scientific
analog

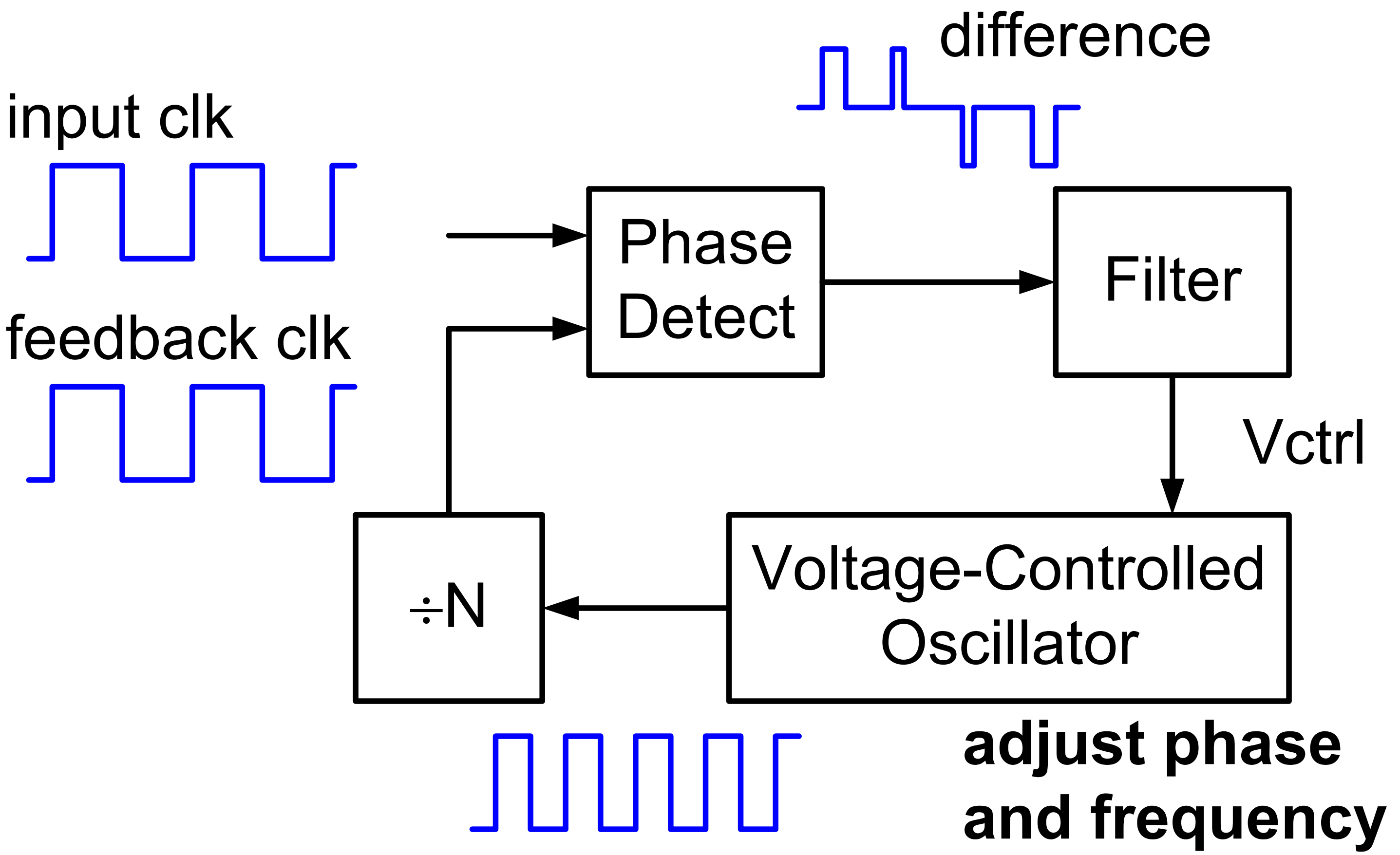
Modeling Digital PLL Components in *XMODEL*

Scientific Analog, Inc.

July 2017

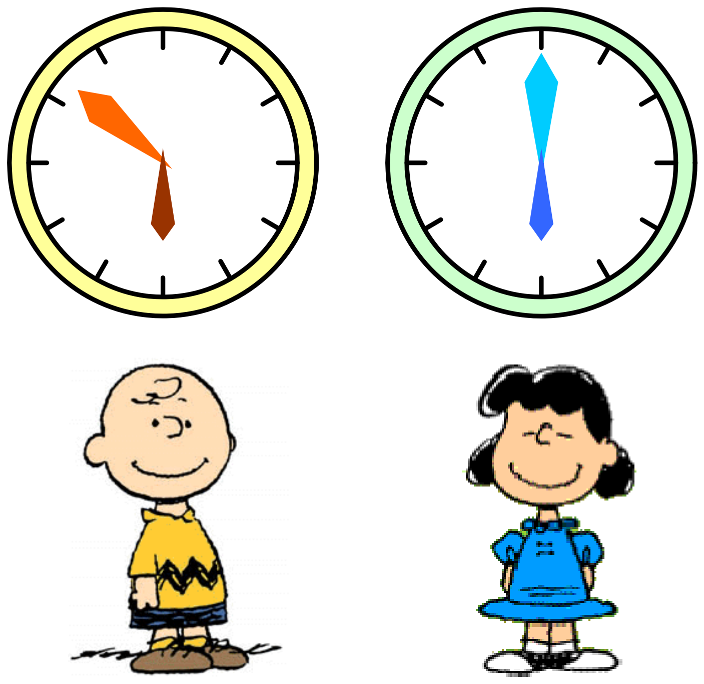
Phase-Locked Loop (PLL)

- PLL is a system that aligns its output clock to the input clock via feedback



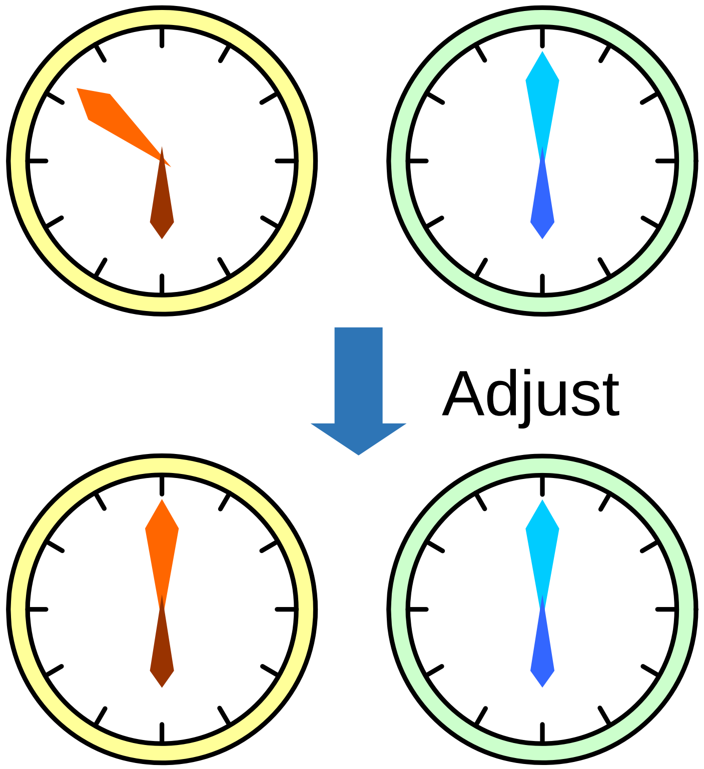
Synchronize Two Clocks

- Charlie and Lucy want to meet at 6 o' clock sharp everyday
- But their watches have
  - phase offset
  - frequency offset
- Only Lucy can adjust Charlie's watch when they meet



