Embedded Systems Laboratory 2
Custom Avalon Slave Programmable Interface

Snoeijs, Jan
jan.snoeijs@epfl.ch

Spieler, Michael
michael.spieler@epfl.ch

07 November 2017

1 Choice of peripheral

We have chosen to implement the slave peripheral for the W2812 LED controller. In this report we present an overview of our hardware realization of the programmable module, the full system architecture and our software code.

2 System overview

Figure 1 shows the block diagram of the whole system. It comprises a NIOS-II CPU, SRAM memory, JTAG UART and our custom LED programmable interface, which are all connected over the Avalon bus. The LED programmable interface has, besides the Avalon slave signals, only one output signal which exits the FPGA on `GPIO_0(0)`. A daisy chain of 16 WS2812 LED controllers is connected to `GPIO_0(0)` which will be controlled by the custom programmable interface.

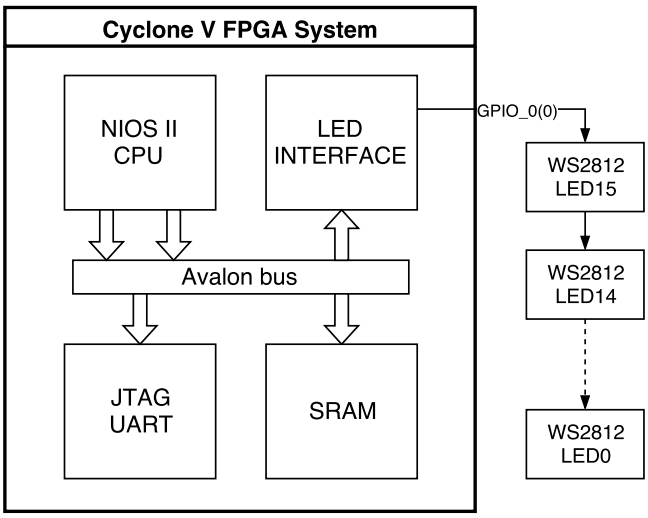

Figure 1: System level block diagram

Figure 2 illustrates the Module-level block diagram of the programmable interface including the Avalon slave signals, `LedData` output signal, the register set and a finite state machine which is documented in section 3.

