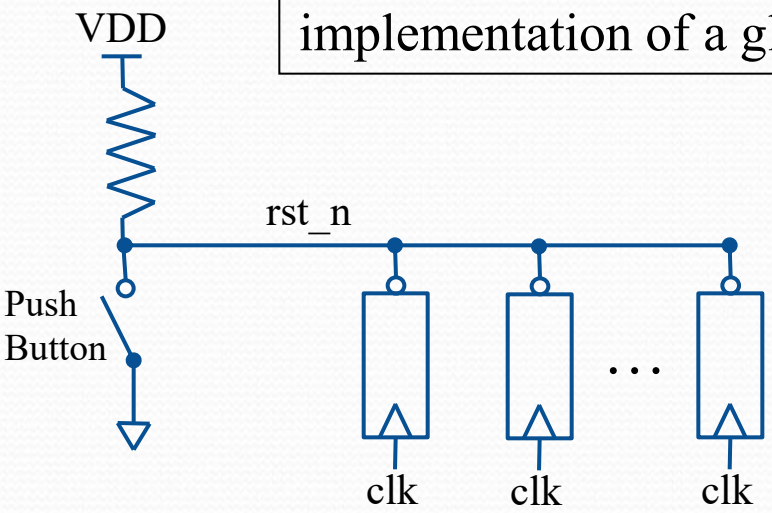
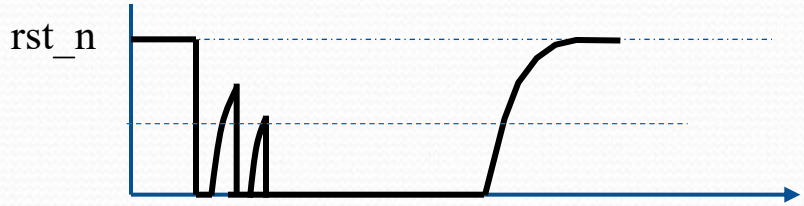
Obtaining a Decent rst_n Signal:

- On the FPGA board we have a couple of push buttons. We will use one as the source for our asynchronous reset.
- It is simply a momentary push button switch to ground with a pull-up resistor.

Imagine what a disaster this implementation of a global reset would be!!





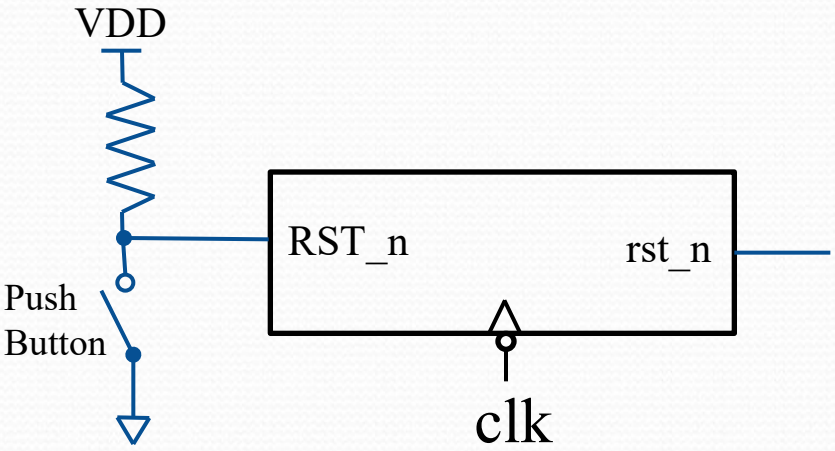
Button pushed (has bounce)

Button released (RC approach)

When does reset deassert relative to clock?