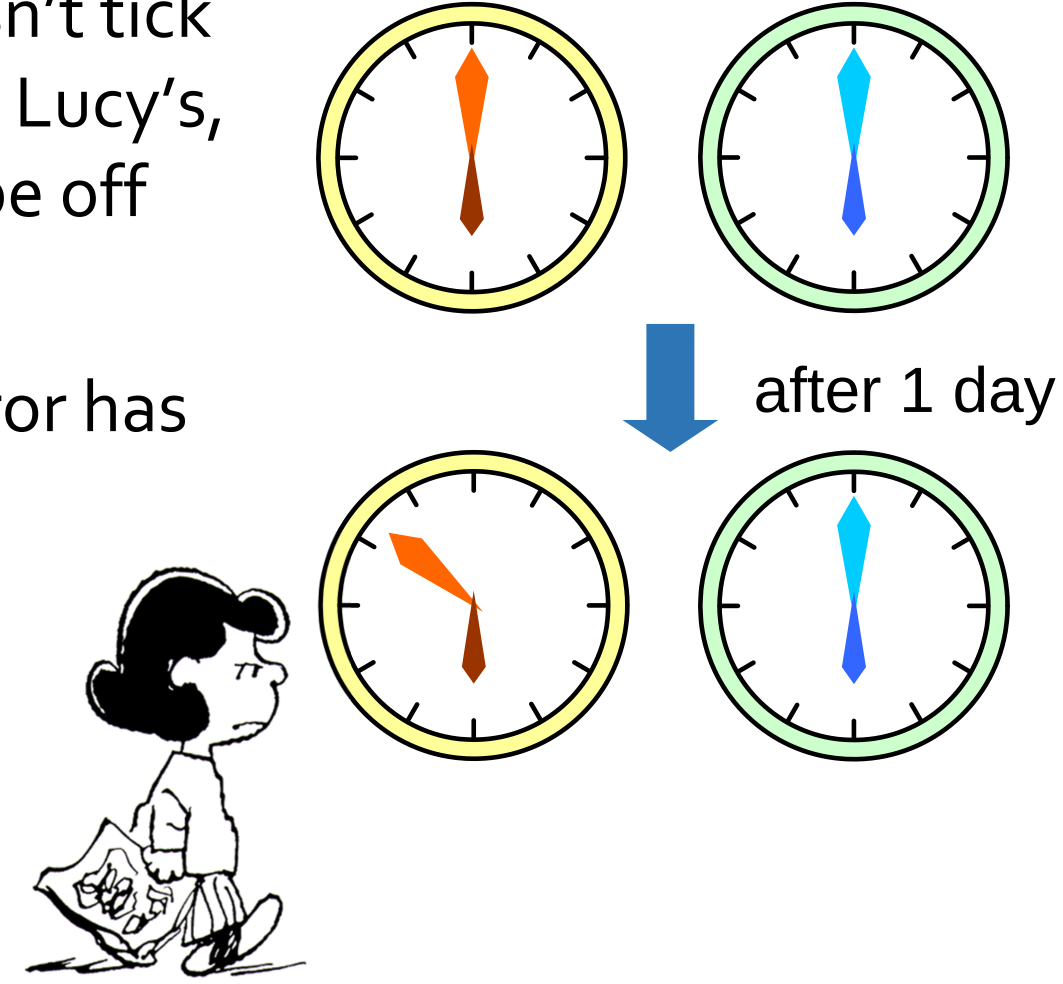
Lucy's 1st Idea

- Adjust Charlie's watch so that it points the same time with Lucy's
- Hoping that they will stay synchronized till tomorrow
- Will it work?



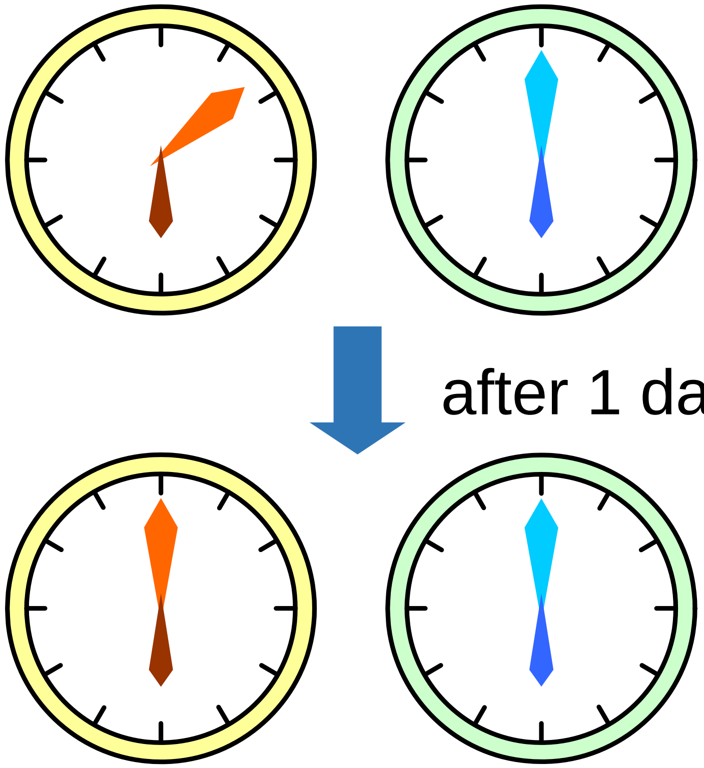
Frequency Offset Causes Phase Drift

- If Charlie's clock doesn't tick at the same rate with Lucy's, his clock will always be off after a day
- Lucy realizes time error has two components
  - phase error
  - frequency error

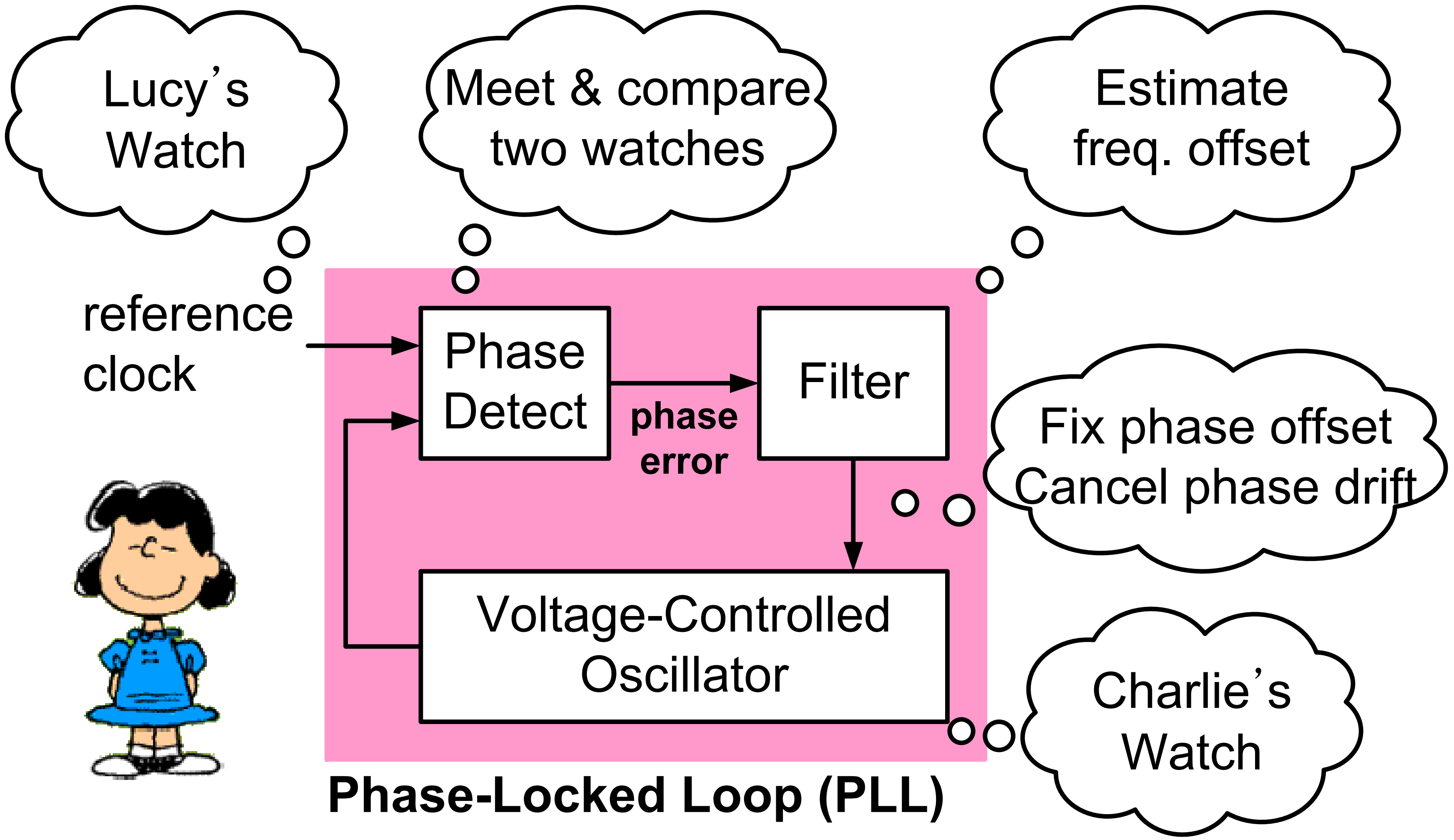


Lucy's 2nd Idea

- Cancel the phase drift in advance
  - so the clocks will be synchronized the next day
- But Lucy must know the drifting rate (freq. offset)
  - Estimate it based on the past time differences
  - while fixing phase offsets



