EE2020 / EE2026: DIGITAL FUNDAMENTALS / DIGITAL SYSTEMS

Academic Year 2017-2018, Semester 2

LAB 3: Sequential Circuits in Verilog

OVERVIEW

A sequential circuit is one where the outputs depend on the current inputs and the sequence of past inputs. As a result, a sequential circuit has memory, also called states. In this lab, some basic sequential circuits will be designed to make an LED blink at various speeds, and D-Flip-Flops will be used to debounce signals in a counter circuit.

The pre-requisites for this lab are:

- A very good understanding of Lab #2 and the various steps involved in designing modules.
- Knowing how to use the Vivado IDE well.
- Familiarity and knowledge on how to use "*Set as Top*", "*reg*" and "*wire*".
- Brief understanding of the purpose of a D-Flip-Flop (D-FF) in a sequential circuit.

This lab will cover the following:

- Using a signal that inverts itself periodically, which shall be called **CLOCK**.
- Making a physical LED blink by using the FPGA clock signal.
- Schematic and implementation of a single pulse circuit, by using two D-FFs with a slow clock signal.
- Incrementing a counter value by 1.

Tasks for this lab include:

- Creating a slower clock from a faster clock.
- Having a physical LED blink noticeably on the Basys 3 development board, by using the slower clock.
- Using a switch to make a physical LED blink at two different speeds on the Basys 3 development board.
- Using a slow clock signal to create a single pulse signal, by using a mixture of modelling methods.
- Observing a pattern in the physical LED array, through a counter that uses the single pulse circuit.

THE BLINKING LED

A simple blinking LED is required to be implemented on the FPGA. To do this, a new signal, **CLOCK**, will be introduced.

The **CLOCK** signal is an external input signal that resembles a square wave of 50% duty cycle. If this **CLOCK** signal is connected directly to a physical LED, the latter will light up when the signal is HIGH, and will switch off when the signal is LOW, as illustrated in ***Figure 3.2***.

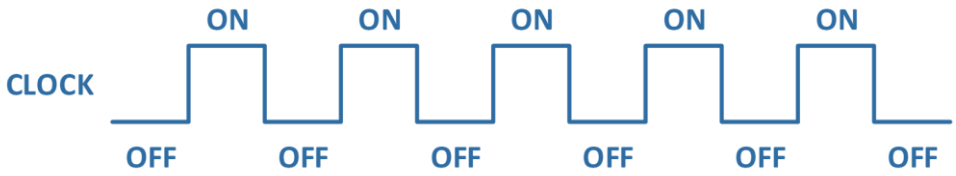


Figure 3.2: *A **CLOCK** signal with 50% duty cycle*

A simple dataflow description in Verilog for a blinky module is written first, followed by a simulation source to verify the design. To create the square wave, or **CLOCK** signal, in the simulation source, a new section of codes will now be introduced:

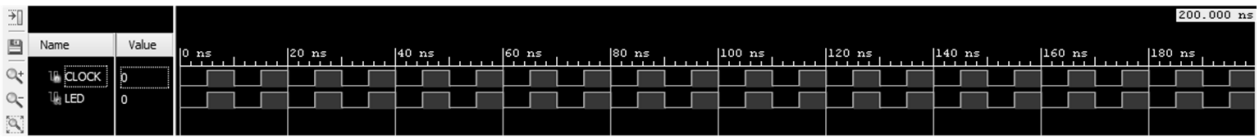
Verilog code for blinky, using the dataflow method

```
module blinky (input CLOCK, output LED);

    assign LED = CLOCK;

endmodule
```

Simulation source code to test the blinky design

```
`timescale 1ns / 1ps

module test_blinky( );

    reg CLOCK; wire LED;

    blinky dut (CLOCK, LED);

    initial begin
        CLOCK = 0;
    end

   always begin
       #5 CLOCK = ~CLOCK;
   end

endmodule
```

Expected simulation waveform for the blinky design



UNDERSTANDING | TASK 1

Based on the Verilog code and simulation results, check your understanding by answering the following questions:

1. What is the unit of time being used in the simulation source?

   _____

2. Every 5 units of time, the value of **CLOCK** is being inverted. What is the clock frequency being used in this simulation?

