# EXERCISE 4
# STATE MACHINE FOR AN LED DRIVER

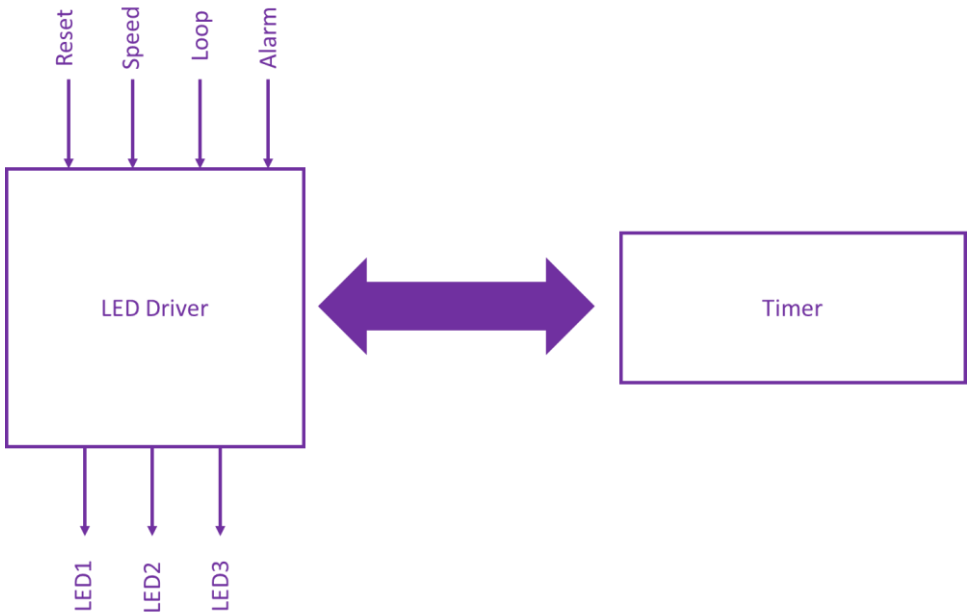
Consider an LED indicator driver with three outputs, one for each of the LEDs. Each state is driven by a digital signal that ranges from 0 to 255, where 0 is OFF and 255 is the maximum brightness. The table below shows the state equivalent.

| LED Name | LED 1 | LED 2 | LED 3 |
|---|---|---|---|
| **State 1** | 255 | 0 | 0 |
| **State 2** | 0 | 255 | 0 |
| **State 3** | 0 | 0 | 255 |
| **State 4 (Standby)** | 255 | 255 | 255 |
| **State 5** | 0 | 0 | 0 |
| **State 6** | 255 | 255 | 0 |
| **State 7** | 128 | 0 | 128 |

We want to design an LED driver that will iterate through the states

**1 -> 6 -> 2 -> 3 -> 7 -> 1**

in two iteration modes: continuous loop or backwards loop with one of three different speeds: every 1, 3 or 5 seconds/state. The default speed is 1 second/state. The default iteration is continuous loop. The initial state is `STANDBY` and it lasts 4 seconds. During standby, the state is `State 4`, after which it turns to `State 1`.



The implementation of this exercise must be done in two different modules. Use separate entities for the LED driver and the timer. This means you need to decide what kind of driver-to-timer interface you want. The interface can be either synchronous or asynchronous, that is, the timer does or does not use the same clock as the driver. Furthermore, you may reset the timer directly or through the driver-to-timer interface.

The driver will have four switches:

- ✓ A reset switch to change between the `STANDBY` state and default settings.
- ✓ Two speed controller switches to iterate through the three different speed modes.
- ✓ An iteration control switch that will change from continuous loop mode (`State 1` after `State 7`) to backwards loop mode (`State 3` after `State 7`, `State 2` after `State 3` and so on).
  **Note** that in both modes, `State 6` always goes after `State 1`. `State 1` is the initial stop so in the backward loop, you would not have `State 1`.

**Task 1**

Your first task is to sketch a state machine diagram that covers all the legal state transitions of the machine. **Keep it as simple as possible,** while showing all the main states and transitions.

**Task 2**

Model the LED driver using separate entities for the controller and the timer. Decide what kind of interface you want to use and implement it. The timer should be aware of the clock speed (predefined) and should send a signal to the LED driver every 1/10 of a second. In the LED Driver, there is a counter that decides the state machine transitions together with the switches.

Create a *testbench* that ensures all the possible transactions of the LED driver. Simulate the design.

**Task 3**

Now we will modify the current design to use a generic timer. Having a counter on the LED Driver partly defeats the purpose of having a different timer, because our timer is, so far, just a frequency divider for the clock. Moreover, the state machine is event-based if we put all the time-related functionality in the timer, and therefore we can have much more efficient hardware implementation for the LED driver,

In this task, write a generic timer module and modify the interface. The input to the timer will now be the number of milliseconds (or 1/10th or 1/100th of a second, as you prefer), that the timer should count before rising a flag. When a flag from the timer is raised, it is time to change state in the LED Driver (if something else has not happened in the meantime with the switches!).

Implement it on Zedboard

**Task 4**

Now we add an additional input

- ✓ An alarm switch that will trigger a 7 second alarm state (starting after the switch has been deactivated). During this state, the LED will blink at a frequency 10 times faster (with respect to the frequency it had when the alarm was triggered). The LED State during the alarm is `State 4.`

The duration of the alarm state is 7 seconds, but now there are substates that also require timed transitions. You could have a counter for those in the LED Driver to count the 7 seconds based on the shorter ones… but that would again partly defeat the purpose of having a timer. Instead of that, use two instances of the same timer entity, using the new one for the alarm substates.

**Task 5**

Implement your code to the Zedboard.

**Task 6 (Grade 4 and 5)**

For this task, you should turn an LED on/off using a button as input. You need to write a debouncing module (with an interfacing purpose similar to the PWM modules you designed before), because the button signal is noisy, and you would otherwise get multiple rise and fall events for a single button push.

**Task 7 (Grade 5)**

Finally, modify your code and use buttons instead of switches for all your inputs:

- ✓ Reset button
- ✓ Alarm button
- ✓ Speed controller button
- ✓ Iteration control button

For the latter two, you will need two simple states machines to store the speed and iteration states.