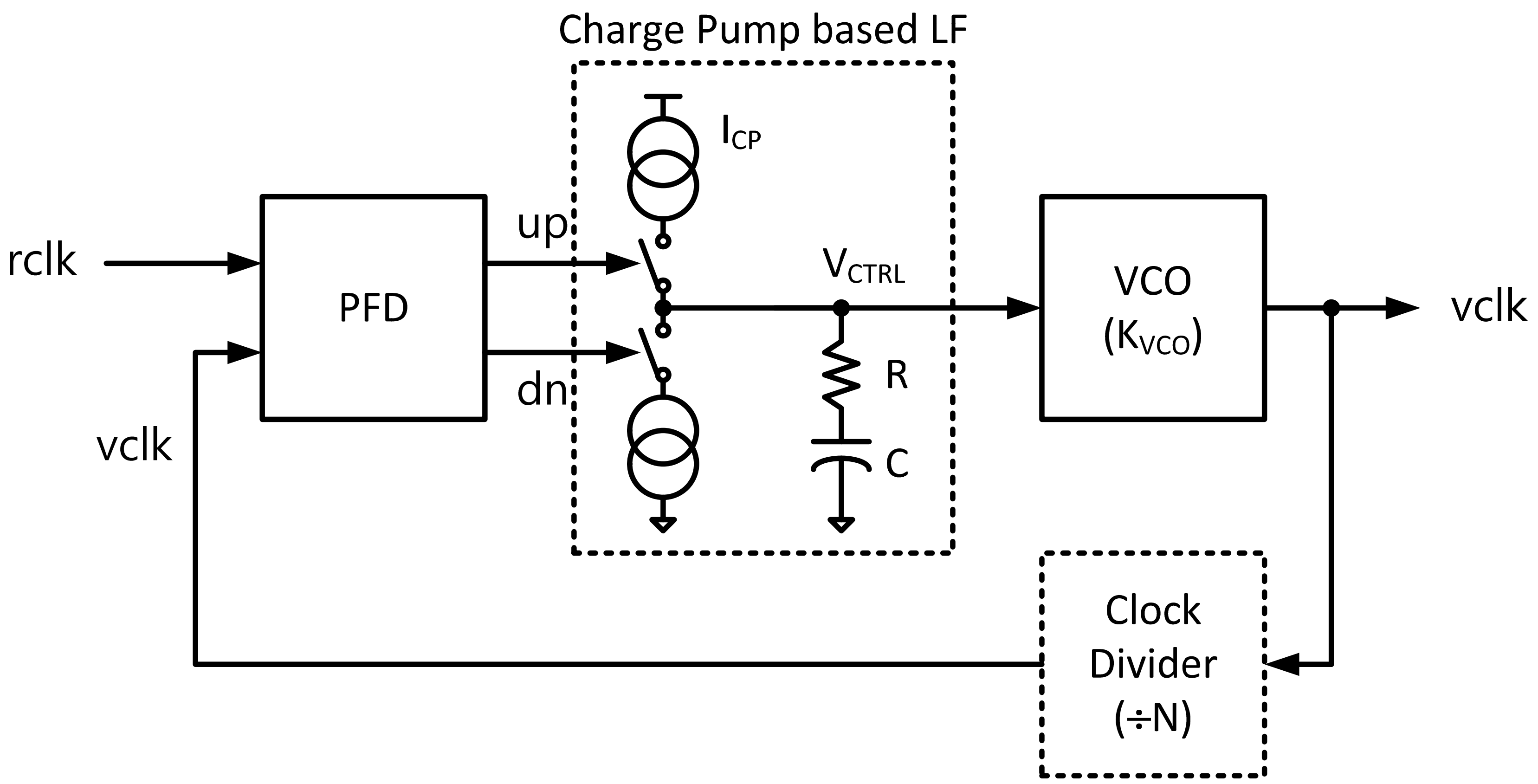
That's What a PLL Does



PLL Control Basics

- Goal is to make the reference phase (ϕ_{ref}) and feedback phase (ϕ_{fb}) the same
  - Make Lucy and Charlie arrive at the same time
- Upon the detection of the phase error (ϕ_{err}), the PLL adjusts two things from the output clock:
  - The phase: $\phi_{out} \leftarrow \phi_{out} + K_1 \cdot \phi_{err}$
    (The current time of Charlie's watch)
  - The frequency: $\omega_{out} \leftarrow \omega_{out} + K_2 \cdot \phi_{err}$
    (The current estimate of Charlie's watch tick ra
- There are two variables that Lucy needs to control – a second-order system!

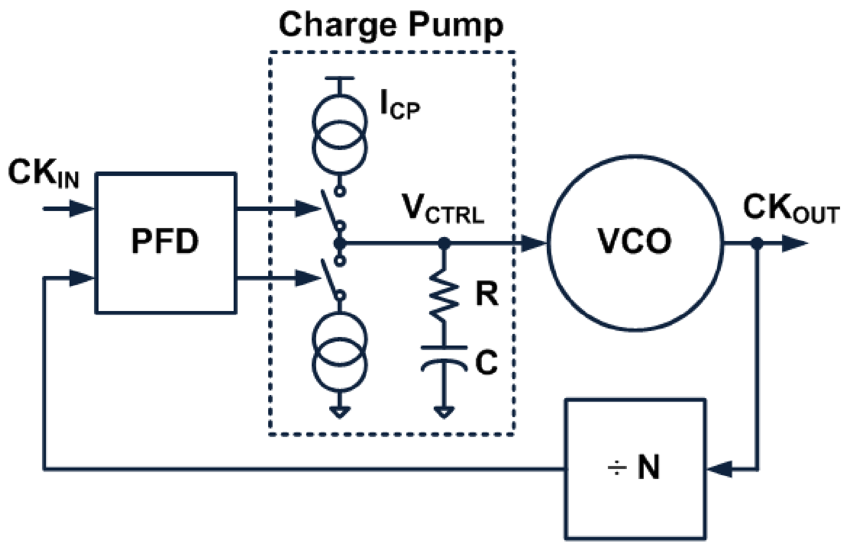
Charge-Pump PLL

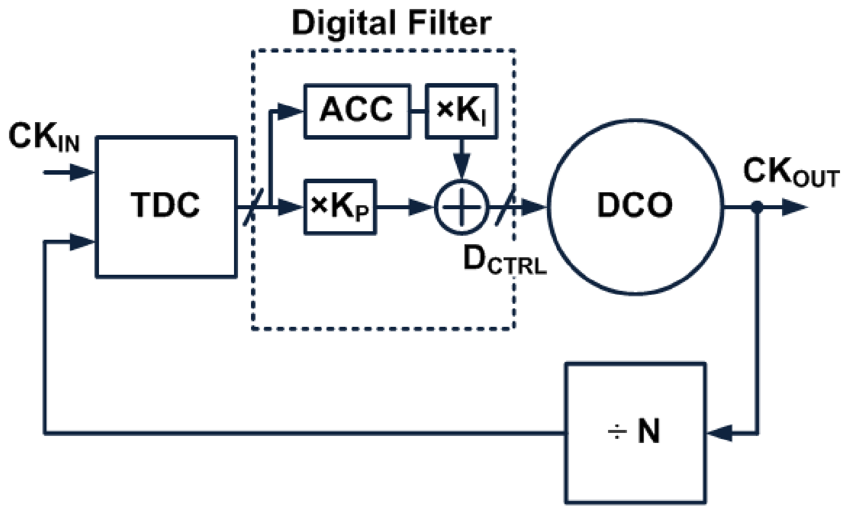


- Uses a charge pump followed by an RC filter as the loop filter (controller)

Digital PLLs

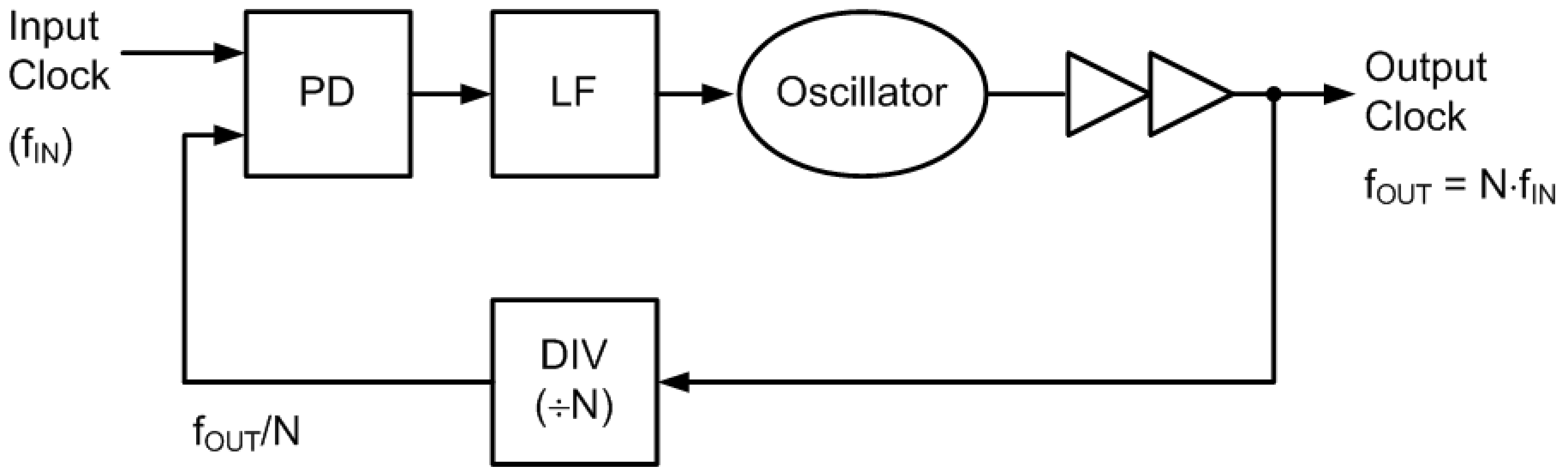
- Use accumulators instead of charge pumps to:
  - Overcome technology limitations such as CP current mismatch, capacitor leakage, design difficulty, etc.
  - Implement transfer functions, calibrations, fast-settling algorithms that are difficult with analog





Frequency Multiplication with PLL

- By inserting a divider (/N), the PLL will lock when the output frequency is N times the input frequency

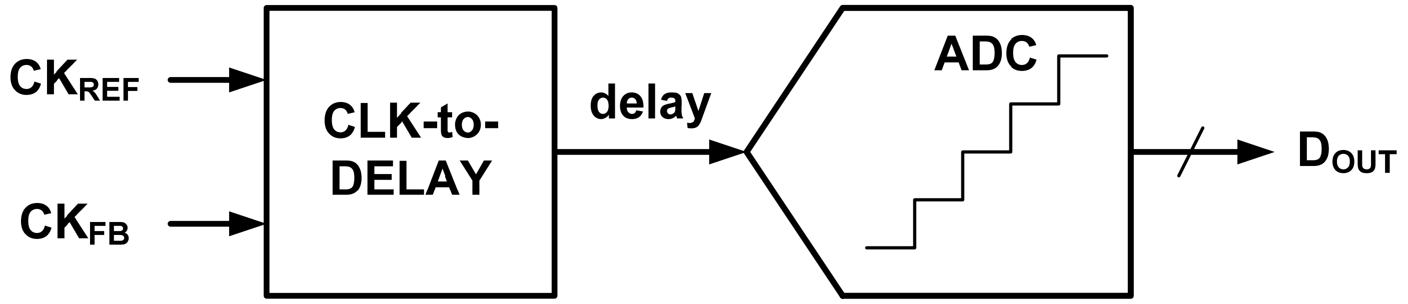


PLL Components

- Just with any feedback loop, a PLL comprises of:
- **Producer (DCO):** generates the output clock with desired phase/frequency
- **Sensor (TDC):** measures the error between the output and the reference input
- **Loop filter** or **Controller:** decides how to adjust the control input of the producer based on the measured error

