CALIFORNIA STATE POLYTECHNIC UNIVERSITY, POMONA
COLLEGE OF ENGINEERING

ECE 2300L Fall 2019

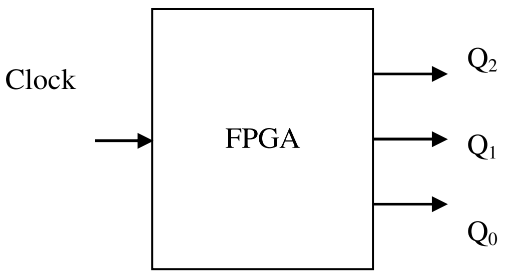
LAB 10: Counters

Objective: Build a four-bit up counter using T and D flip-flops in Verilog. Start the use of instantiation technique.

Introduction:

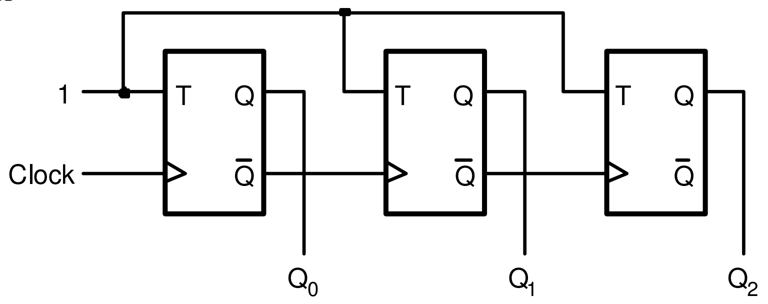
The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation. The counters built in this lab are asynchronous counters, or ripple counters, because the clock signals are propagated from one stage to another (not synchronous).