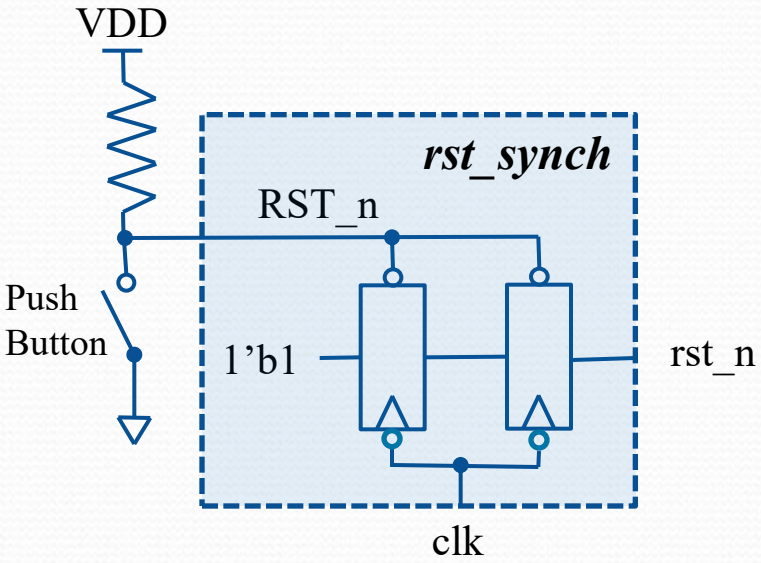
Remember…you want your reset de-asserted on the opposite edge of clock that your other flops are active on. This means we want our reset to de-assert (rise) on negative edge of clock.

Exercise 11 (Synchronizing a PB reset):



We want to build a reset synchronizer that takes in the raw push button signal and creates a signal that is deasserted at the negative edge of clock.

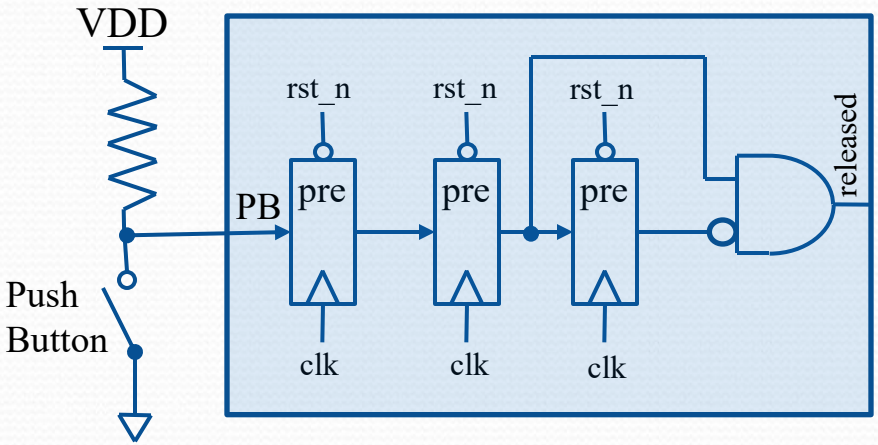
It will have an interface of:
RST_n = raw input from push button
clk = clock, and we use negative edge
rst_n = our syncrhronized output which will form the global reset to the rest of our chip.



Push of the button will asynch reset the two flops. When button is released we have a double flopping (metastability reasons) to produce our global **rst_n**. The flops are negative edge triggered so our global reset will deassert on the opposite edge of all our other flops.

Code this reset synch unit (**rst_synch.sv**)

Exercise 11 (Synching & Edge Detecting a PB input):



PB_release.sv

NOTE: these flops are asynch preset not reset

Study this…does it make sense?

- The first two flops are just double flopping for meta-stability purposes. PB is asych input right?
- Third flop is used to implement a rising edge detector. If the input to 3rd flop is high, but it output is low then a rising edge must be coming through (i.e. the release of the button.)