Time-to-Digital Converter (TDC)

- Converts the timing difference between the two input clocks into a digital code
- The ideal TDC can be constructed using the *clk_to_delay* and *adc* primitives



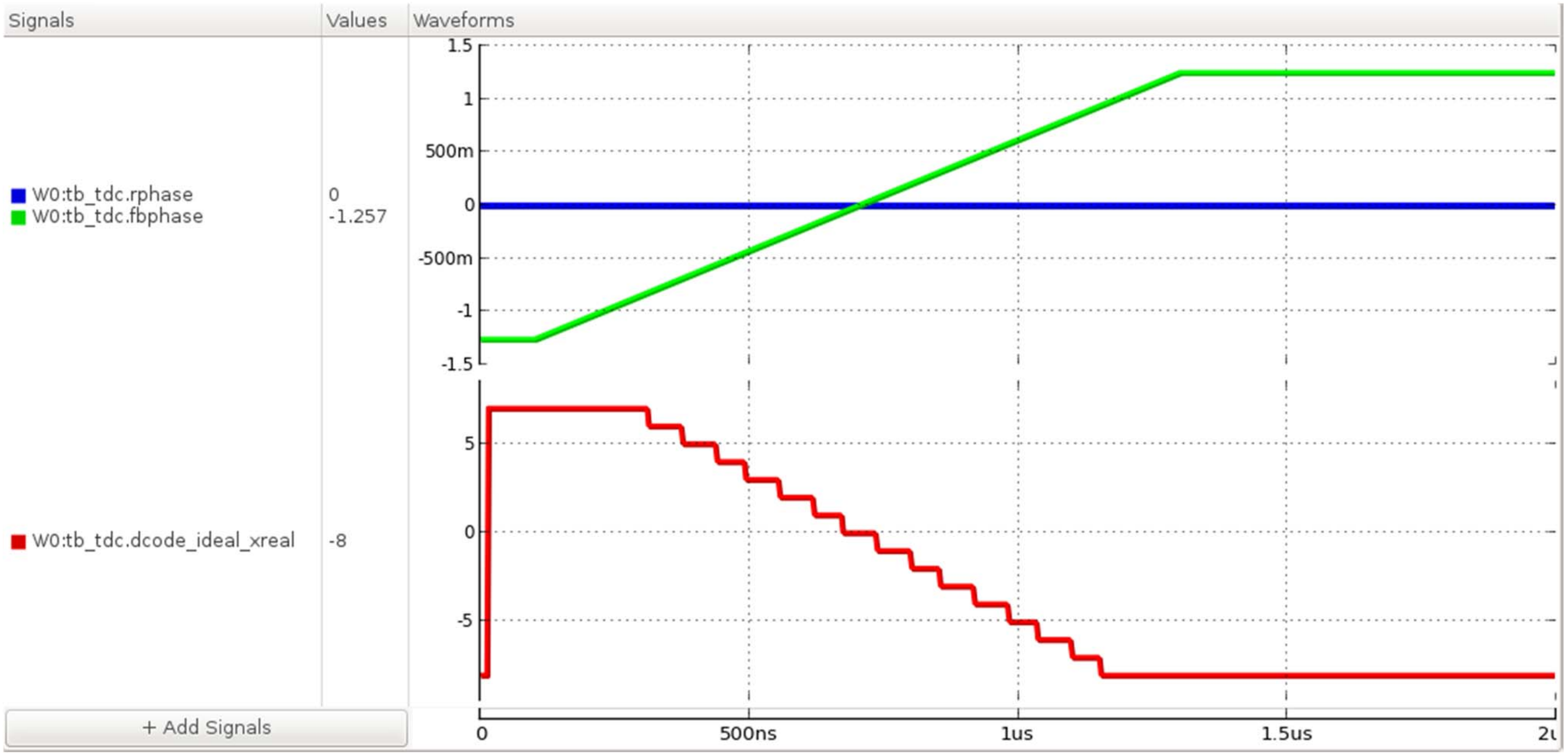
Ideal TDC Model

- models/tdc_ideal.sv:

```
…
clk_to_delay clk2delay(.trig(clk_ref), .in(clk_fb), .out(delay));

adc    #(.num_bit(4), .value_min(-8*unit_delay),
         .value_lsb(unit_delay))
       delay2dig(.in(delay), .out(dout_xb));

xbit_to_bit    #(.width(4)) conn(.in(dout_xb), .out(dout));
assign out = dout - 4'b1000;
…
```

Exercise: TDC Simulation

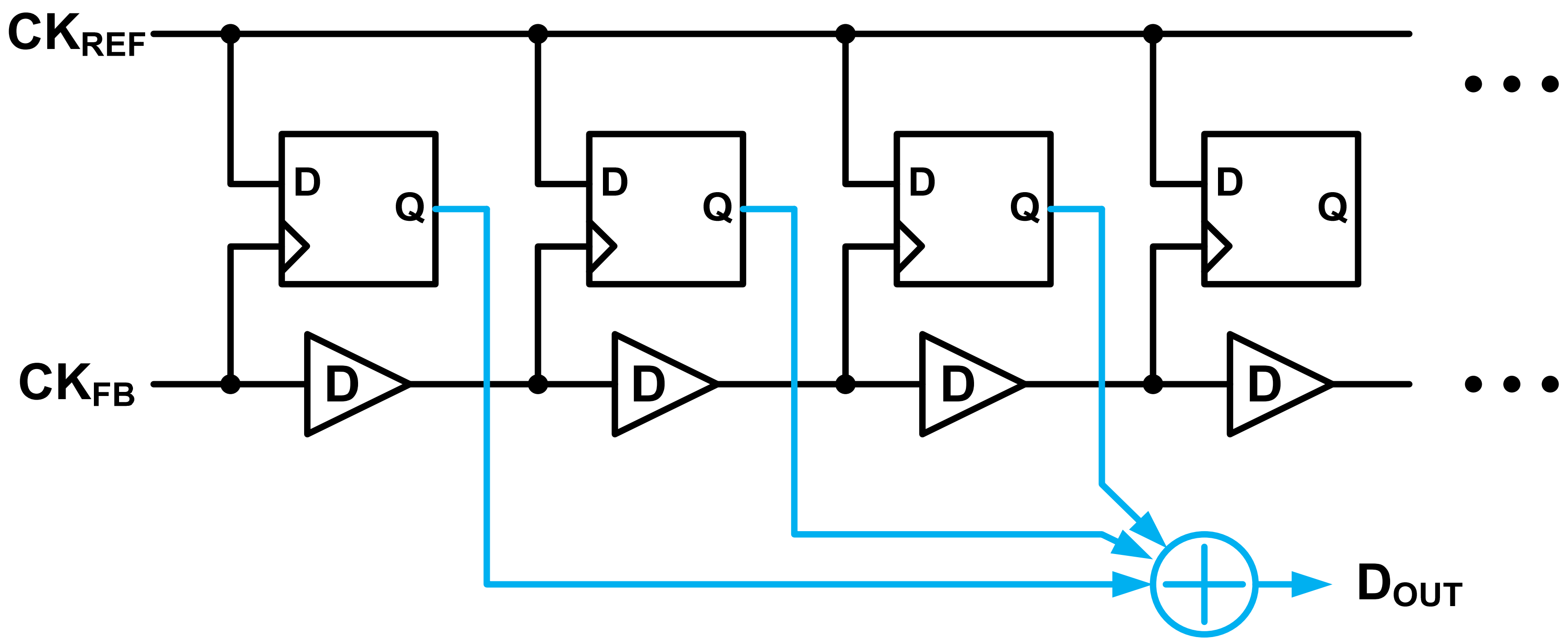
- Run the testbench in sim/tb_tdc_ideal:



What are the gain and resolution of this TDC?

