Embedded Systems

Lab 2.2

Richard Ismer

Dimitri Jacquemont

November 25, 2022

1 Problem Statement

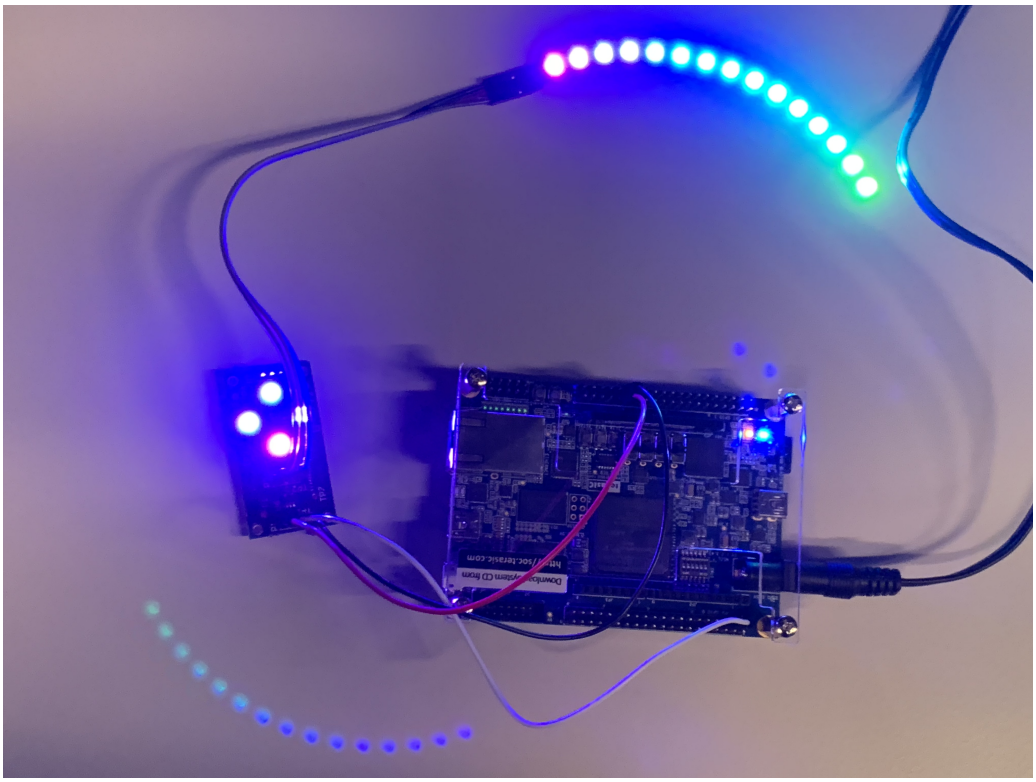

Figure 1: Project's outcome

After using a pre-existing standard PIO IP component in lab 2.0 and creating our own custom IP component in lab 2.1, we chose to design a custom slave IP component supporting WS2812, an intelligent control LED light source. This project consists in creating a VHDL slave component and the firmware operating it. In this project we are using the DEO-Nano-SoC board and WS2812.

