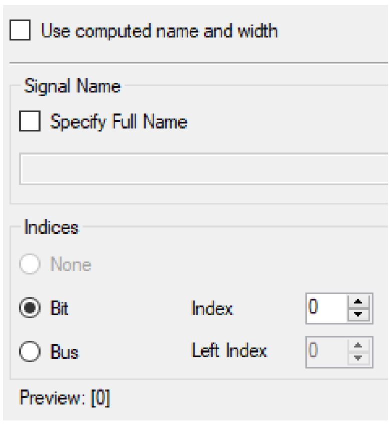
Go to the configuration menu of the component and click the "Datasheet" button. You should see a Cypress datasheet for the *Basic Counter* component appear in your PDF viewer. Every component has a datasheet, and it's a good idea to familiarize yourself with it before using a component for the first time. This will be especially important once components start requiring software to operate. For now, skip to the "Input/Output Connections" section. You should find definitions for each of the inputs/outputs visible on the counter component. Familiarize yourself with them, and ensure each pin's function makes sense.


Width 4
Parameter Information
Width: Width of the counter. Must be between 2 and
Value:
Type: uint8

Accessing a component's datasheet.

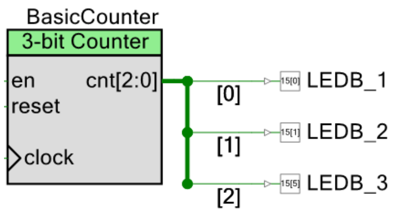
Now you can start using your *Basic Counter* component. First, we need to change the size of the counter. A 4-bit binary counter can count 0-15, but we only want to count 0-7, requiring only 3 bits. Double click the counter and change the "Width" parameter to 3. Now the component should show 3-*bit Counter* at the top.

With the *Basic Counter* size fixed, connect your *Clock* from before to the counter's `clock` input. Connect two of the DIP switch input pins to the `en` and `reset` inputs on the counter. Now the outputs: the 3-bit counter's outputs are grouped together into a single output, labeled `cnt[2:0]`. This single output contains all three counter bits, represented as a **bus** (a grouping of related wires). Each of `cnt 0`, `1`, and `2` represent one output bit of the counter. We can access each individual wire by breaking them out in the schematic.


☐ Use computed name and width
Signal Name
☐ Specify Full Name
Indices
None
[✓ Bit] Index 0
Bus Left Index 0
Preview: [0]

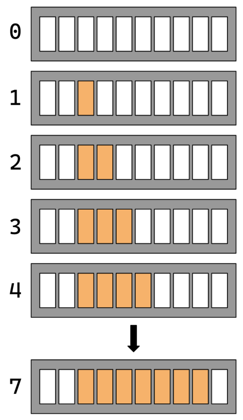
Editing a wire's properties.

Start by drawing some extra wire outward from the bus output (this wire will appear thicker than a normal wire). Draw three more wires out from this wire, until you have something resesmbling a dinner fork. Starting from the top wire segment, double-click each wire segment to pull up a configuration menu for that wire segment. For each of the three wires, uncheck "Use computed name and width", uncheck "Specify Full Name", and then under "Indices", select "Bit", and then select the desired index (a number between 0 and 2). The index corresponds to the wire in the `cnt` bus you want to select.



The end result of "breaking out" the wires from a bus.

Now you can connect each individual wire from the bus to an LED output in your schematic! Use 3 *Digital Output Pins* next to each other, and remember to connect them to the proper pins in the `.cydwr` file (see the table on the front page for pin assignments). After programming the PSoC, use the counter datasheet and your DIP switches to control the counter. You should be able to see a 3-bit binary count happening in your LED bank.

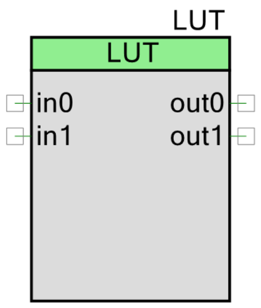


The desired display output. Note that the lower two LEDs on the board are not connected.