Oversampling TDC

- Digitize the timing difference by oversampling it
  - Multiphase clocks are obtained using a delay line
  - Hence the TDC resolution is set by the buffer delay D



Oversampling TDC Model

- models/tdc_dline.sv:

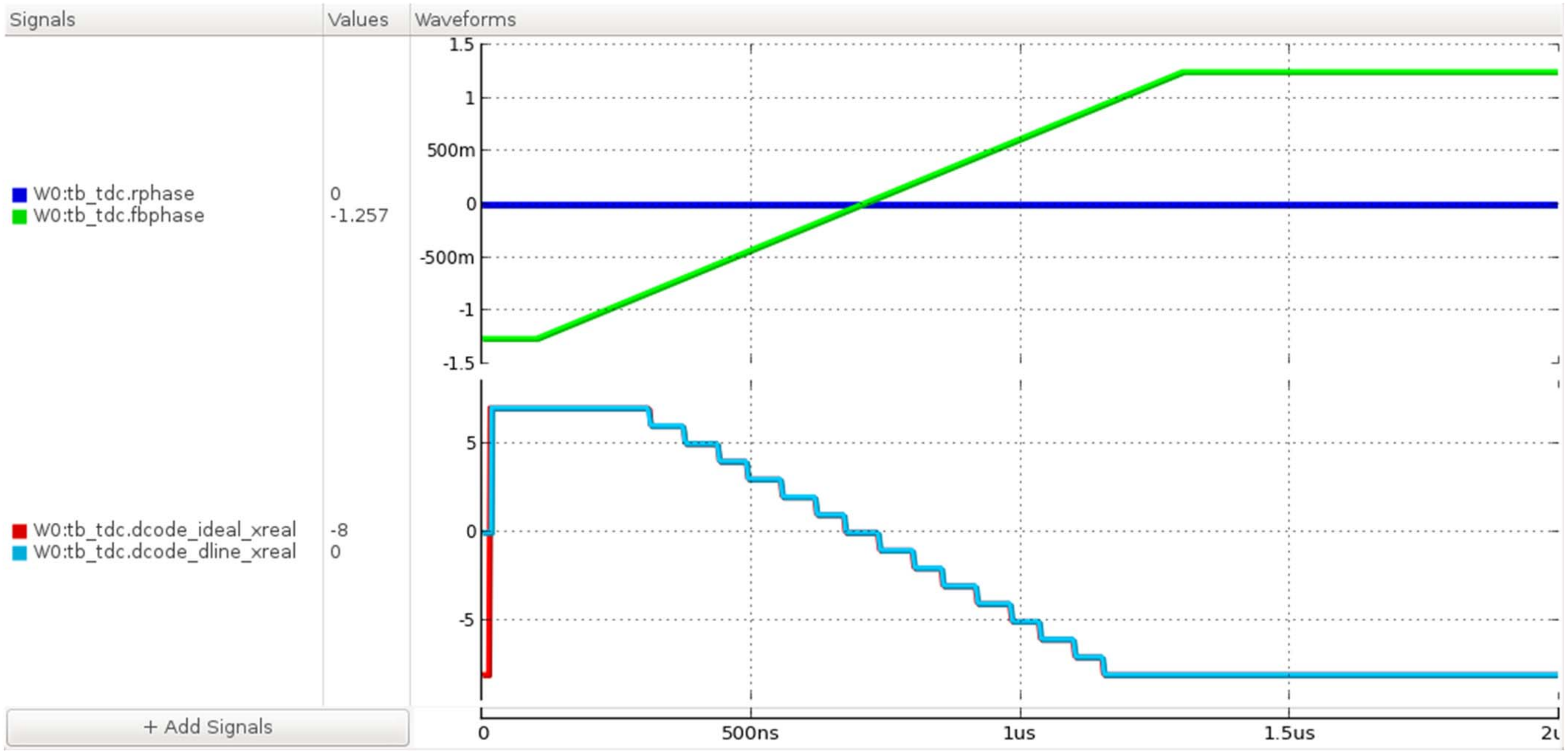
```
// delay line
delay_xbit     #(.delay(0.0))
               delay_fb_0(.in(clk_fb), .out(fbclk[0]));
delay_xbit     #(.delay(unit_delay))
               delay_fb_1(.in(fbclk[0]), .out(fbclk[1]));
…

// phase detectors
dff_xbit       sampler_0(.d(rclk), .q(pd[0]), .clk(fbclk[0]));
dff_xbit       sampler_1(.d(rclk), .q(pd[1]), .clk(fbclk[1]));
…
```

Exercise

- Finish the encoding logic in tdc_dline.sv so that the TDC has the same characteristics with the ideal TDC

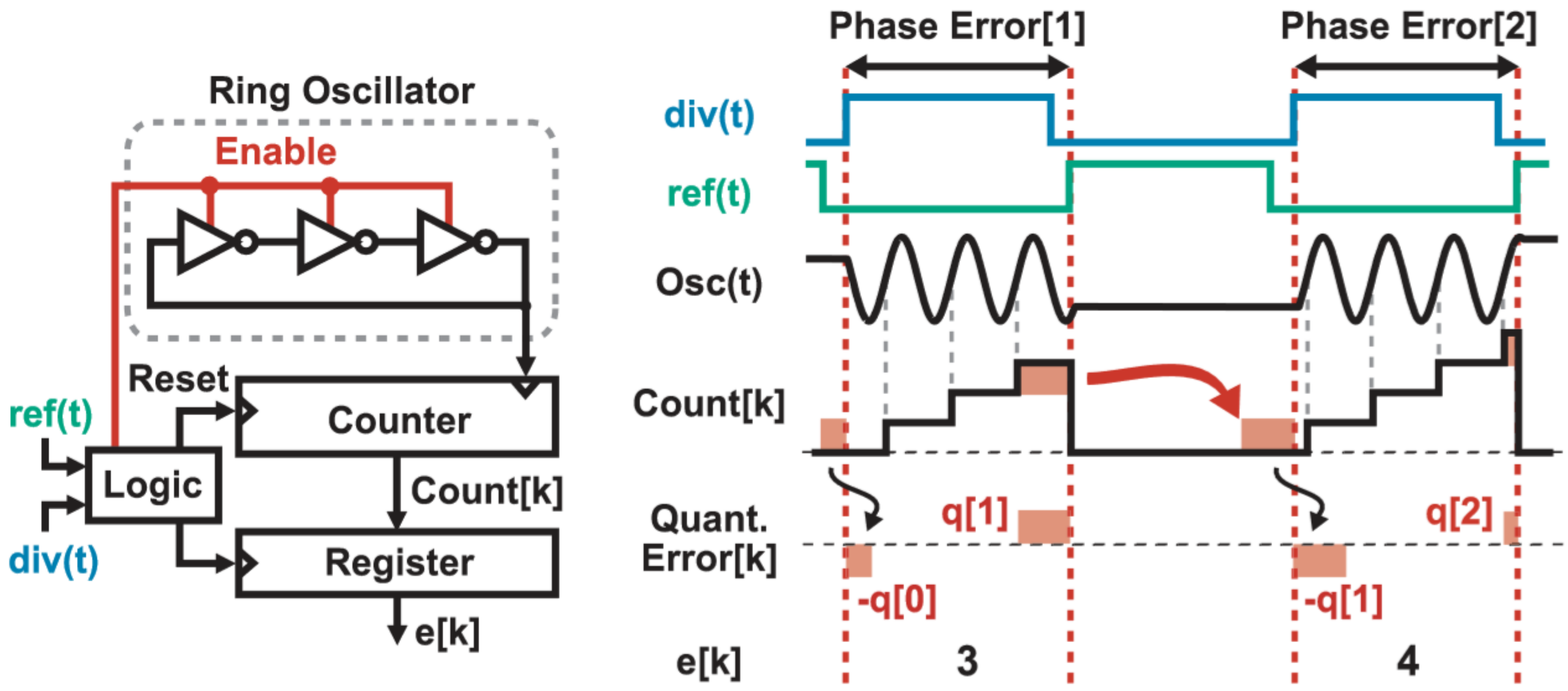
```
always @(pd_bit) begin
    if (pd_bit == 16'b0000_0000_0000_0000)
        out = 4'b1000;
    else if (pd_bit == 16'b1111_1111_1111_1111)
        out = 4'b0111;
    else
      // fill the logic expression to compute the output here
      out = ...;

end
```

Results


Signals
Values
Waveforms
🟦 W0:tb_tdc.rphase
🟩 W0:tb_tdc.fbphase
0
-1.257
🟥 W0:tb_tdc.dcode_ideal_xreal
🟦 W0:tb_tdc.dcode_dline_xreal
-8
0
+ Add Signals
1.5
1
500m
0
-500m
-1
-1.5
5
0
-5
0
500ns
1us
1.5us
2u