   _____

3. What would happen if the testbench code **CLOCK = 0** is removed?

   _____

For the hardware implementation, instead of using an external signal generator for the **CLOCK** signal to the Artix-7 FPGA, the Basys 3 development board includes a single 100 MHz clock generator connected to pin W5 of the Artix-7 FPGA.

Import and **copy into the project** the **complete and unedited** Xilinx's design constraint file template (**Basys3_Master.xdc**), followed by these steps:

1. Uncomment lines 7 to 9 to create a clock signal of 100 MHz with 50% duty cycle. If required, rename the signal to the name used in your **blinky** code. In our example, the name **CLOCK** was used, and the final changes may look similar to ***Figure 3.3***.

2. Configure the output signal **LED** that is present in your **blinky** code (or the name chosen by you while writing the code) by linking it to any physical LED on the Basys3 development board.

```
1  ## This file is a general .xdc for the Basys3 rev B board
2  ## To use it in a project:
3  ## - uncomment the lines corresponding to used pins
4  ## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
5
6  ## Clock signal
7  set_property PACKAGE_PIN W5 [get_ports CLOCK]
8      set_property IOSTANDARD LVCMOS33 [get_ports CLOCK]
9      create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports CLOCK]
10
11 ## Switches
12 #set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
13     #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
14 #set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
15     #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
```

Figure 3.1: *Modifying the Basys3_Master.xdc*

UNDERSTANDING | TASK 2

You may optionally generate the bitstream and upload your code to the Basys3 development board. What do you *notice* about the *blinking* LED?

_____

_____

THE NOTICEABLE BLINKING LED

To be able to observe a blinking LED at a frequency that is visible to the human eyes, modifications need to be done to the Verilog code. Let us introduce a temporary variable **COUNT** that is incremented by 1 at every rising edge (transition from low to high, and also called a positive edge) of the **CLOCK** signal, as shown in ***Figure 3.4***. By making use of **COUNT**, a lower frequency signal can be obtained, while the Verilog code for **COUNT** can be created by using the behavioural method of modelling.

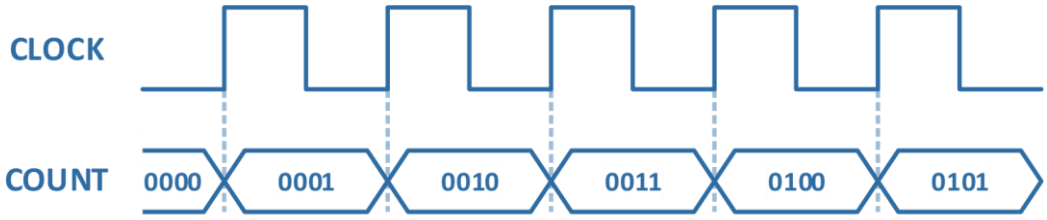


Figure 3.4: *Increasing* ***COUNT*** *at each rising edge of* ***CLOCK***

Verilog code for a slower blinky, using behavioural modelling

```
module slow_blinky_module (input CLOCK);

    reg [3:0] COUNT = 4'b0000;

    always @ (posedge CLOCK) begin
        COUNT <= COUNT + 1;
    end

endmodule
```

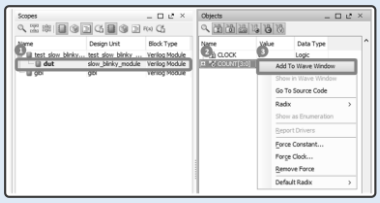
UNDERSTANDING | TASK 3

Create a simulation source for the **slow_blinky_module** design, and observe the waveform of signal **COUNT.**

[NOTE] Analysing a variable in the simulation waveform window

By default, the simulation window only shows the waveforms of input and output signals. To see the waveforms of variables during the simulation, such as the variable **COUNT**:

1. Select the **dut** under the simulation module being used
2. In the **Objects** window, right click on the **COUNT** variable
3. Choose **Add To Wave Window**



After adding the variable **COUNT** to the wave window, the current simulation needs to be re-run. Follow these steps:

1. Restart the simulation
2. Set the simulation time and units
3. Run the simulation for the amount of time set in step 2