Lookup Table (Component Catalog : *Digital* → *Logic* → *Lookup Table*)

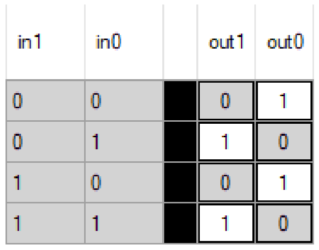
The final piece of the egg timer was the display decoder, which translates the counter binary output to a seven-segment display. On the PSoC board, we have a multi-segment bar display instead of a numeric display. A sensible way to count using this is to fill up the bar as the count advances: at 0, the bar is empty, while at 7, the bar has 7 segments lit up. We'll have to whip up our own decoder to display the count this way.

Luckily, the PSoC has a handy component for situations like this called a *Lookup Table*, or *LUT*. A lookup table is capable of outputting a desired value based on the set of inputs it receives, something like a customizable logic gate. This is essentially the function of a decoder: it reads a set of inputs (a binary count in our case) and "looks up" what value it should output. The LUT component allows us to configure what output values we can get.



The *LUT* component.

Drag a *Lookup Table* into your design. You will notice it starts off with two inputs and two outputs. In the configuration menu, change the number of inputs/outputs to suit our counter. Remember that our counter outputs three bits total, and our LED bar display has eight total LEDs to use. Connect the *Basic Counter* outputs to the *LUT* inputs, and the *LUT* outputs to the LED output pins.



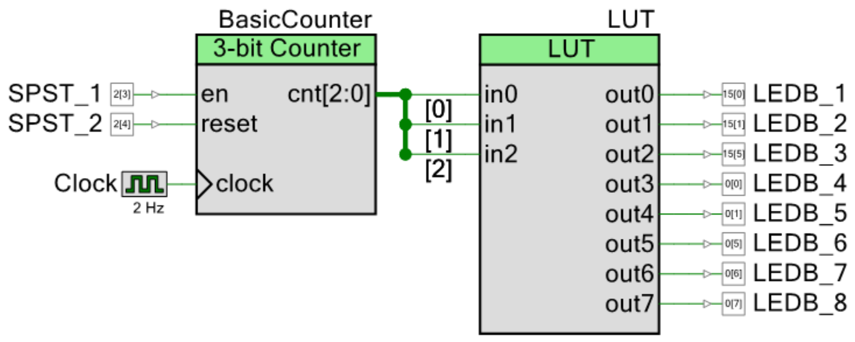
in1	in0		out1	out0
0	0		0	1
0	1		1	0
1	0		0	1
1	1		1	0

An example LUT configuration where each unique combination of inputs sets a single a single output to 1.

The configuration menu allows you to change the output signals for a given set of inputs. If, for instance, the inputs are all 0, or logic LOW (the first row in the table), and we set `out0` to one, or logic HIGH, then an LED connected to `out0` will turn on when all inputs to the LUT are LOW. We can similarly configure every combination of inputs to output a value of our choosing. Use the *LUT* to create a sensible way to display a binary count on the LEDs (something like the output shown at the beginning of this section). You may find that one of the LEDs goes unused. Once you think you've got a good pattern, program the PSoC with your design and test it out!

Conclusion

On the next page is the completed build, which mimics the egg timer hardware! This should start to give you an idea of the PSoC's power. This is where we'll stop with the egg timer for now; there are, however, a few improvements you can make to the project if you are interested in experimenting. These are listed on the next page.

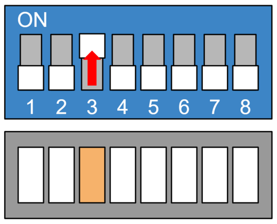


- The LED bar has one more LED you can use. Make the egg timer cycle from 0 to 8 and then wrap around. To do this, you'll need to use increase the counter size by an extra bit, and then use the counter's output to trigger the counter's `reset` pin.
- The previous extension removes the manual reset switch from the counter, since you need the `reset` pin to wrap around. Try re-adding the DIP switch to enable a manual reset while also automatically wrapping around at 8. You might find an OR gate useful to trigger the reset from both a switch and the wrap-around signal.