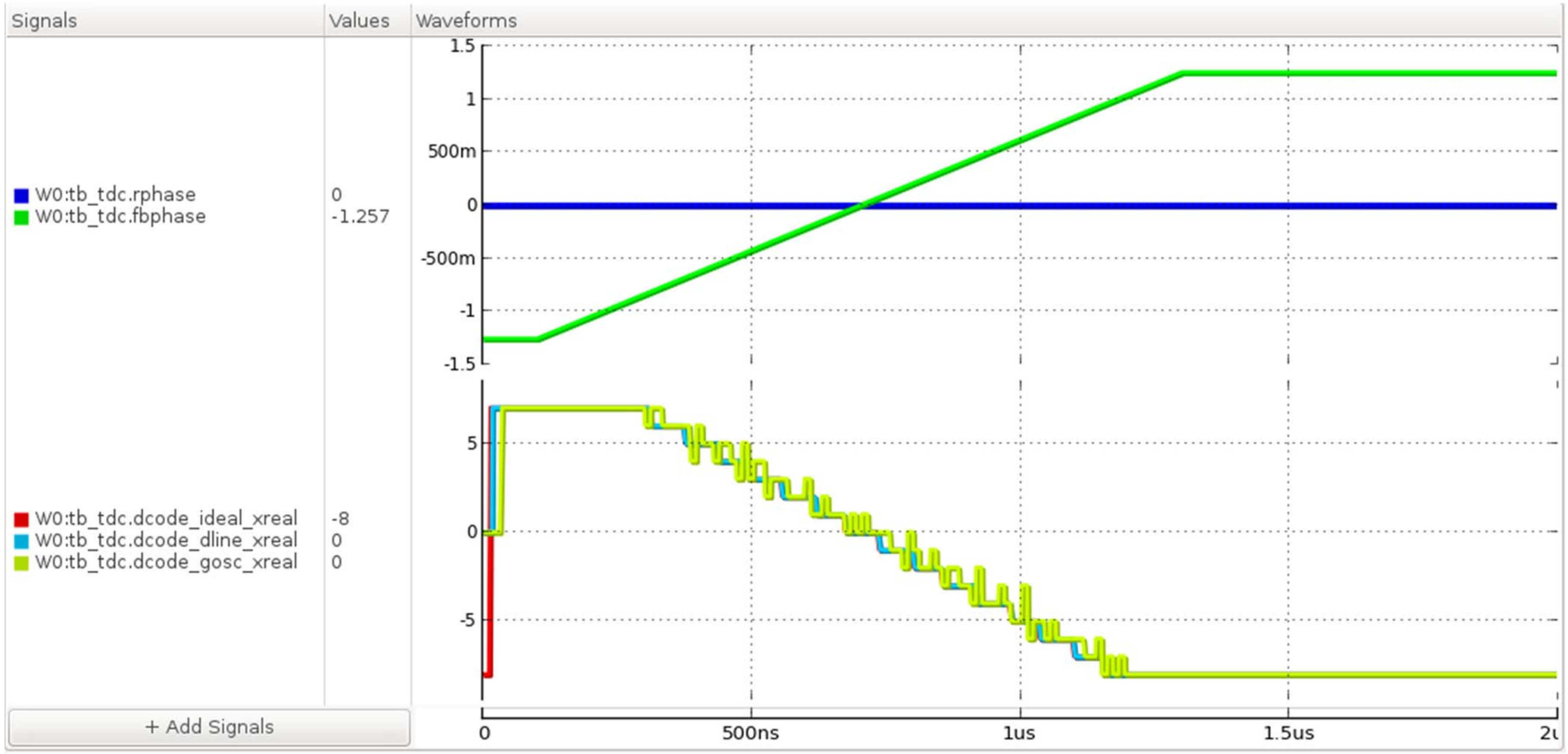
Gated Oscillator TDC

- Measure the pulsewidth of the PFD output by counting the frequency of the oscillator enabled only while the UP/DN pulse is high [M. Perrott]



Exercise

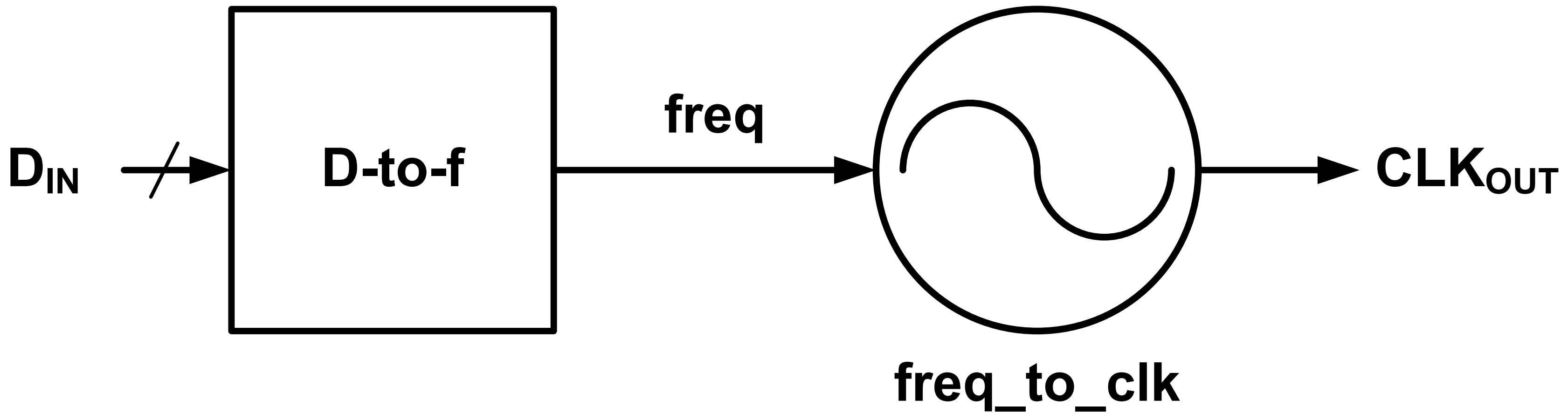
- Compare the characteristics of all three TDCs
  - Run the testbench in sim/tb_tdc_all



Signals	Values
🟦 W0:tb_tdc.rphase	0
🟩 W0:tb_tdc.fbphase	-1.257
🟥 W0:tb_tdc.dcode_ideal_xreal	-8
🟦 W0:tb_tdc.dcode_dline_xreal	0
🟩 W0:tb_tdc.dcode_gosc_xreal	0

Digitally-Controlled Oscillator (DCO)

- Produces a clock whose frequency is controlled by a digital input
  - Can be easily modeled using a ***freq_to_clk*** primitive



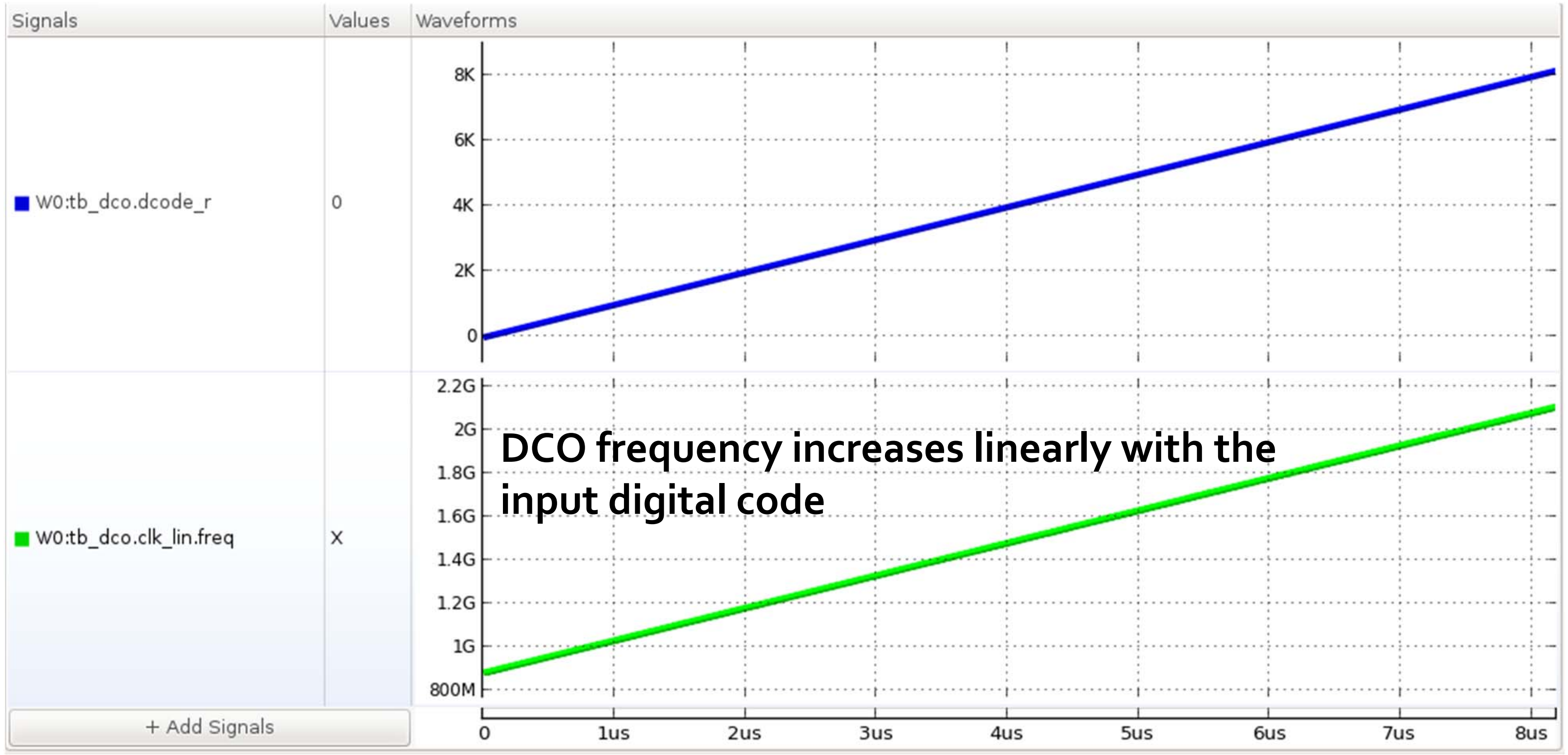
DCO with Linear D-to-f

- models/dco_lin.sv

```
// digital-to-frequency conversion (linear)
always @(in) begin
        freq = fcenter + Kdco*real'(in - `pow(2, Nbit-1));
end
real_to_xreal conn(.in(freq), .out(freq_xr));

// frequency-to-clock conversion
freq_to_clk #(.num_phase(1), .PN_foffset(PN_foffset),
              .PN_fcenter(PN_fcenter), .PN_dbc(PN_dbc))
            freq2clk(.out(out), .in(freq_xr));
…
```