2 High-level Description

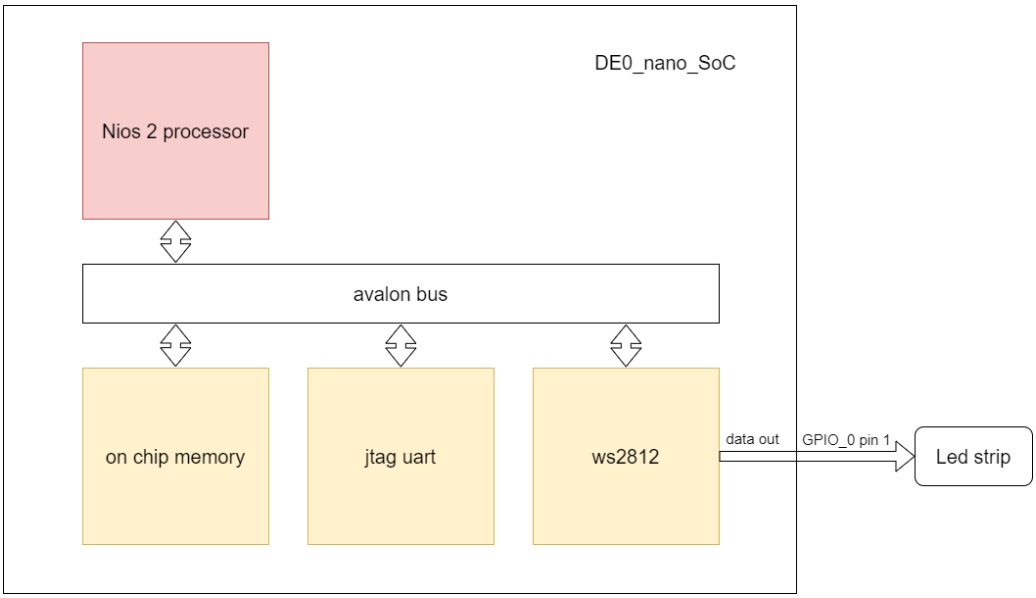

Figure 2: Global view of the peripherals and their interaction

In order to run our custom slave on the DEO-Nano-SoC, we are making use of the Nios 2 processor, a 32-bit embedded processor architecture designed specifically for the Altera family of FPGA integrated circuits. This IP component will be the brain operating our IP slave component.

Our custom slave peripheral is communicating with the processor through the Avalon bus, another IP component. On the adress bus are transmitted the address of our peripheral WS2812 as well as an internal address corresponding to the color (R, G, or B) of a certain LED. On the data bus is transmitted a byte for the intensity of the color of the LED that is to be changed.

Base address ws2812	LED number (6 to 2)	Color selection (1 to 0)

Table 1: WS2812 address format

3 Low-level Description

To be able to display the right colors on the intelligent LED strip the peripheral has to output a signal with precise timings. Each color bit is encoded as a high level followed by a low level with their length depending on this bit and a burst of LED's are separated by a reset time that consist of a 0 on the data output line. The next figure show the different timings to encode a 0 or a 1. Since our clock runs at 50 MHz and the timings for different codes are in nanoseconds no clock divider is needed. We have the following timings:

- For code 0: T0H is 0,35 µs (17 clock cycle) and T0L is 0,8 µs (40 clock cycle)
- For code 1: T1H is 0,7 µs (25 clock cycle) and T1L is 0,6 µs (30 clock cycle)
- For ret code: Treset is above 50 µs (more than 2500 clock cycle)

These clock cycles are only valid if the clock runs at the same rate.

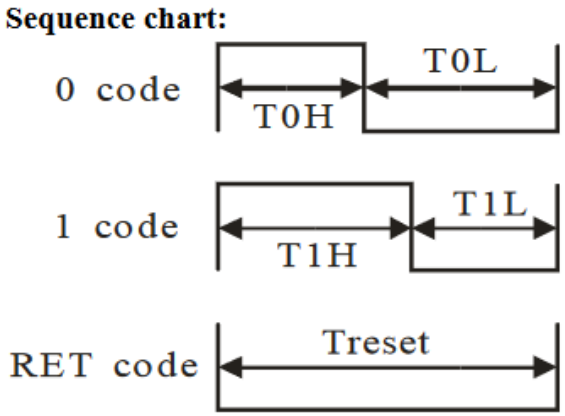

Figure 3: Timings for bit encoding

To store the state of the LED (remember the color of each LED) we have an array of logic vectors. The array has a length of 24 for a maximum of 24 LED. Each vector has a length of 24 bits to store the red, green and blue value. The content on the array can be accessed through the Avalon bus for writing and for reading.

Each time the state is changed in our state machine, a variable **counter** is incremented with every clock cycle. Our state machine is composed of 3 states as displayed in figure 4:

- CODE_0 :
  - If **counter** $> TOH+TOL$, the counter is done, and a new state is to come depending on the value of the next bit to send, or reset if it was the LED's last bit.
  - Else if **counter** $< T0H$, the counter keeps counting while the value 1 is output.
  - Else, the **counter** keeps counting to $TOH + TOL$ while the value 0 is output.
- CODE_1 :
  - If **counter** $> T1H+T1L$, the counter is done, and a new state is to come depending on the value of the next bit to send, or reset if it was the LED's last bit.
  - Else if **counter** $< T1H$, the counter keeps counting while the value 1 is output.
  - Else, the **counter** keeps counting to $T1H + T1L$ while the value 0 is output.
- RESET : the variable **led_counter** and **bit_counter** are reset to zero
  - If **counter** $= Treset$, the counter is done, and a new state is to come depending on the value of the next bit to send, i.e. the first bit of the first LED.