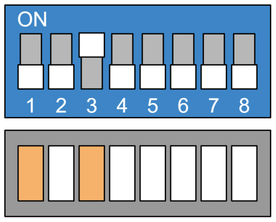
In the next exercise, we'll repurpose the egg timer to help power a much more interesting game.

Exercise 3: The Switch Game

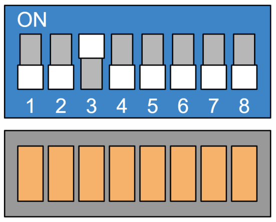
In this exercise, you will build the Switch Game. The game is simple: every second, a random LED on the PSoC board's bar display toggles on/off. When that happens, the player must flip the corresponding DIP switch to match. If the next LED toggles before the player has matched the switches, the game ends and all LEDs turn on. The goal is to survive for as long as possible. See the figures to the right for an example game sequence. After the game ends, you can press the `RST` button on the PSoC stick to play again.



LED 3 lights up, so the player flips Switch 3.



Now LED 1 lights up, but the player fails to react.



Since the player didn't flip on Switch 1, the game ends.

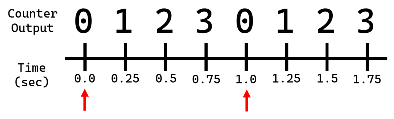
You will be building the Switch Game entirely using PSoC hardware components like the ones from previous exercises! This may seem like a complicated task, but you will divide up the task into four components: randomly selecting/toggling LEDs, remembering the current LED values, detecting mismatches between LEDs and switches, and ending the game. We will only build the Switch Game with four LEDs to keep your schematic from getting too cluttered, but it can easily be extended to use all eight.

Before you start this exercise, create a new project in your Creator workspace called "Switch Game". Put all your work for the next exercise into this project.

Theory of Operation - "Random" LED Selector

First, we need to select an LED to toggle on/off at a regular interval. The selection must be close to random (i.e. unable to be consistently predicted by the player). There are several possible ways of doing this with the PSoC, but this build uses a creative application of the egg timer from the previous exercise to do it.

We can configure our counter to be only two bits wide, so that it repeatedly counts 0-3. We can then set the counter's clock to a very high frequency so that it cycles through the count fast. Finally, once per second, we "sample" the counter's current output (we store the output of the counter at the exact moment we choose to observe it). Given the counter's high frequency, it seems virtually impossible to predict which of the four values we'll observe when we sample. If we assign an LED to each of the counter's four possible outputs, the counter acts as a mechanism for randomly selecting LEDs.

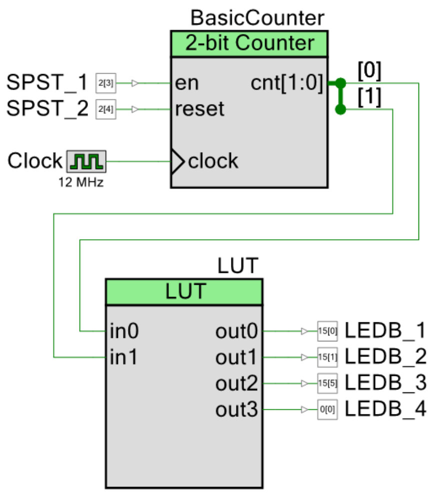


If the counter increments every $\frac{1}{4}$ second, and we sample once per second (the red arrows), we observe the same value each time!

However, is it actually impossible to predict which values we'll observe? Consider the following scenario: you set the counter clock's frequency to be 4 Hz and you observe the counter's output every second. Can you predict what sequence you would observe? Now think about the case with a counter clock frequency of 4 MHz (4,000,000 Hz), a relatively high frequency. Does the output sequence change? What sequences would you observe with a counter clock frequency of 6Hz or 6MHz?

It turns out that any pairing of counter/observation frequencies will cause issues like this. Each time the game is played, the same LED/sequence of LEDs will be picked over and over again. However, we benefit from an accidental property that clocks on the PSoC (and clocks in general) are not perfect, and will sometimes oscillate at *ever so slightly* the wrong times. When we build the randomizer sampling circuit, we'll use a separate clock to trigger the sampling; the inaccuracy of this clock is what creates this circuit's "randomness". It is helpful to know such properties of components you use, mostly so you can watch out for them, but also so that you can take advantage of them if needed.