Exercise

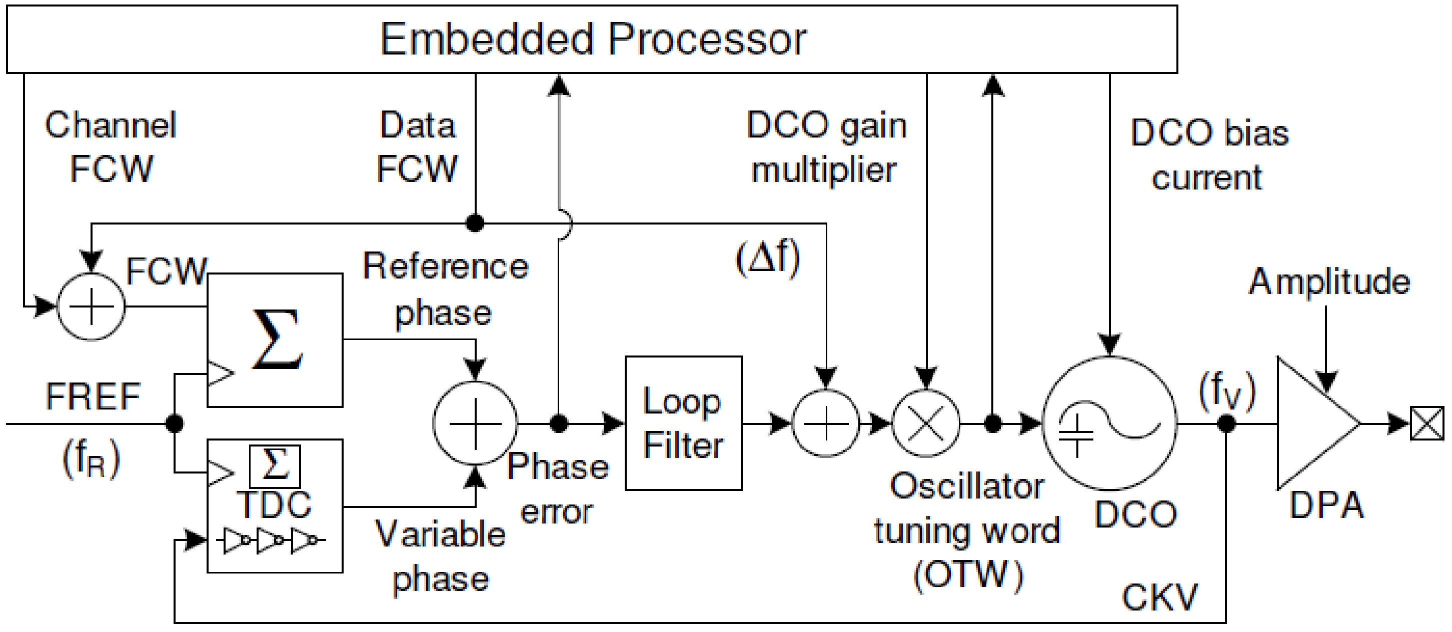
- Plot the D-to-f characteristic of the DCO
  - sim/tb_dco_lin



DCO frequency increases linearly with the input digital code

One Caveat with Digital PLLs

- PVT dependencies of DCO and TDC characteristics often require calibration circuits and routines
  - Complexity may become much higher than CP PLLs



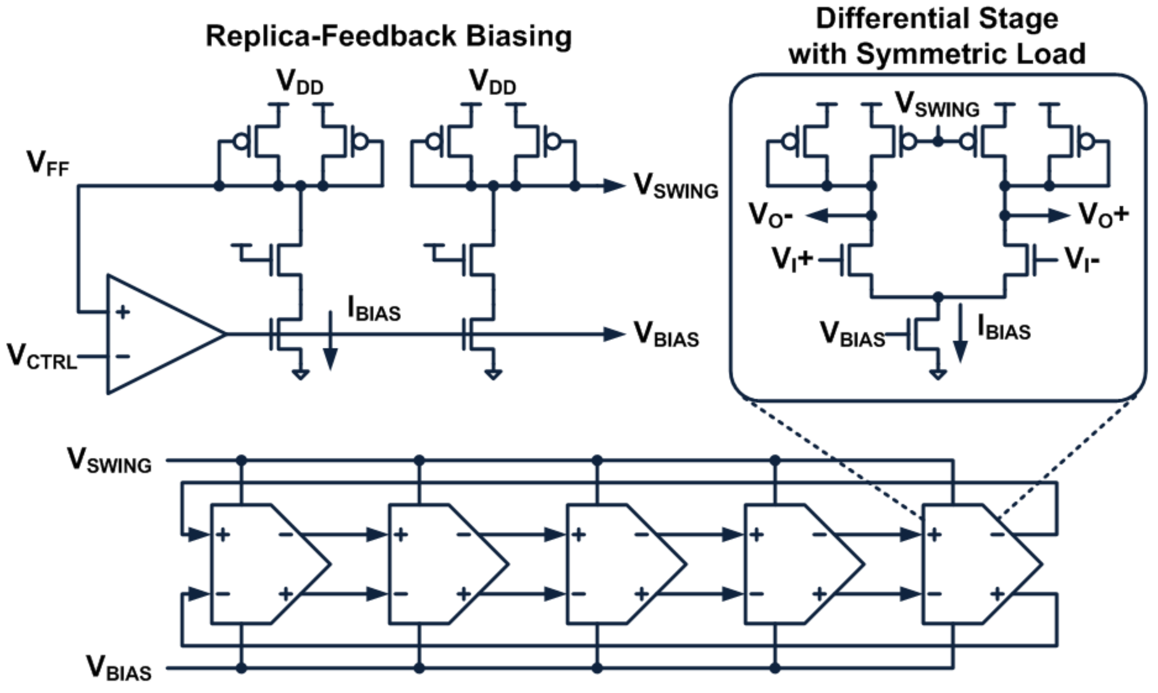
[Staszewski10]

Self-Biased (Adaptive-BW) PLLs

- In analog PLLs, a common way to address PVT is to scale bandwidth with frequency [Maneatis 96]
  - Input frequency is the most reliable to set the BW with

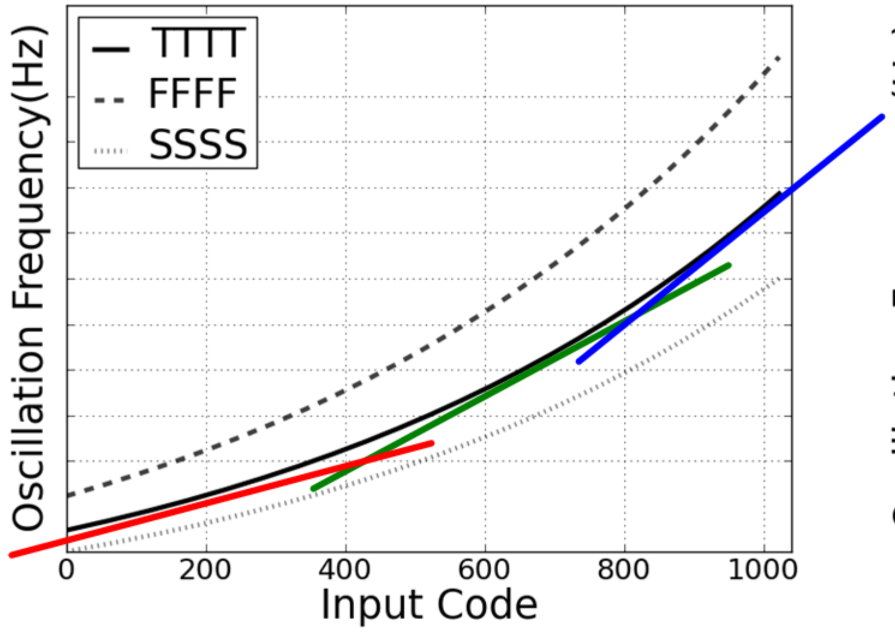
$$I_{CP} \propto I_{VCO}$$
$$R_{CP} \propto R_{VCO}$$

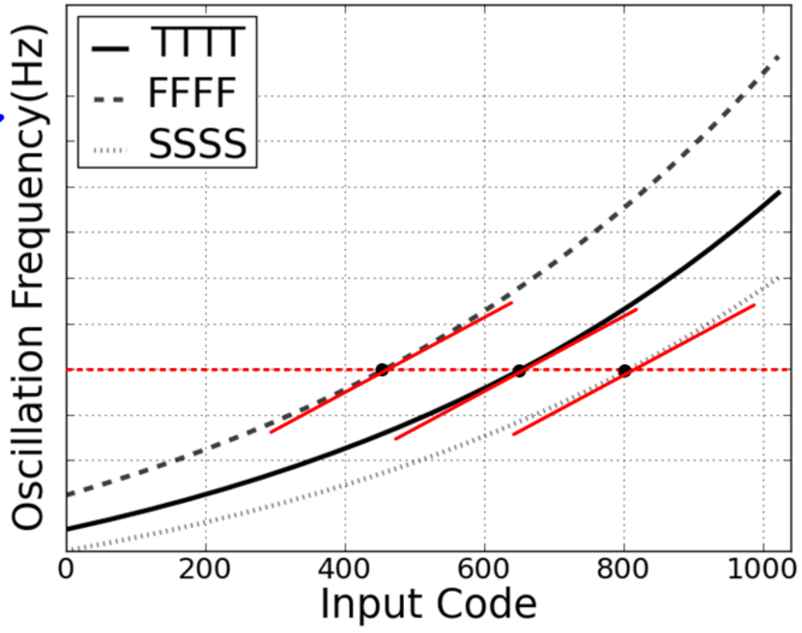
$$BW/f_{REF} \sim \text{const.}$$
$$\zeta \sim \text{const.}$$



Adaptive-Bandwidth Digital PLLs

- Equivalent adaptive-BW can be achieved by realizing an ***exponential D-f curve*** for the DCO
  - K_{DCO} scaling with the frequency (left)
  - K_{DCO} constant over PVT at a given frequency (right)





