ECE4/530 Activity 02 – Frequency Divider & Edge Detections

Sequential Components in Verilog

Sequential objects always react on a clock signal and a reset (asynchronous or synchronous). The synchronous design methodology is the most commonly used practice in designing sequential circuits where are sequential elements are controlled by a global clock signal and data is sampled and stored at the rising or falling edge of the clock. In Verilog this is modeled using an always block that is sensitive the clock edge and asynchronous rest. All signal assignments within this block are done using non-blocking assignment, whose basic syntax is: `[variable_name] <= [expression or value]`

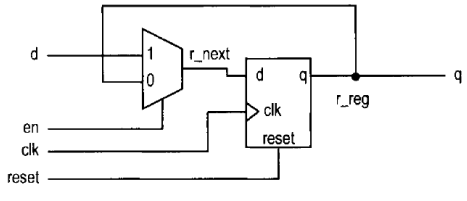
1. The basic syntax for a rising-edge triggered D-FF with a low-active Asynchronous reset is:

```
module D_ff (
              input Clk, Rst, D,
              output reg Q
             );
always@(posedge Clk or negedge Rst)
       if (Rst == 0)
              Q <= 1'b0;
       else
              Q <= D;
endmodule
```

   - This is usually called a 1-bit register – multibit registers can be done by declaring the data input and out signals as busses or vector: [31:0] D *// 32-bit register.*
   - Rising edge is expressed by the keyword *posedge* while the falling edge is by *negedge*.
   - The *always* block must be triggered by edge sensitive pulses only.
   - IMPORTANT → No other signals should be included in the sensitivity list.
   - Since the output Q is assigned inside an always block, it must be declared as reg.

2. Sometimes, it is necessary to add an enable signal to control when the FF samples its input

```
module D_ff (
              input Clk, Rst, En, D,
              output reg Q
             );
always@(posedge Clk or negedge Rst)
       if (!Rst)// same as Rst == 0
              Q <= 1'b0;
       else if (En)
              Q <= D;
endmodule
```



   - Each signal is assigned in the if and else branches gets a register.
   - The Enable signal is examined only on the clock edge – i.e. Synchronously.
   - If Enable is not asserted, the FF keeps its previous value. Note there is no else branch for the if (En) statement
   - Enables are used also to eliminate gated clock designs
   - This same design can be done using two always blocks. One for the D-FF and the other for the Mux with the proper wiring connections.

Testbench for sequential circuits

- Define the timescale directive
  - *`timescale 1ns/1ps*
- A test bench module has no ports
  - *module tb_Design ();*
- Declare local signals as tb_design_port. All design inputs are declared as reg and outputs as wire
  - *reg tb_Clk, tb_data_in, tb_en*
  - *wire tb_data_out*
- Declare a constant for the clock period
  - *parameter period = 50*
- Instantiate the design under test UUT same as before
- Generate a clock
  - Using an initial block

```
initial
begin
    CLk = 1'b0;
    forever #(period/2) Clk = ~Clk;
end
```

  - Using always block

```
always
begin
    CLk = 1'b0;       // Clock is low
    #(period/2);      // wait for half a period
    Clk = 1'b1;       // set clock to high
    #(period/2);      // wait for half a period
end
```

  - For a fixed number of clock cycles

```
integer i, num_cycles = 100;
parameter period = 50;
always
    begin
        Clk = 0;
        for (i =0; i<num_cycles; i=i+1)
             #(period/2) Clk = ~Clk;
        $stop;
    end
```

- Generate a low-active reset that is not aligned with the clock edges and covers both edges of the clock

```
initial
begin
    Rst = 1'b1;       // initial value is 1 (inactive)
    #(period/4);      // misalign with the clock edge
    RST = 1'b0;       // activate the reset
    # period;         // wait for a period
    Rst = 1'b1;       // deactivate the reset
end
```