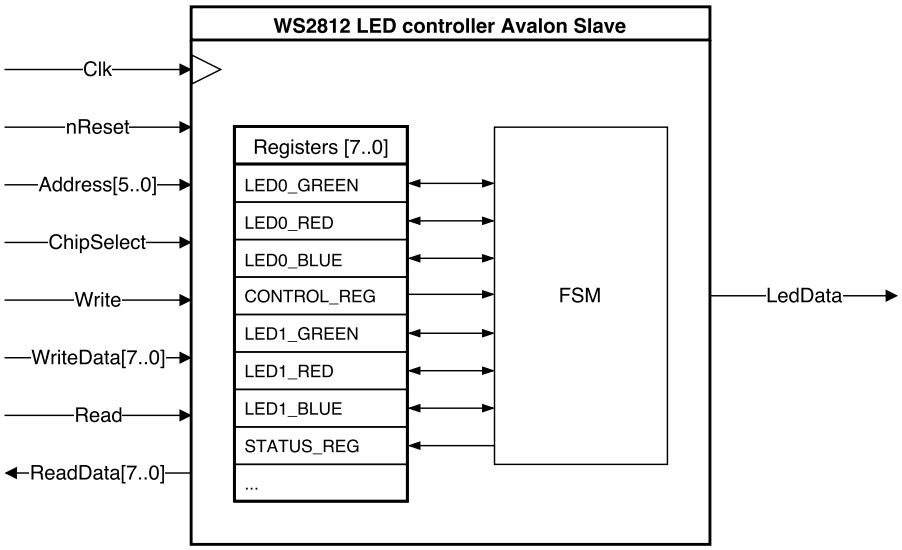

Figure 2: Module-level abstract block diagram

3 Finite state machine - VHDL implementation

In this section we describe the VHDL implementation of the programmable interface for the LED controller. The implementation we chose is the finite state machine shown in Figure 3. In the IDLE state, the machine is waiting and the registers are ready to be written. A signal sent by the processor (software) handles the transition to the LOAD state where the data registers are written. In this state, the Avalon bus is allowed to write in the registers. The transition is again allowed by the software by setting the control register to 2. From then on every write operation in the registers is not taken into account until the system is back in IDLE state. In the WR0 and WR1 states are similar. In WR0, a signal '0' is sent over the LedData signal ('1' for 0.4 μs and '0' for 0.85 μs) by decrementing a counting register ("wr0 cnt"). In parallel of this, a series of counters are decremented to determine how many bits are left to write and what is the value of the next bit. This process determines whether the state machine stays in the same state, switch to the other writing state or return to IDLE.

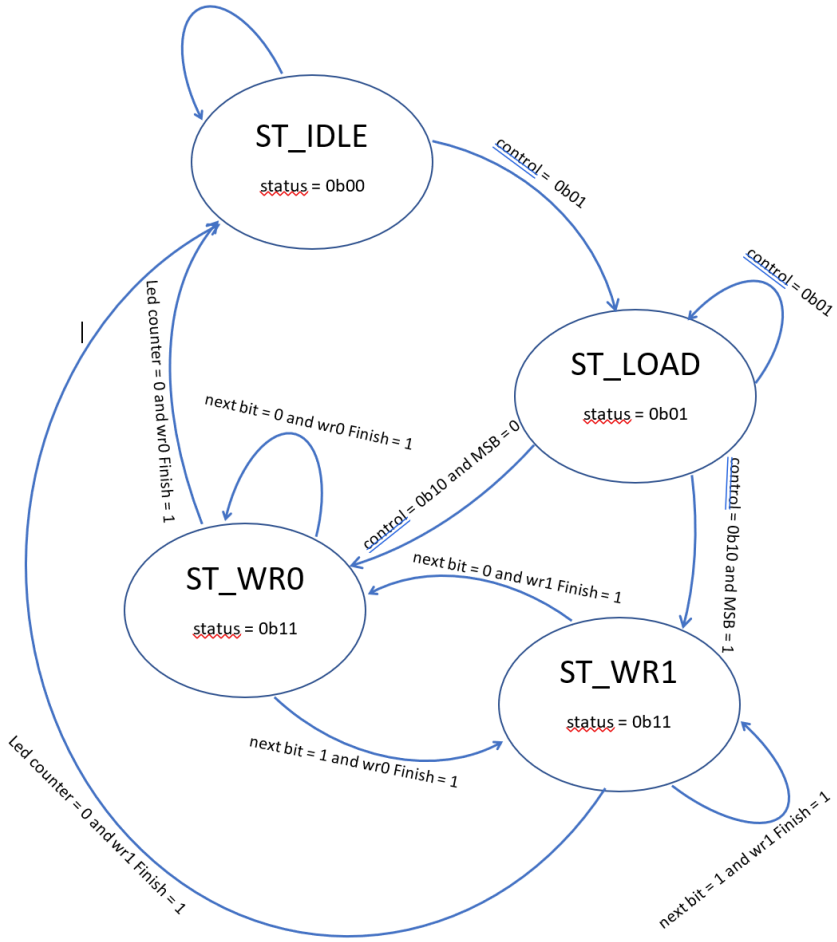

Figure 3: State diagram of slave avalon Led controller FSM

3.1 Detailed description of the VHDL code

Our implementation of the FSM described before in VHDL is divided into several processes. To make sure the design is synchronous we wrote two processes dependent only on clock and reset. One instantiates the state register for the FSM, and the other one instantiates all the other used registers, divided into the software-accessible registers and inaccessible internal registers.

A separate process handles the transition between the states of the FSM.

Then, outside of any process, status signals are asserted in a concurrent way. These signals are of Boolean type and toggle to '1' if their associated counter has reached its final value. The processes sensitivity list are dependent

on these status signals and on the registers, which shouldn't compromise the synchronization of the design because the status signals only depend on registers which behave synchronously.