Implement this in system Verilog. Call it: **PB_release.sv**

Don't worry about testing it yet

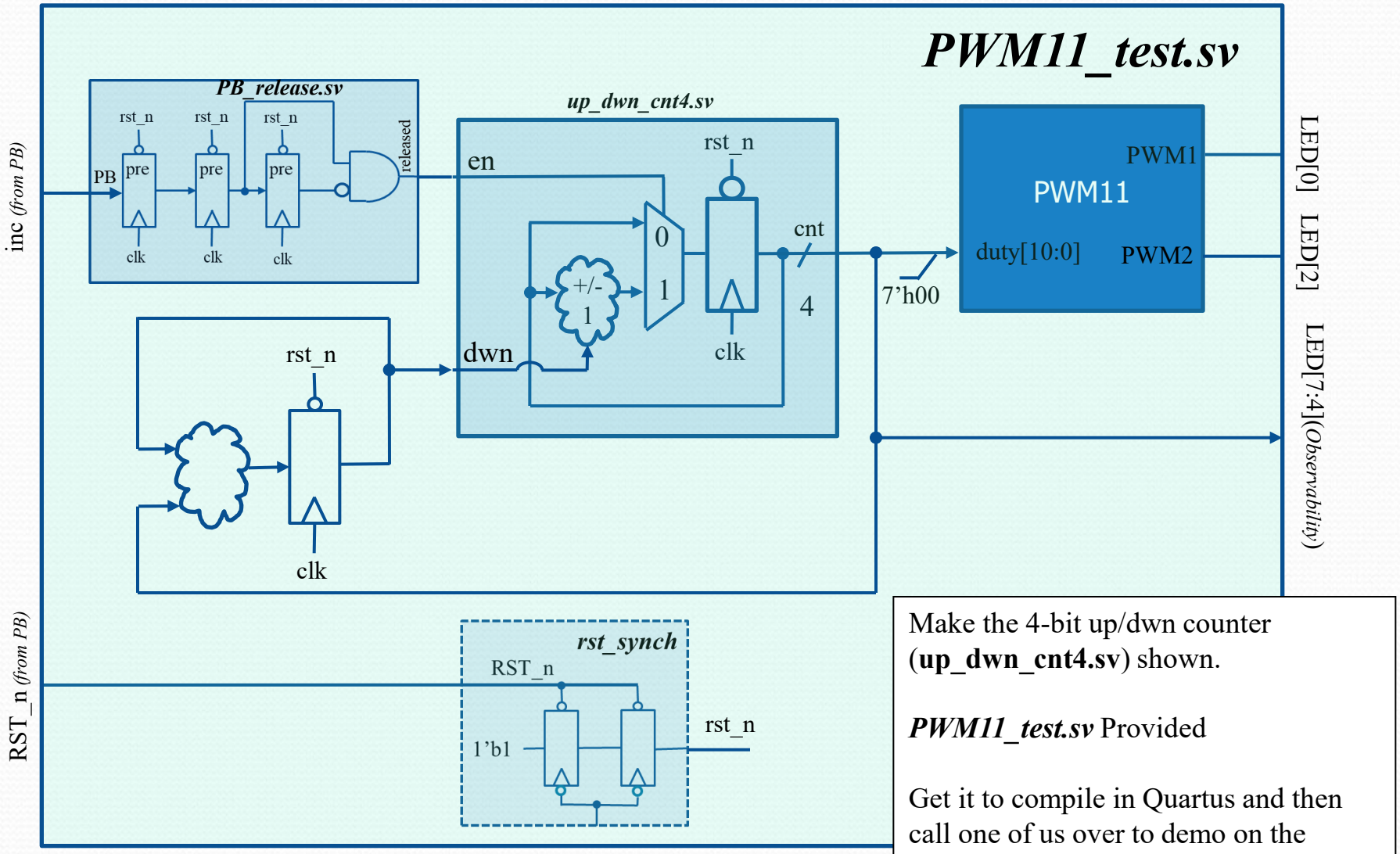
Exercise 11 (Testing of PWM11):

- In Exercise10 you coded **PWM11.sv** which has the interface shown below.

Signal:	Dir:	Description:
clk	in	50MHz clock
rst_n	in	Active low asynch reset
duty[10:0]	in	Result from balance controller telling how fast to run each motor.
PWM1 & PWM2	out	Output to the H-bridge chip controlling the DC motor, true & complement with non-overlap
PWM_synch	out	Used to synch changes in duty
ovr_I_blank	out	Used to ignore over current detect

- **PWM11_test.sv** is provided and is a wrapper that will use your PWM11 to vary the intensity of a LED.
- The test wrapper (**PWM11_test.sv**) will contain a 4-bit up/down counter that is enabled by the signal from **PB_release.sv**.
- The 4-bit counter is connected to bits [10:7] of **duty**. Bits [6:0] being 0. You need to code this simple up/down counter.
- The counter will start at 0000 and initially count up with every push of a button (coming from **PB_release.sv**) and when it hits its full value (1111) it will toggle a flop and then start counting down.