Assembly:



Begin by copying the egg timer from the previous exercise's project to this new project. To copy over the egg timer, select all the components from the project, right click them and press "Copy", then right click your new project's schematic and press "Paste". The *Basic Counter* we used has three bits of output, meaning it can count values 0-7. Since we only need four possible values, we can change this to a 2-bit counter. Double click the counter and change "Width" to 2. Now the output of the counter should show `cnt[1:0]`, signifying only 2 bits.

Now we need to change our lookup table to match the new timer and display format. Update the *LUT* to only use 2 inputs and 4 outputs, one for each of the selectable LEDs. For each possible counter output, we only want 1 LED to be activated; choose a sensible *LUT* configuration that fulfills this requirement. At this point, your schematic should look like the one to the left.

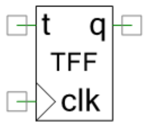
Test this setup by connecting the *LUT* to 4 LEDs and connecting the counter's *en* pin to a DIP switch. Set the clock component to a high frequency such as 12 MHz. Program the PSoC, and then use the switch to pause the randomizer and "sample" its output. At such a high frequency, you should be unable to stop the counter at a predictable value.

Theory of Operation - Remembering Values

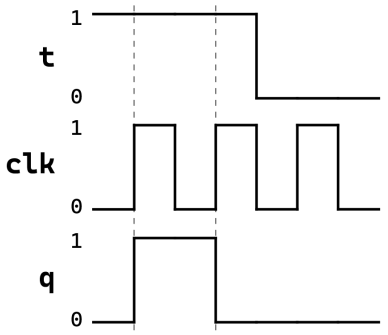
We have a way to generate somewhat random values; now we need a mechanism to sample the randomizer at a fixed interval. This presents a new problem: we want to remember the previously sampled value from the randomizer while also running the randomizer to generate the next value. How can we use a circuit to "remember" a value?

The answer lies in a new type of logic component: the flip flop. There are many different kinds of flip flops, but the general principle is the same: they take a set of inputs and generate a single output *that persists* even after the inputs have changed. This is different from the logic gates we began with in this lab, which have outputs that are completely dependent on the state of their inputs at any given time. A flip flop is thus useful for storing values!

To get a feel for how a flip flop works, we will start with a toggle (or T) flip flop (*Digital → Logic → Toggle Flip Flop*). It has two inputs (`clk` and `t`) and an output `q`. The output begins at 0. To change the output, the `t` input must be 1, and then the `clk` input must experience a "rising edge", or a transition from 0 to 1. This cues the flip flop to toggle it's output: if it was 0, it is now 1, and vice versa. The key observation is that the flip flop samples `t` and updates `q` only on a `clk` rising edge; at any other time, it simply stores and outputs the current value of `q`. For our game, we need to store the value of an LED and toggle it at a regular frequency, which means a T flip flop is exactly what we need!



The *T Flip Flop* component.

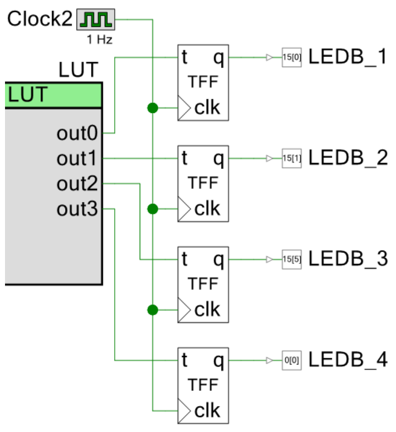


The T flip flop's clock diagram. Note how `q` only changes on `clk`'s rising edges (marked by the dotted lines).

Assembly

Disconnect the LEDs from the outputs of your *LUT*. For each output of the *LUT*, drag in a single *T Flip Flop*. Connect each output of the *LUT* to each `t` input of the flip flops. Connect your LEDs to the `q` outputs.