In an other process we define the signal assertions that need to be done in each state of our FSM. In this process, our different counters (leds, color, bits, wr0) are decremented in a cascade way, meaning that the bit counter is decremented by one only when wr0 is fully decremented and so on. This should ensure that the data is sent in the right order to the LEDs.

Two similar processes (WR0, WR1) define how long a pulse should last to write a '0' or a '1' in the leds. We have done this in accordance with the data sheet specifications taking tolerances into account (Table 1). The reset code of 50 μs between two sets of data is handled in software.

Finally, we wrote a process for the Avalon interfacing, where the 8-bit vector stored in "WriteData" is written into our data register (array of 16x3x8 bits) depending on the "Address" and the data stored in our status register is assigned to "ReadData". All these operations happen in accordance with the Avalon bus requirements; ChipSelect and Write have to be '1' for a write and ChipSelect and Read have to be '1' for a read.

	nominal value [μs]	tolerance [ns]	measured value [μs]
H1	0.7	+/- 150	0.8
L1	0.6	+/- 150	0.46
Total WR1	1.4	+/-600	1.26
H0	0.35	+/- 150	0.45
L0	0.8	+/- 150	0.85
Total WR0	1.15	+/- 600	1.3

Table 1: LED controller protocol — specifications vs measurements

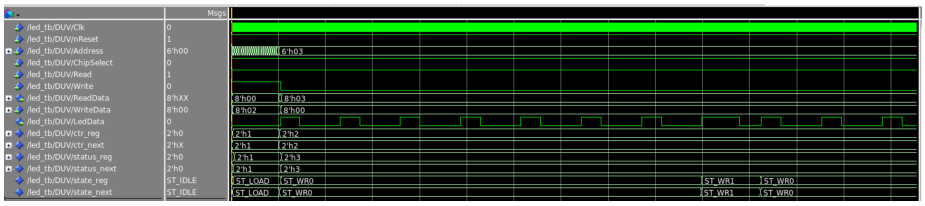

Figure 4: State register behavior — Modelsim

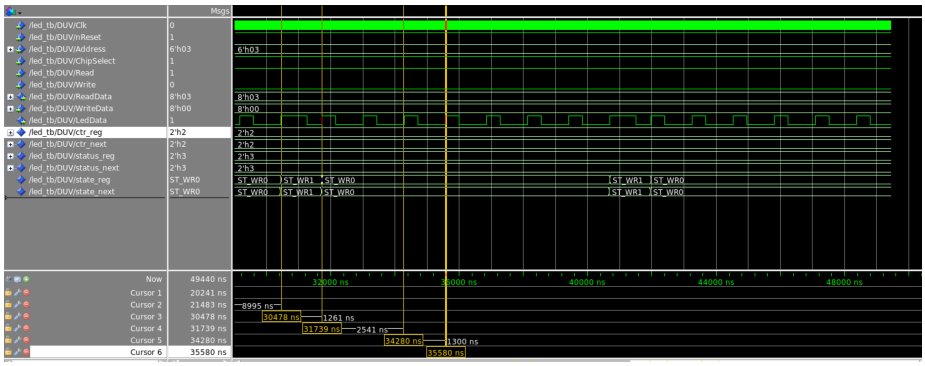

Figure 5: Output signal behavior — Modelsim

Figure 4 shows the behavior of the state register and Figure 5 shows the output signal of the module and shows the timings of a write '1' to the LED and a write '0'.

4 Software

In this section we briefly describe how the programmable interface is accessed from software running on the NIOS-II processor.