Top-View of the counter

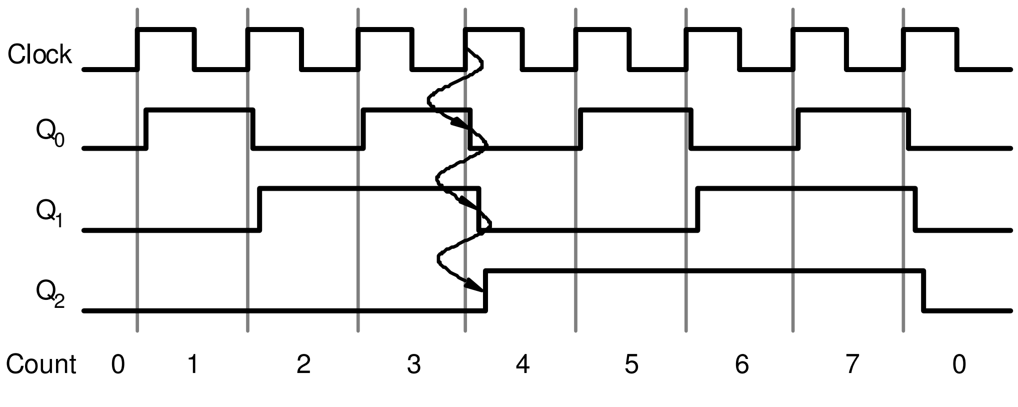


Use LEDs as the outputs device

Up-counter

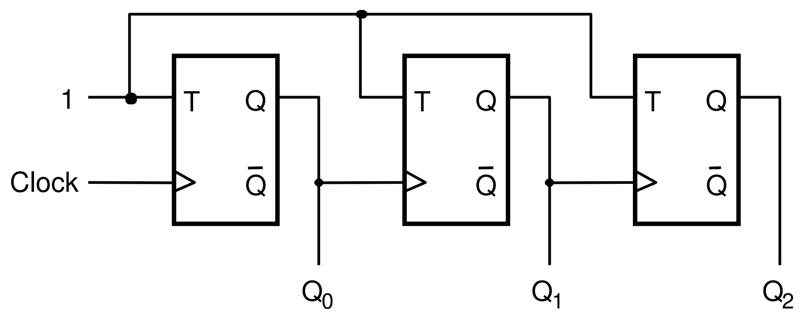


(a) Circuit

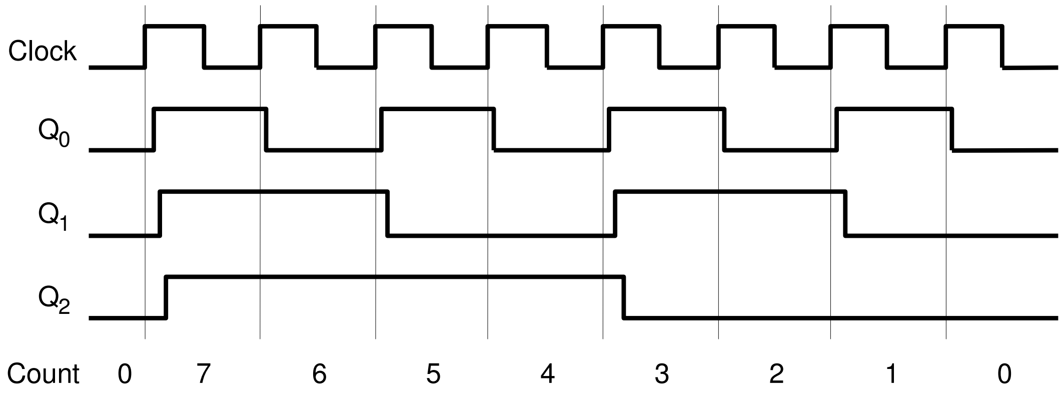


(b) Timing diagram

Down-counter



(a) Circuit



(b) Timing diagram

As observed on the schematics above, the Q output of each stage is used as the clock for the next stage. Hence, the output of the next stage depends on the output of the previous stage, thus there will be a ripple effect.

PRELAB:

1. Construct the corresponding Verilog code to build the 4-bit up-counter using T flip-flops.

   Note: You need to start to use the notion of instantiation which involves the implementation of a design through the use of building blocks.

   In the previous lab part 4, you have built a T flip-flop function. Let assume that module has the following format:

   ```
   module Tff (Tinput, clock, Q);
   ```

   where 'Tinput' is the T input, 'clock' is the input clock and 'Q' is the output of the flip flop.

From that module, you can create a 2-bit up-counter based on part of the schematics provided above:

```
module up_counter_2bit (clock, Q);
input clock;
output [1:0]Q;

        Tff  t0  (1, clock, Q[0]);
        Tff  t1  (1, ~Q[0], Q[1]);


endmodule
```

where Tff is the original T flip flop module while t0 and t1 are called the module instance. If you follow the schematics, you should be able to realize how the module instances are setup. Based on that 2-bit counter example, you should be able to extend the design into a 3-bit counter and then a 4-bit counter design.

In addition, for having a better way to implement the design and for better documentation of the process, it is recommended to place the instantiation calls as follows:

```
//      Tff      (Tinput, clock , Q      );
        Tff  t0 (1       , clock , Q[0] );
        Tff  t1 (1       , ~Q[0], Q[1] );
```

First, copy the original definition of the module and place it at the beginning of the calls. Next, make that line as a commented line by placing the '//'. The next step is to line up the instance calls such that each field is lined up with the field in the commented line. Moving the ',' would allow the fields to be lined up.

By using this technique the designer will know what to apply in each field as defined in the original definition of the function and therefore fewer mistakes will be made. In addition, this will better document the design for everyone to better understand the design.

2. Simulate your 4-bit up counter circuit to make sure it is working properly. Use the example to generate a good amount of clock pulses needed to verify the entire count sequence.

```
module up_cnt_4bit_tff_sim;

// Inputs
	reg clock;

// Outputs
	wire [3:0] Q;

// Instantiate the Unit Under Test (UUT)
	up_counter_4bit uut (clock, Q);

initial begin
	clock = 0;
	forever #50 clock = ~clock;
end

endmodule
```

LAB:

1. First, you need to design a 7-segment decoder that takes 4 inputs (from switches) and based on the binary value of those inputs, 7 segment outputs are generated (See figure 1 on attached schematics). Look at the following link to get the decoding scheme:

   http://en.wikipedia.org/wiki/Seven-segment_display

   Scroll down to the section with the heading 'Hexadecimal'.