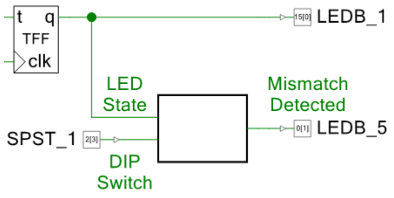
Now, grab a new *Clock* component and set it's frequency to 1 Hz. Because this clock operates at a low frequency, change its "Source" to be the "ILO" like you did in the last exercise. Recall that the inaccuracy of this clock causes the pseudo-randomness of this system. Connect this clock to each of the `clk` inputs on the *T Flip Flops*. At this point, your schematic should look something like the figure on the right.



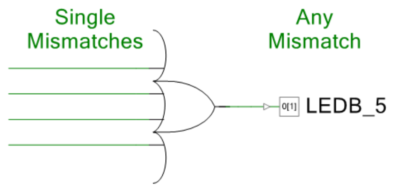
Before you program the PSoC, try to figure out what is going on in this setup. Each second, the clock will transition from 0 to 1, creating a rising edge. This signals the *T Flip Flops* should sample their `t` input and alter their `q` output. Given what you know about the *LUT*'s outputs, what will happen every second to the LEDs?

Now program the PSoC. You should see that every second, a random LED is chosen, and if it was off, it turns on, and vice versa. We are now randomly selecting and toggling LEDs!

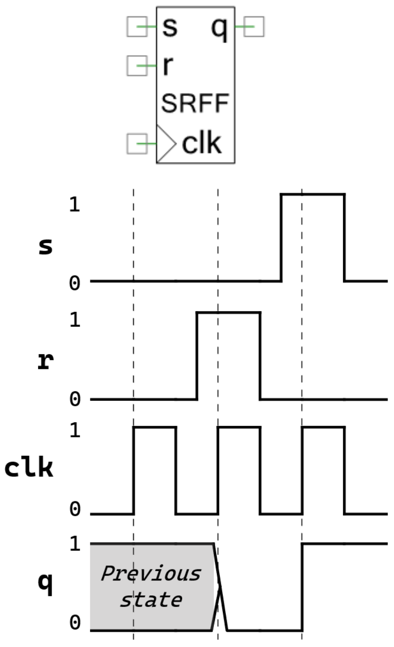
LED	Switch	Mismatch?
0	0	
1	0	
0	1	
1	1	



A single mismatch detector.



The global mismatch detector.



The *SR Flip Flop* and its clock diagram.

Theory of Operation - Mismatch Detector

Next, we need to compare the DIP switches with the current state of the LEDs and detect whether or not there's a mismatch. This sounds like a job for a logic gate: we have two inputs (an LED's state and the DIP switch) and an output (1 if they don't match, 0 otherwise). Use the truth table on the left to help determine what logic gate fits this purpose.

Once we can detect a mismatch at each individual LED, we will want to use them to create a single signal that is 1 when any of the mismatch detectors are on, and 0 otherwise. When this signal is 1, we know we should end the game. We can use an OR gate with 4 inputs to create this signal: the output of the OR gate will be 1 if any of its inputs are 1.

Assembly

Repeat the following steps for each LED: using the mismatch detecting gate you selected, connect one of its inputs to the *T Flip Flop* output (the LED's current state) and the other to its corresponding DIP switch. Connect the output of the mismatch detector to one of the four unused LEDs on the LED array; this way you can see the mismatch detectors at work while the game is played. Program the PSoC; you should see that for each LED whose state doesn't match its DIP switch, the corresponding mismatch detector turns on!

Once you're satisfied with the individual detectors, connect each one to an OR gate to create a single global detector. To change the number of inputs on the OR gate, double-click the component and change "NumTerminals" to 4. Disconnect the individual detectors from their LEDs and instead connect the OR gate's output to one of the free LEDs. After programming the PSoC, global detector should light up whenever there's a mismatch at any of the LEDs.