4.1 Register map

Table 2 shows the register map of the programmable interface.

offset	register	R/W	description
0b000000	LED0_G	W	8 bit register storing the color intensity
0b000001	LED0_R	W	–
0b000010	LED0_B	W	–
0b000011	CR	W	2 bit control register with 2 defined values: 0b01 start writing, 0b10 finished writing
0b000100	LED1_G	W	–
0b000101	LED1_R	W	–
0b000110	LED1_B	W	–
0b000111	SR	R	2 bit status register with 3 defined values: 0b00 ready, 0b01 waiting for data, 0b11 busy
0b001000	LED2_G	W	–
0b001001	LED2_R	W	–
0b001010	LED2_B	W	–
0b001011	Unused	W	–
...	...	...	–
...	...	...	–
0b111100	LED15_G	W	–
0b111101	LED15_R	W	–
0b111110	LED15_B	W	–
0b111111	Unused	W	–

Table 2: Register map of the LED interface

LED data register The LED data registers are 4 byte aligned where LEDx has the offset 0bxxxx00. Each LED has 3 8bit data registers: LEDx_G with offset 0b00, LEDx_R with offset 0b01 and LEDx_B with offset 0b10. The offsets 0bxxxx11 are unused except for the control register CR which is at offset 0b000011 and the status register which is at offset 0b000111.

Control Register The control register is 2 bit wide. Before writing LED data, the value 0b01 must be written to CR. Then the value 0b01 must be written to initiate the transmission.

Status Register The status register is 2 bit wide and represents the state of the programmable interface: 0b00 ready, 0b01 waiting for data, 0b11 busy. The software can poll the status register to detect when the transmission is complete.

4.2 Source Code

The source listing 1 shows a simple demo program that turns on a green LED wich propagates along the LED strip. The register map is represented through C macros defined at line 12. On line 42 the load command is written to the control register followed by the LED data write. The data write is finished by

writing the start command to the control register on line 56. Then on line 59 the program polls the status register until the transmission was successful. On line 61 we inserted a delay long enough to let the transmitted data be updated to the corresponding LED and to obtain a visible animation.

Listing 1: main.c

```
1  /*
2  Title: Lab2 LED Programmable interface
3  Authors: Jan Snoeijs, Michael Spieler
4  */
5
6  #include <inttypes.h>
7  #include <stdio.h>
8  #include <system.h>
9  #include <io.h>
10
11 // LED register access defines
12 #define LED_RED(N)      (4*(N) + 1)
13 #define LED_GREEN(N)    (4*(N) + 0)
14 #define LED_BLUE(N)     (4*(N) + 2)
15 #define BLINKY_CR       3
16 #define BLINKY_SR       7
17
18 // LED CR & SR bitmask
19 #define BLINKY_CR_LOAD  0x01
20 #define BLINKY_CR_START 0x02
21 #define BLINKY_SR_BUSY  0x03
22
23 void delay(uint64_t n)
24 {
25     while (n-- > 0) {
26         asm volatile ("nop");
27     }
28 }
29
30 void led_write_data(unsigned i, uint8_t R, uint8_t G, uint8_t B)
31 {
32     IOWR_8DIRECT(BLINKY_0_BASE, LED_RED(i), R);
33     IOWR_8DIRECT(BLINKY_0_BASE, LED_GREEN(i), G);
34     IOWR_8DIRECT(BLINKY_0_BASE, LED_BLUE(i), B);
35 }
36
37 int main()
38 {
39     unsigned active_led = 0;;
40     while (1) {
41         // enter state LOAD
42         IOWR_8DIRECT(BLINKY_0_BASE, BLINKY_CR, BLINKY_CR_LOAD);
43
44         // fill data
45         unsigned int i;
46         for (i = 0; i < 16; i++) {
47             if (i == active_led) {
48                 led_write_data(i, 0, 0xff, 0);
49             } else {
50                 led_write_data(i, 0, 0, 0);
```

```
51              }
52          }
53          active_led = (active_led + 1) % 16;
54
55          // start sending
56          IOWR_8DIRECT(BLINKY_0_BASE, BLINKY_CR, BLINKY_CR_START);
57
58          // wait until done
59          while(IORD_8DIRECT(BLINKY_0_BASE, BLINKY_SR) == BLINKY_SR_BUSY);
60
61          delay(100000);
62      }
63  }
```