Exercise 11 (Testing of PWM11):



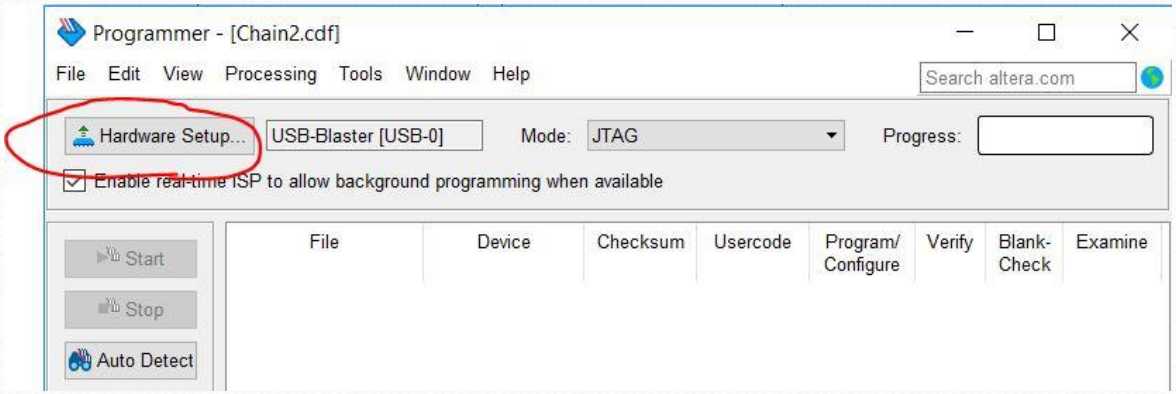
Make the 4-bit up/dwn counter (**up_dwn_cnt4.sv**) shown.

PWM11_test.sv Provided

Get it to compile in Quartus and then call one of us over to demo on the FPGA board.

Exercise 11 (Mapping PWM11_test to DE0-Nano)

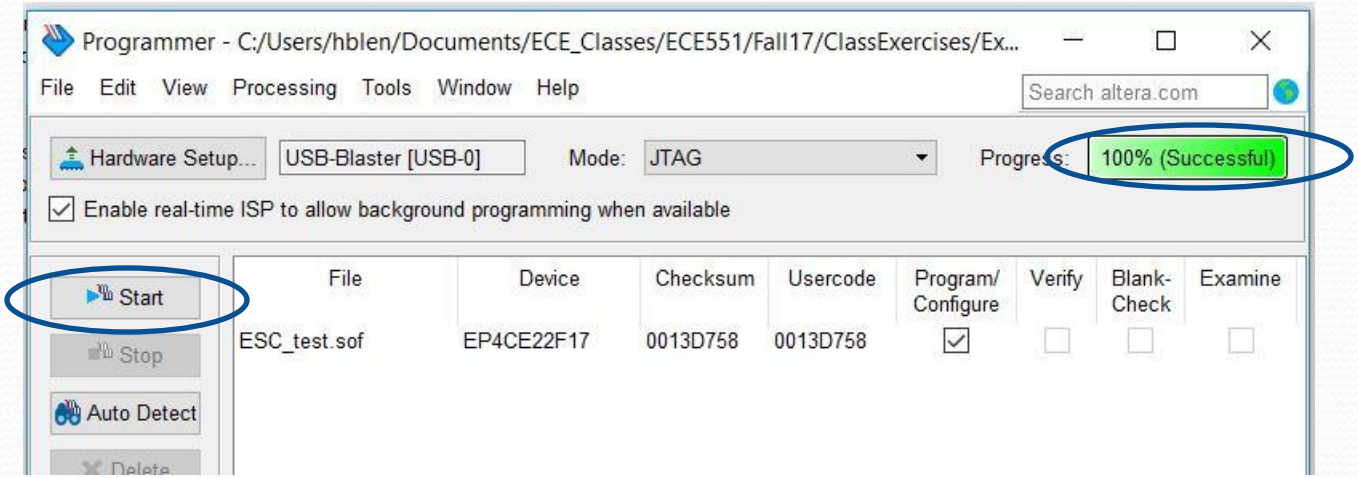
- Create an Exercise11 directory under your ECE551 area of your I: drive
- Ensure the following verilog files reside in this directory: **rst_synch.sv, PB_release.sv, up_dwn_cnt4.sv, PWM11_test.sv, and PWM11.sv**
- Download **PWM11_test.qpf** (Quartus Project File) and **PWM11_test.qsf** (Quartus Settings File) from the website and store in your Exercise11 directory
- Open up Quartus
  - Do a: **File → Open Project** and open up the **PWM11_test.qpf**
  - Compile the design and fix any errors
  - Call over Eric/Julie or Khailanii/Ting-Hung over to test on an FPGA board.
  - Do a: **Tools → Programmer** and check that the USB Blaster shows up (see below) (you may have to wait a while on these CAE machines for it to enumerate)