Theory of Operation - Ending the Game

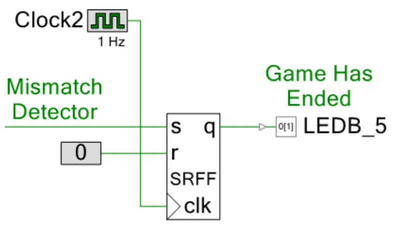
Two problems remain. First, mismatches need to end the game; currently a player is allowed to continue the game no matter how many mismatches have happened. Second, mismatches are detected the instant they occur; if we ended the game as soon as the detector fired, the player wouldn't stand a chance. We need to give the player some reaction time.

We'll use a single component to solve both of these problems: the set-reset (or SR) flip flop (*Digital* → *Logic* → *SR Flip Flop*). Similar to the T flip flop, the SR flip flop has a `clk` input and an output `q`. When `clk` sees a rising edge, the flip flop reads its `s` and `r` inputs: if `r` (or "reset") is 1, the output is reset to 0, while if `s` (or "set") is 1, the output is set to 1. If `s` and `r` are both 0, the output retains its previous state. In the PSoC, the output is 0 when both `s` and `r` are 1, but this is not true of all SR flip flops.

The SR flip flop is perfect for keeping track of whether our game has ended. If we always keep the `r` pin at 0 and feed our mismatch detector into the `s` pin, an uncaught mismatch will set the output to 1, and the output will be unable to change back to 0 from then on. The fact that the output only changes on a rising clock edge also gives the player a single clock cycle to react, since when a mismatch first occurs, the SR flip flop won't register it until the next clock cycle after. If the player fixes the mismatch in time, the `s` input will be 0 on the rising edge, and the game can continue!

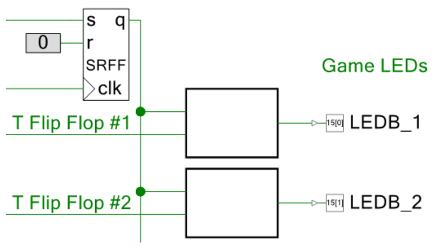
Assembly

Drag in a single SR Flip Flop and connect its `clk` pin to the same clock used on the *T Flip Flops*. Connect the output of the global mismatch detector to the `s` input, and connect a *Logic Low* component to the `r` input to prevent the flip flop from ever resetting. Finally, connect the `q` output to an LED you haven't used yet. Program the PSoC; you should have a single clock cycle to flip each DIP switch, and if you don't switch it in time, the test LED should turn on. Even if you correct all mismatches, the LED will remain on.



Creating the game over signal with an *SR Flip Flop*.

Now we need to use the *SR Flip Flop* output to turn on **all** the LEDs when the game is over. We can use logic gates to accomplish this. Each LED involved in the game should be on if the corresponding *T Flip Flop* is on, or if the game over signal is on. For each LED, connect its *T Flip Flop* output and the game over signal to the inputs of a logic gate that models this behavior. Connect the output of this gate to the LED.



Using the game over signal to toggle the LEDs on.

Program the PSoC. If you let a mismatch occur, the LEDs should all light up. If you fix the mismatch, the LEDs will remain lit up. Congratulations, you now have a working Switch Game!

Wrapup

Now that you've built the egg timer and the Switch Game in the PSoC, you should have a lot of experience working with the PSoC's programmable hardware components. We've still used very little of what the PSoC is capable of; for now, however, you have enough tools to start experimenting on your own. You can start by attempting some extensions of the Switch Game:

- Increase the amount of reaction time the player has by allowing them to make three consecutive errors without the game ending. This will involve increasing the delay between a detected mismatch and the game ending. Look into using the *D Flip Flop* component to accomplish this.
- Add a fifth selectable state in the randomizer that inverts all the LEDs at once. Then use the push-button on the PSoC to invert the function of all the DIP switches at once in response. The push-button on the PSoC is connected to Pin 2.2; note that the button connects the pin to *ground*, so you'll need to set the pin's drive mode to "Resistive pull up".
- Count the number of cycles a player survives. Then, when the game ends, display the count in binary on the LEDs. You'll need some way to choose between displaying the LED state and the cycle count; look into using the *Multiplexer* component for this.

Some of these features may involve components that we haven't talked about yet. If you need to use something new, you can always look at the datasheet of any Creator component to understand how it functions.