- Generate the stimulus
  - If the system is positive-edge triggered, inputs must be stable around the rising edge of the clock signal to satisfy setup and hold time constraints. An easy way to achieve this is to change input signal's value at the negative edge (opposite of the triggering edge).
    - ***@(negedge Clk)***
  - To wait for multiple clock cycles use
    - ***repeat (10) @negedge Clk; //repeat 10 times***
  - To wait for a special condition on a signal such as "Data_out is equal to 2"
    - ***(wait Data_out == 2);***
  - To wait until a signal change in value
    - ***wait (signal-name);***
  - To display the results in a textual format on the tcl window use $display
    - Syntax: ***$display (["format_string"] , [argument] , [argument, …]***
    - Example: ***$display ("at %d; signal x = %b", $time, x)***
    - The print out should be: *at 5230; signal x = 00110001*
  - $monitor system task displays text when an argument changes its value
    - Example:
```
$display(Time    Test_in0  Test_in1  Test_out");
$monitor("%d        %b       %d        %b ,
        $time , test_in0, test_in1, test_out
```
    - Output:

time	test_in0	Test_in1	Test_out
0	00	00	1
200	01	00	0
400	01	11	0
600	10	10	1
800	11	11	0

  - Do not forget to end your simulation with $finish or $stop

Part 1: Frequency Divider

Background:

A frequency divider is a simple component whose job is to reduce an input frequency. This is sometimes also referred to as a clock divider. The frequency divider needs to be implemented through the use of a scaling factor and a counter. The scaling factor is a relationship between the input frequency and the desired output frequency such that:

$$Scale\ Factor = \frac{f_{in}}{f_{out}}$$

Assuming that we have some input frequency of 100 MHz, and need an output frequency of 500 Hz, we could calculate this as:

$$Scale\ Factor = \frac{100MHz}{500Hz} = 200{,}000$$

Thus, the counter of the frequency divider generates an output signal every 200,000 clock cycles. This then provides a simple way for slowing down a signal for usage as an enable signal for other sequential components that might not be able to operate with the faster (original) clock signal.

Procedure:

For this part of the exercise, you are to create a Verilog implementation of a frequency divider along with corresponding testbench. You will take the input frequency of the Zybo board and slow it down using two different modules to blink two different LEDS: [LED02 and LED03]. LED2 should blink at a rate of 5Hz, and LED3 at 1Hz. To do this, your module should have four ports: *Clock_in*, *Clock_1Hz*, *Clock_5Hz*, and *Reset*.

Think about what your always block needs to contain in order to be asynchronously reset, and to be sensitive to the clock input. Keep in mind, a square wave clock signal is typically a 50% duty cycle.

Note 1: There is a 125Mhz clock available on the board. Make sure to connect it to the appropriate pin of the FPGA through the constraints file.

Note 2: Use a reasonable period for the clock signal in the testbench so that you don't wait long time for the simulation to end.

Note 3: Use PB0 for global reset. (it should reset when you push the button)

Part 2: Synchronization and Edge Detection

The Problem

Quite often, some sub-system components need to react to a change on specific control signal. That can be an external input, something saying that another part of the circuit has done it's job and that we can continue (strobe signal). All sorts of scenarios exist that call for a signal generated by one part of a system to be detected by another part or a different system.

Why is it a problem?

All this asynchronous stuff is fine and dandy if you're still in 1980, but not so good if you want to use an FPGA. Why? Well, every time the keyword “posedge” is used on any signal, the FPGA tools think "Aha! A clock!" and act accordingly. They route the signal in question through a clock buffer, and if there are a few of these edge-detect fragments, the FPGA will pretty soon run out of clock buffers. Then the tools start using general routing for clocks and the nice, dependable, low-skew clocks design go completely out of the window.

How do I solve the problem?

Don't do these things.

Oh. You want more? Well, the basic principle is that FPGAs do asynchronous stuff really very badly. They thrive on synchronicity (is that even a word?). "KEEP IT SYNCHRONOUS". Yes, unless your design is very small you're going to have to have more than one clock domain, but keep the number of clocks as small as humanly possible. Instead of generating a new divided or gated clock, consider using a clock enable to control one widespread system clock. That makes best use of the clocking resources on the chip.