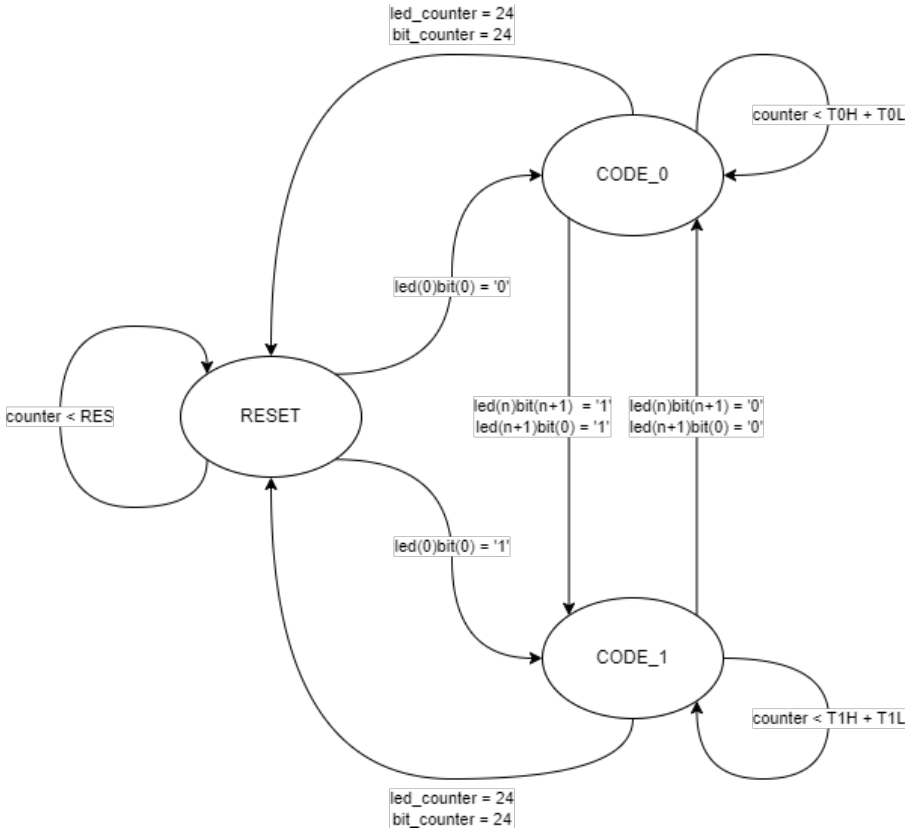

Figure 4: State machine of the WS2812

We can find below the internal structure of our component. Only the array is interacting with the Avalon bus.