Might have to go under "Hardware Setup" to get it to choose USB-Blaster

Exercise 11 (Mapping PWM11_Test to DE0-Nano)

- Program the DE0-Nano



- Hit “Start” and look for 100% Success

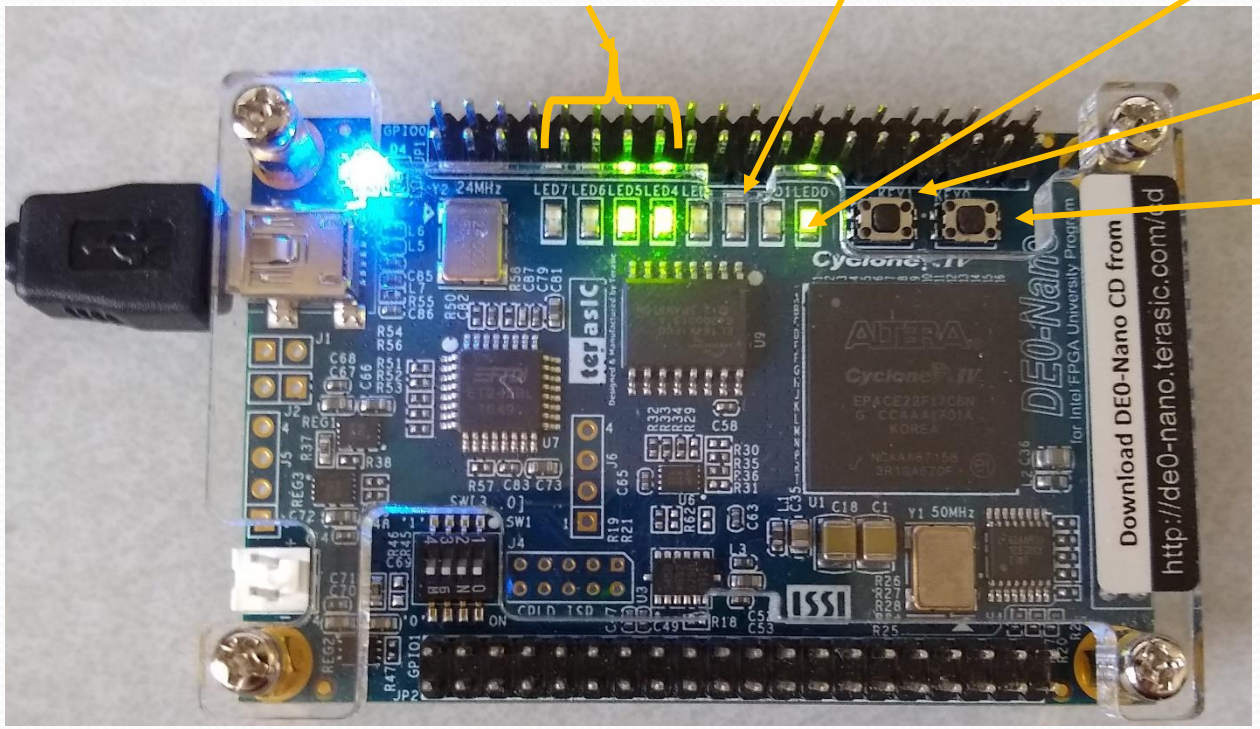
- See next page for mapping of functions to DE0-Nano

Exercise 11 (Mapping PWM11_Test to DE0-Nano)

Upper nibble of LEDs will be your 4-bit intensity counter

LED[2] will vary in intensity with duty cycle of PWM

Lowest bit of LEDs will vary in intensity with duty cycle of PWM



"inc" push button

"RST_n" push button

Test your design. Intensity of LED[0] should increase at first as counter increases. When count gets to 1111 then counter should reverse and start counting down. Call us over when you have it working and we will "check you off"