The **COUNT** variable can then be expanded by clicking on the **+** symbol to the left of **COUNT**. This allows for every individual bit to be observed as independent waveforms:

State the frequency of **COUNT[3]**, **COUNT[2]**, **COUNT[1]** and **COUNT[0]**: _____ , _____ , _____ , _____

UNDERSTANDING | TASK 4

In the **slow_blinky_module**, insert the following line of code in the always block:

```
always @ (posedge CLOCK) begin
    COUNT <= COUNT + 1;
    LED <= ( COUNT == 4'b0000 ) ? ~LED : LED ;
end
```

Do additional modifications in the **slow_blinky_module** and the testbench code, and simulate the **slow_blinky_module** design.

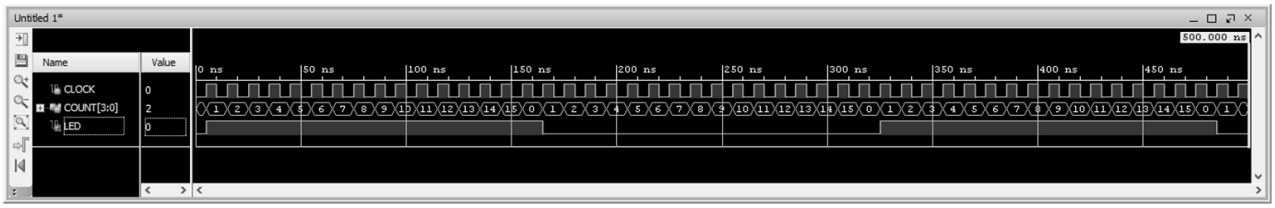
[NOTE] Hints on the modifications related to slow_blinky_module

▶ `LED` is an output signal of the design module

▶ A signal declared as **`reg`** can be reused within the design module. An example is an output signal that needs to be reused as input within the design module

▶ A design module signal declared as **`reg`** can be given an initial value, especially if toggling is involved

If the codes have been correctly written, a simulation waveform similar to what is shown below can be obtained:



Observe the waveform that you have obtained in your simulation window, and calculate the frequency of the signal **LED**:

f = _____ MHz

From your understanding of the multiplexer, state what the following line of code means:

```
LED <= ( COUNT == 4'b0000 ) ? ~LED : LED ;
```

_____

_____

UNDERSTANDING | TASK 5

Based on your knowledge obtained from **UNDERSTANDING | TASK 3** and **UNDERSTANDING | TASK 4**, calculate the number of bits for **COUNT** which will allow **LED** to have a frequency of around 4 Hz.

Number of bits required for **COUNT**: _____

Name that waveform with the frequency of 4 Hz as **SLOWCLOCK**. Modify the **slow_blinky_module** design, and implement the LED blinking at a frequency of around 4 Hz on the Basys 3 development board.

** **Warning:** Do not simulate the code. Why? **

DEBOUNCING: CREATING A SINGLE PULSE CIRCUIT

In this section, a debouncing / single pulse circuit will be created. The single pulse circuit is best analysed and understood with a slow clock, such as the **SLOWCLOCK** signal of 4 Hz. Mechanical switches cause bounce in the signal when toggled. There are many ways to implement a debouncing but for signal bounce of less than one clock period, the circuit from ***Figure 3.4*** can be used to generate a debounced output signal. The output signal will be a synchronised logic true signal that lasts for the duration of one clock cycle. In order to create the single pulse circuit, two D-FFs and an AND gate will be used, as illustrated in ***Figure 3.4***.