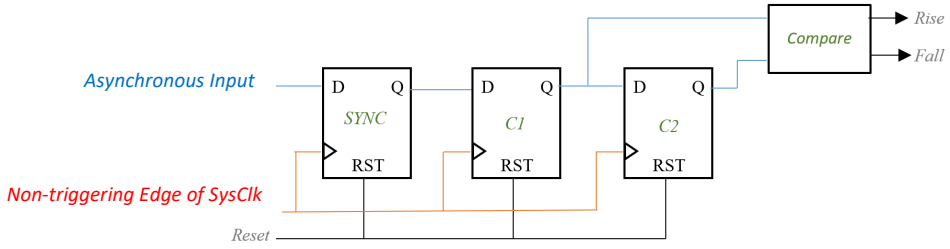
Don't use internal tri-states at all. Don't use latches unless you know exactly why you need them; your FPGA synthesis tool gives you a warning about latches for a good reason: they're almost always a mistake and they really, really, don't help the implementation or timing checking of the design during place and route. Make sure you are familiar with the sort of HDL code that generates latches, so you can avoid writing them.

So how do I do that edge detection?

How can you provide a synchronous notification of an asynchronous edge? First thing we need to do is to get a synchronous version of that asynchronous input signal.

Why? – Because if the input signal changes within a flip-flop's setup and hold times the output may be unusable for various reasons; the flip-flop may or may not have propagated the new value, or perhaps got confused for a while and not settled to any real output value ("metastability"). If the input signal changes within the "metastability window" the output could take a long (theoretically infinite) time to settle to a stable value. That time could well be longer than one clock cycle, so we add another flip-flop just in case. It's vanishingly unlikely for the second flip-flop to get hit by metastability.

Synchronization is conventionally done with a two-stage shift-register that is clocked by the target domain's clock. That's great, our input is now synchronous to the sysclk. We need to detect a change, and we can do that by extending the shift-register a little and comparing the values of different bits:



The first stage of the shifter takes the hit of timing errors, and we compare the next two stages (C1 and C2) to see whether we have a rise or a fall on the input signal. We output two signals that indicate a rise or a fall; each will be high for one sysclk cycle. The compare logic is trivially simple.

Other things to think about

How are you going to use the rise and fall pulses? They are high (active) for only one cycle at a time, so if you need the indication for longer you may need to register their values. They are synchronous to sysclk, so to use them in blocks with other clocks you will of course need to synchronize them.

What is the nature of the thing generating the input signal? Is it a clean change from one value to another or does it glitch or bounces? You might need to clean it up with some sort of glitch filter before you use this edge-detection. If your signal comes from something really messy like a mechanical switch you may need something more than the usual shift-register de-bouncer.

Another lesson about synchronization: A student tells me about a horrible bug he uncovered for another student. The student's design had two FSMs that kept getting into deadlock, and he'd burned a few fuse-based chips trying to diagnose the problem. Like the nastiest bugs it was intermittent and hard to reproduce. RTL simulation was absolutely perfect, so what was the issue? The engineer had very carefully synchronized signals between clock domains, as you do, but in his eagerness to obey the rules, he had synchronized one signal into a clock domain in two different places. As a result, minute timing differences in the paths to these two synchronizers meant that the two FSMs working together saw different vales for this signal. Hence the deadlock. Bottom line: only synchronize a signal once per clock domain.

Procedure

Build the synchronizer and edge detector as described above. Simulate it and check the proper functionality. Elaborate the design and generate the schematic. Create a top module with instantiations of the modules from part I and 2 and Name it Activity_02. Connect the 1Hz signal from part 1 to the Sysclk input and PB3 to the signal input. Connect the Rise output port to LED0 and the Fall output port to LED1. Push the PBO and observe the status of the LEDS. Take a picture and include it with your submission.

Submission

Submit all the Verilog files and testbench files (well commented files only – no whole project) Take a snap shot of the simulation waveforms, zoom to a visible view and annotate them at critical points and add comments. Do the same with the synthesized design. Annotate how such components are related to the code you designed. Take pictures of the board showing the status for part1 (annotate it) and a short video of part2 showing the LEDS blinking (narrate it). Put all this information in the report file and submit the report along with the **well commented** Verilog design and testbench files, constraints file, bitstream file, picture and video into Canvas. Name the submissions as *Lastname_Firstname_A_02*