22.10.2024

ELEC 305 - Lab 1 Sheet - Example

Blinking LED for a Capsule Coffee Maker

Task Description

A company building capsule coffee makers has asked for a circuit that can blink an indicator LED at a specific frequency when the power of the machine is switched ON:

The blinking frequency is predetermined as ½ Hz and will not need to change during operation. The rest of the machine is irrelevant, only the LED blinking circuit should be considered here. The company has agreed to an FPGA proof-of-concept implementation for now, using a switch on an FPGA board as the power switch and an LED on the same board as the indicator LED.

Acceptance Test

The company representative will switch the power ON and simply count how many times the LED blinks within a 10-second interval with a stopwatch. Based on the ½ Hz blinking frequency requirement, the LED needs to blink 5 times within that 10-second interval.

Requirements

The requirements of the system are listed below, including interface (**`IRQ#`**), functional (**`FRQ#`**) and timing (**`TRQ#`**) requirements.

Interface requirements

`IRQ1` – Input: 1 binary switch (HIGH - LOW) that maintains its state once toggled by the user (i.e., not a “push button” that snaps back to its original position when released). The state of this switch is called “*SWT”* in the following.

`IRQ2` – Output: illumination intensity of 1 LED. This LED is called “*LED”* in the following.

Functional requirements

FRQ1 – Read *SWT* and manipulate the state of an LED blinking routine with it. Specifically, the blinking routine should start when *SWT* = HIGH, and stop when *SWT* = LOW.

FRQ2 – The LED blinking routine should alternate the illumination level of *LED* between its highest and lowest possible levels at a specified frequency called "*FREQ*" in the following.

Timing requirements

TRQ1 – The LED blinking routine should have *FREQ* = ½ Hz. Specifically, *LED* should stay ON (highest illumination) for 1 second, and then stay OFF (lowest illumination) for 1 second.

Implementation

We describe this circuit in VHDL, and use Vivado to automatically synthesize and implement it for use on the Basys3 board. Therefore, SW0 on the Basys3 board will act as *SWT* (**IRQ1**), and LD0 on the Basys3 board will act as *LED* (**IRQ2**).

SW0 is a single-pole-double-throw (SPDT) switch between ground (0V) and logic high (3.3V) on the Basys3 board, so simply reading it as a standard digital input satisfies **FRQ1**.

To satisfy **FRQ2** and **TRQ1**, we use an up-counter synchronized to the system clock on Basys3, which is a crystal-based clock running at 100 MHz. The counter starts counting from 0, counts up by 1 at every clock tick (signal name: "`count`"), and toggles a signal called `pulse` once `count` reaches 199999999. Since this takes 200M clock cycles, the `pulse` signal becomes a 50% duty cycle square wave with a 100 MHz / 200M = ½ Hz frequency. We wire the `pulse` signal directly to LD0 since the logic signals can drive the LED between high and low illumination states.

Since the 100 MHz system clock, only 1 switch and only 1 LED is used, we modify the master Basys3 XDC file as follows:

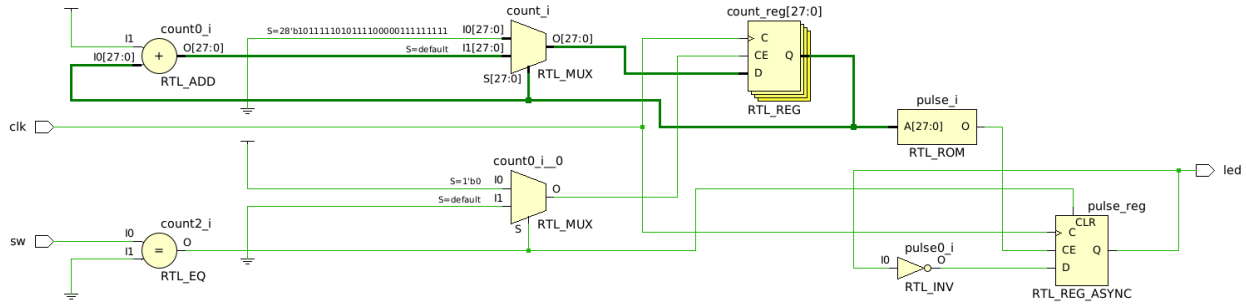
```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]


## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {sw}]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {sw[1]}]
#set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {sw[2]}]
  ...
## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led}]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
```

Note that the led and sw ports are not vectors anymore, we labeled them as singular entities by removing the square brackets.

The VHDL code is very short so we add it here to compare it with the behavioral block diagram:

```
1. library IEEE;
2. use IEEE.STD_LOGIC_1164.ALL;
3.
4. entity coffeemaker is
5.   Port ( clk : in  STD_LOGIC;
6.          led : out STD_LOGIC;
7.          sw  : in STD_LOGIC
8.        );
9. end coffeemaker;
10.
11.architecture Behavioral of coffeemaker is
12.  signal pulse : std_logic := '0';
13.  signal count : integer range 0 to 199999999 := 0;
14.begin
15.  process(clk, sw)
16.  begin
17.    if sw = '0' then
18.        pulse <= '0';
19.    elsif clk'event and clk = '1' then
20.      if count = 199999999 then
21.        count <= 0;
22.        pulse <= not pulse;
23.      else
24.        count <= count + 1;
25.      end if;
26.    end if;
27.  end process;
28.
29.  led <= pulse;
30.   end Behavioral;
```

Before synthesis, the behavioral block diagram of the circuit looks like the following (we obtained this by clicking on Open Elaborated Design → Netlist view → Schematic button):



The IF conditions are implemented via multiplexers, the `count` value and the `pulse` value are kept in 28-bit and 1-bit registers respectively, and the sw and clk are external inputs.

Note that since no starting state is specified for the LED blinking routine in the requirements, we did not explicitly program a specific reset behavior to the counter, and the default implementation is the `pulse` signal getting an async reset trigger (see bottom right, CLR input for pulse_reg) when the switch turns off. This manifests itself by the `count` register being synchronous (RTL_REG), and the `pulse` register being asynchronous (RTL_REG_ASYNC).

After characterizing the functionality of the netlist and verifying that it reflects what we intended to realize in the VHDL code, we synthesize and implement the circuit, and check utilization. Since the timing constraints are very loose for this project, we can assume that automated synthesis and implementation did their job and produced valid circuits that realize our behavioral block diagram with satisfactory timing performance (½ Hz is extremely slow compared to our system clock, so on-chip delays will not be significant). We can just check utilization.

In the Project Manager view, under Project Summary / Overview, the Utilization section shows two tables for Post-Synthesis and Post-Implementation utilization.

Synthesis:

Resource	Estimation	Available	Utilization %
LUT	10	20800	0.05
FF	29	41600	0.07
IO	3	106	2.83
BUFG	1	32	3.13

Implementation:

Resource	Utilization	Available	Utilization %
LUT	9	20800	0.04
FF	29	41600	0.07
IO	3	106	2.83
BUFG	1	32	3.13

The difference in LUT utilization shows that our implementation step optimized the LUT usage in the post-synthesis schematic and removed one redundant LUT.

Verification

The timing constraints in this project are very loose so we don't need complicated verification. The acceptance test itself is satisfactory.

References

- Syntax highlighting for VHDL done with GeShi, alternative: hohli