   Fill in the remaining module:

```
module seven_segment_decoder(sw, seg);
input [3:0] sw;
output reg [6:0] seg;


always @(sw)
	case (sw)
	//	3210			gfedcba
		4'b0000:	seg =7'b1000000;	// Number 0
		4'b0001:	seg =7'b1111001;	// Number 1
		.
		.
		4'b1111:	seg = 7'b0001110;	// Number F
		default:	seg = 7'b1111111;
	endcase
 endmodule
```

Implement the design using the on-board set of 7-segment display of your Basys3 or Nexys4 board. The '.xdc' file does have a section that shows the assignments of the different segments on the board (seg_a to seg_g).

Remember that those 7-segments are common-anode type meaning a '0' will turn the segment while a '1' will turn it off. Be aware of that all the 7-segment displays on the board will show the same number. We will deal later on how to turn on only one display.

2. Next, build the counter circuit on the Prelab using 4 LEDs as the output devices and one of the push-button switches as the clock input. See Figure 2 on the attached schematics. Demonstrate the counter using the push-button switch to generate the clock. The LEDs should show the binary count going from 0000 to 1111 and back to 0000 to restart the count again. **Be aware of the debouncing effect of the push-button. Due to its mechanical construction, a push-then-release on the button can potentially create an effect of two or more clock pulses being generated and that can make your output skip a count or more.**

3. Copy the module **"slower" clock generator** provided at the Appendix A at the end of this document to your design as a separate module (to be appended after your main counter module). Instantiate that module into your original design by feeding the clock output of this clock module to your counter's clock input. The purpose of this approach is to replace your push button switch by a free running clock. The input clock is now 100MHz. It gets divided down to a 1Hz clock (1 second period) through the use of the "slower" clock generator and then that becomes the main clock to your 4-bit counter. The 100 MHz clock is the same pin as used in the last part of the previous lab. See Schematics Figure 3.

   Your main design should be an instantiation of two modules. The first one is your clock generator module and the second one is your 4-bit counter from part 2.

   Demonstrate the new counter. It should count up automatically every second.

4. Instead of the T Flip flops, change the original design by using a 4 D-type flip flops. The input clock is still the 100 MHz clock. See Schematics Figure 4. You need to replace your 4-bit counter based on T flip flops with a new one strictly implemented using D flip flops. There are a lot of simple examples of Verilog code using D-type counters on the web and the code should be very simple.

5. Next, take the 4 outputs of the counter and feed them into a 7-segment decoder to drive 7 outputs for the 7-segment display. See Schematics Figure 5. The design will take the 100-Mhz clock as the only input. From there, a 1-Hz clock output is generated and then gets fed into the 4-bit counter. The outputs of that counter are then inputs to the 7-segment decoder whereas 7 outputs (seg[6:0]) are generated.

If implemented properly, the 7-segment display should show a count that goes from 0 to F and it will repeat itself.

There are 4 7-segment digits on the Basys3 and 8 7-segment digits on the Nexys4. We only need to turn on one 7-segment display. To do so, we need to control the 'an[x]' pins that are also outputs from this design. The module header for the design should be:

Module counter_to_7Segment (clock, seg, an)

For the Basys3, the outputs for 'an' should be 'output [3:0] an;' and add the line 'assign an[3:0] = 4'b1110;'

For the Nexys4, we should use 'output [7:0] an;' and 'assign an[7:0] = 8'b11111110;'

APPENDIX A

Building a "Slower" Clock Generator

The oscillator provided on your FPGA board is 100 MHz, which is way too fast for us to recognize the counter outputs. Since $2^{25} < 50,000,000 < 2^{26}$, a 27-bit counter would be needed to divide the frequency to generate a 1 Hz signal (or something on that range so that you can tell the differences on the outputs). Generate the clock signals by dividing the input 100-Mhz clock signal (see the '.xdc' file of your board for the proper pin to use). The following code shows an example:

```
module slowerClkGen(clk_100Mhz, clk_1Hz);

input clk_100Mhz;
output reg clk_1Hz;

reg [26:0] counter;

   always @ (posedge clk_100Mhz)
   begin
        counter = counter +1;              // Increment counter
        if (counter == 50_000_000)         // check if it reaches 50,000,000
           begin
                clk_1Hz=~clk_1Hz;          // if so, then flip the logic state of the
                counter =0;                // output and reset the counte
           end

   end
endmodule
```