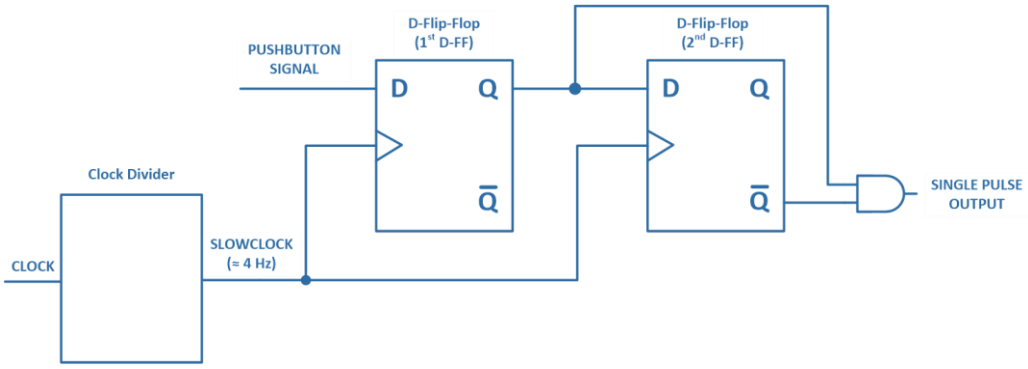


Figure 3.4: *Single pulse circuit*

The waveform for the single pulse circuit is as shown in ***Figure 3.5***.

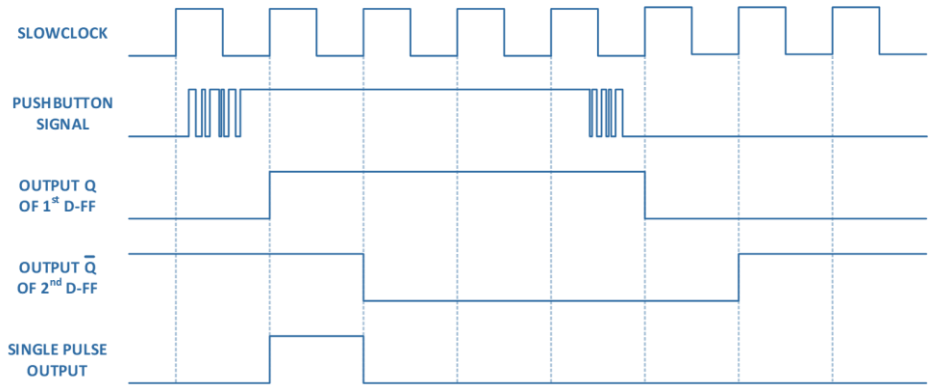


Figure 3.5: *Waveform from a single pulse circuit*

UNDERSTANDING | TASK 6

Through behavioural modelling in verilog, the code for a positive-edge triggered D-flip-flop can be created:

Verilog code for a positive-edge triggered D-flip-flop

```
module my_dff(input DFF_CLOCK, D, output reg Q);

    always @ (posedge DFF_CLOCK) begin
        Q <= D;
    end

endmodule
```

Using structural, dataflow and/or behavioural modelling, create the circuit shown earlier in ***Figure 3.4***. Simulate your single pulse design module with an appropriate testbench.

** **Note:** If you are simulating the single pulse, **temporarily** set your **SLOWCLOCK** to a frequency in the MHz range to avoid waiting for too long during the simulation **

APPLYING THE SINGLE PULSE

UNDERSTANDING | TASK 7

From **UNDERSTANDING | TASK 6**, it is noted that no matter how long the pushbutton is held, a single synchronous pulse of only one clock period is created.

Before proceeding further, ensure that your D-flip-flops are using the **SLOWCLOCK** signal of 4 Hz for this lab task. Now, an 8-bit counter whose 8 bits are represented by **LEDARRAY**, and which increments by only 1-bit no matter how long the pushbutton is pressed, can be created. Create a design module that achieves this task, and which includes the Verilog code shown below:

Partial Verilog code for a counter that increments by one at the positive edge of the synchronised single pulse signal

```
reg [7:0] LEDARRAY = 8'b0000_0000;

always @ (posedge SINGLE_PULSE) begin
    LEDARRAY <= LEDARRAY + 1;
end
```

Simulate your design to verify that your code is functioning as intended. Subsequently, implement the code on the Basys 3 development board. Press and release the pushbutton 5 times, each time recording the 8-bit **LEDARRAY** value below:

Assert Pushbutton	1st	2nd	3rd	4th	5th
Value of LEDARRAY					

What kind of pattern do you notice in the values?

_____

_____

PRACTICAL TEST IN WEEK 7

A design task that is based on what you have learnt in all the three labs (lab 1, lab 2, lab 3) will be given. More information regarding the practical test will be provided in due course.