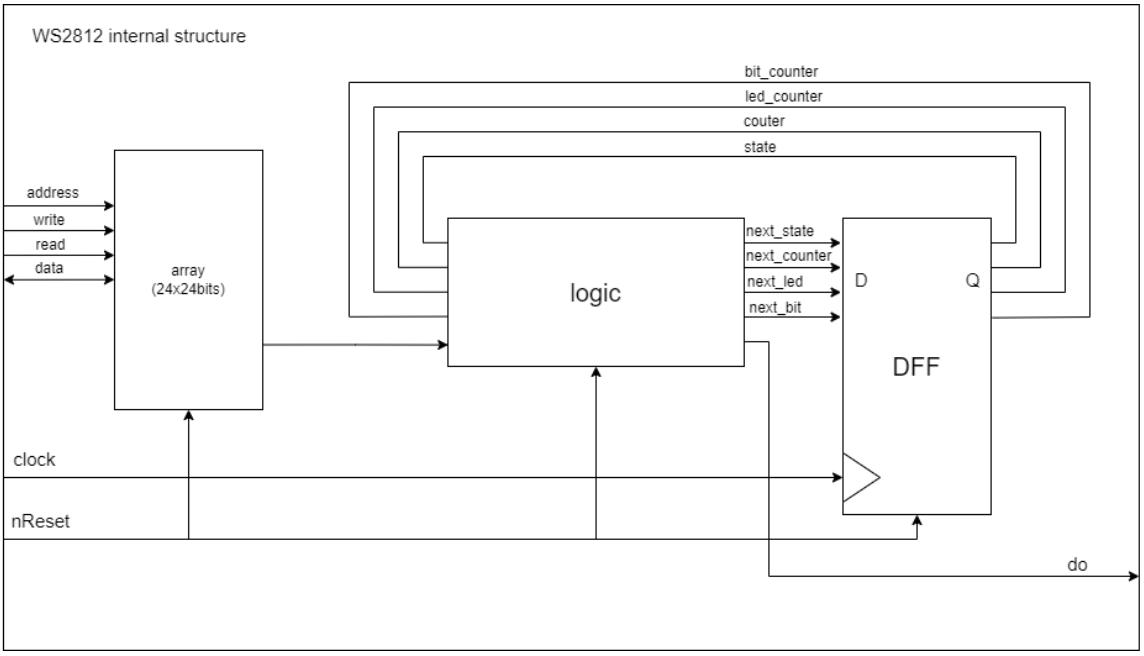

Figure 5: Internal structure of the WS2812

In order to test our design, a testbench was created and then tested on ModelSim. The results obtained validated our design.

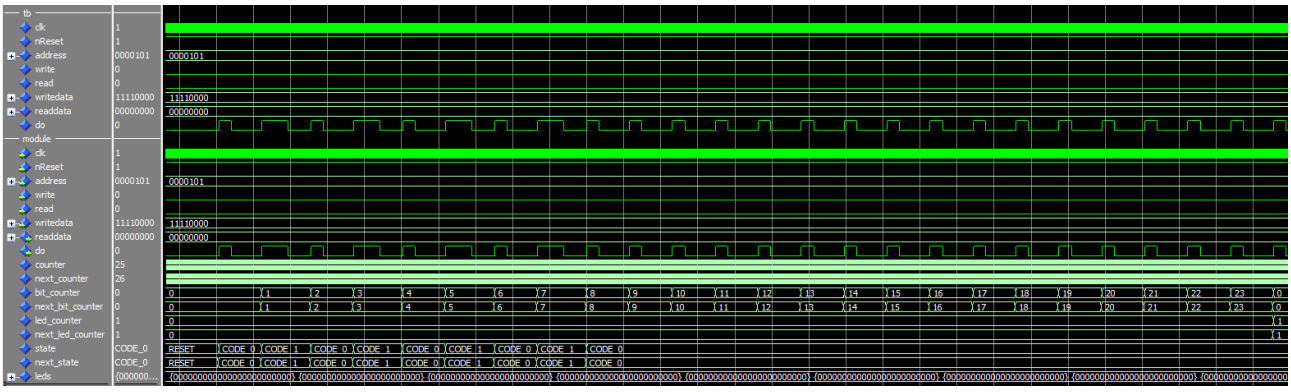

Figure 6: Simulation in Modelsim

4 Firmware

To display the capabilities of our component we decided to have a rainbow circling on the LED. First things first we need to be able to write a color in our component. To do so we have created a function that will write all the colors for a specific LED.

```
void writeRGB(uint r, uint g, uint b, uint led) {
  if(r > MAX_COLOR || g > MAX_COLOR || b > MAX_COLOR || led >= LED_NUM) {
    return;
  }
  IOWR_8DIRECT(WS2812_0_BASE, (led << LED_OFFSET) + BLUE_OFFSET, b);
  IOWR_8DIRECT(WS2812_0_BASE, (led << LED_OFFSET) + RED_OFFSET, r);
  IOWR_8DIRECT(WS2812_0_BASE, (led << LED_OFFSET) + GREEN_OFFSET, g);
}
```

It takes the different color component and the led as input and writes each of them to the ws2812.

We then have created a function that will write colors to all the LED's with a given offset. The color written are stored in a table (**tab**) with all the different colors for the rainbow we want. By changing the offset we can change the position of a certain color on the LED strip.

```
void writeLEDS(uint offset) {
  for(int i = 0; i < LED_NUM; i++) {
    uint* line = tab[(offset + i) % LED_NUM];
    writeRGB(line[0], line[1], line[2], i);
  }
}
```

We then have to change the offset at a constant time interval to have the LED start moving. This is done by the main function.

```
int main()
{
  uint counter = 0;
  while(1) {
    writeLEDS(counter);
    if(++counter == LED_NUM) {
      counter = 0;
    }
    usleep(20000);
  }

  return 0;
}
```

At this point we have a rainbow circling on the LED strip.

5 Conclusion

Our system meets the project's requirement. The component works as expected since we can see the LEDs light up. Designing a slave component was an interesting challenge that was met with success.