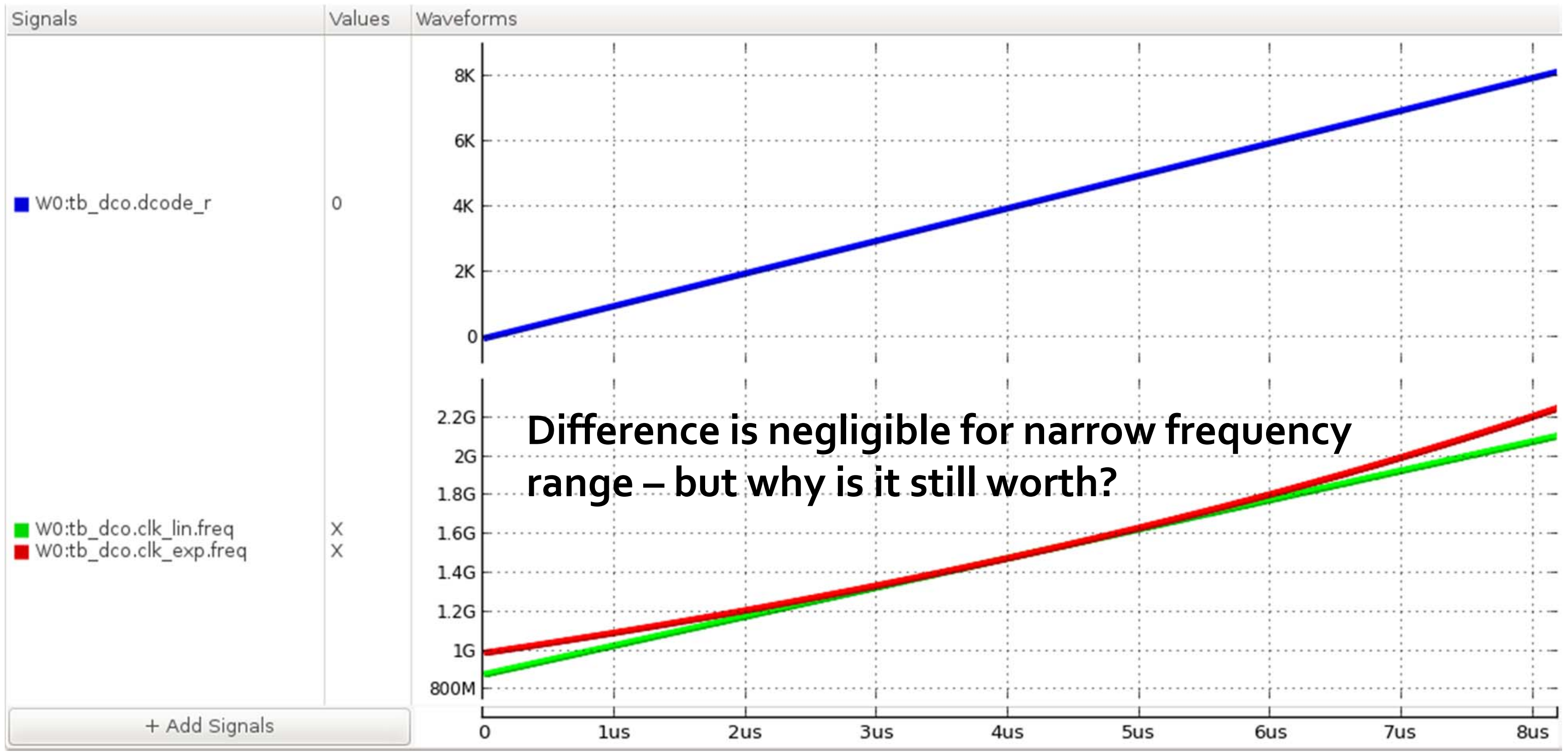
Exercise

- Fill the blank in dco_exp.sv so that the DCO has:
  - 12-bit resolution (f=5GHz when D = 12'b1000_0000_0000)
  - Relative gain of 0.004%/step

```
// digital-to-frequency conversion (exponential)
always @(in) begin
    // FILL BELOW TO DESCRIBE AN EXPONENTIAL
    // D-to-F CHARACTERISTIC
    // HINT: Use `pow(X, Y) to compute X^Y
    freq = ...;
end

real_to_xreal conn(.in(freq), .out(freq_xr));
```

Results

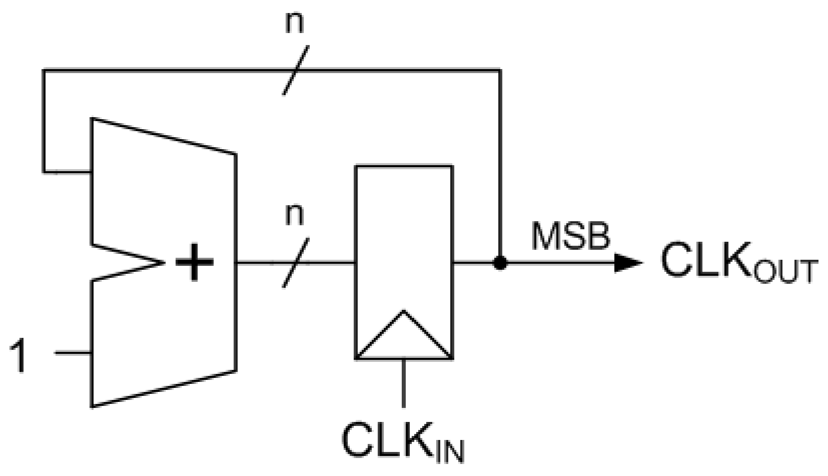
- sim/tb_dco_all:

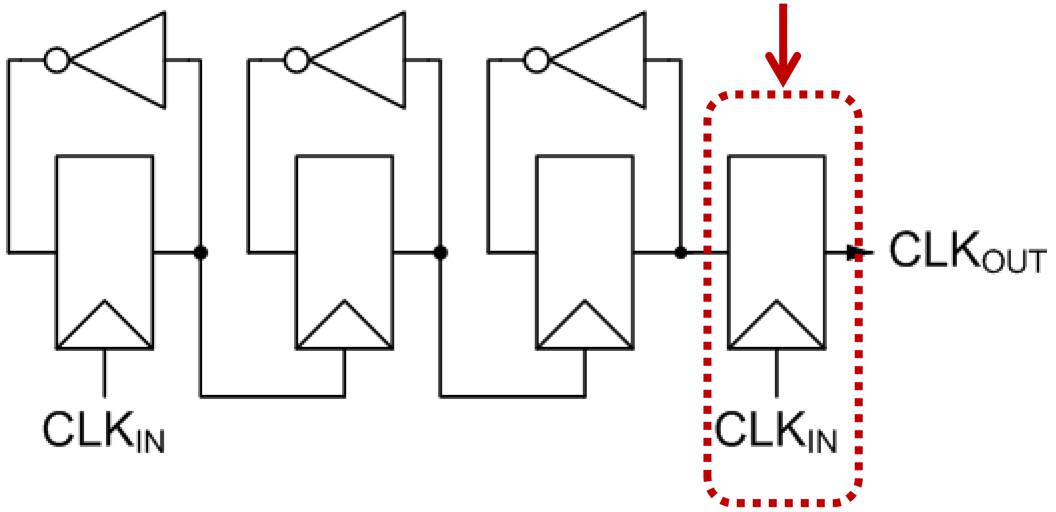


Difference is negligible for narrow frequency range – but why is it still worth?

Frequency Divider

- Synchronous dividers have best jitter but slow
- Asynchronous dividers (e.g. ripple carry counter) are simple but accumulate jitter through the long delay path

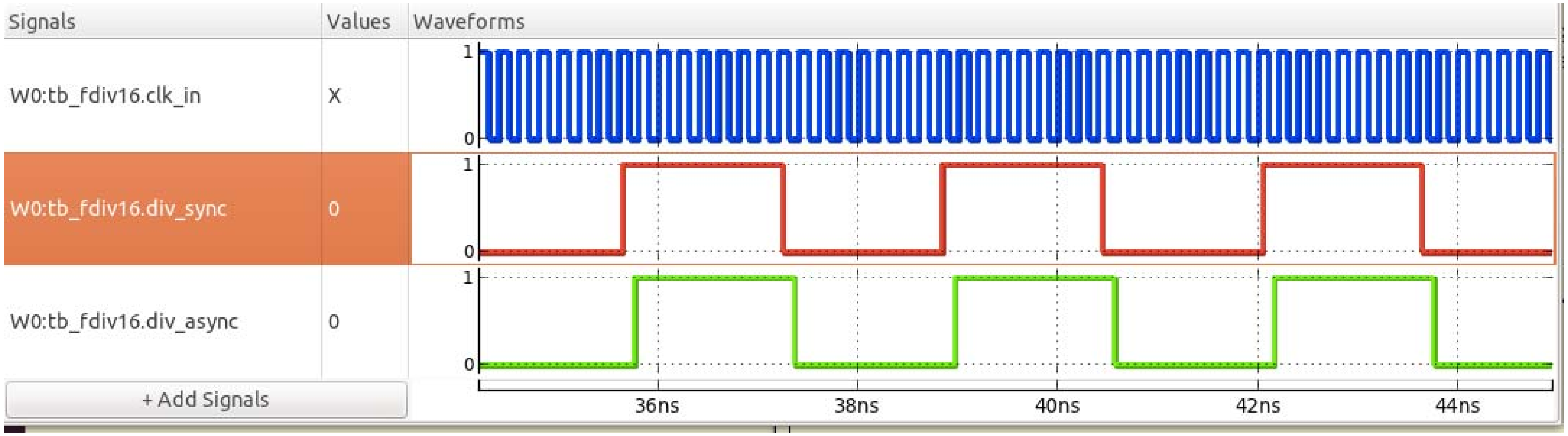




Synchronous Divider **Asynchronous Divider**

Exercise

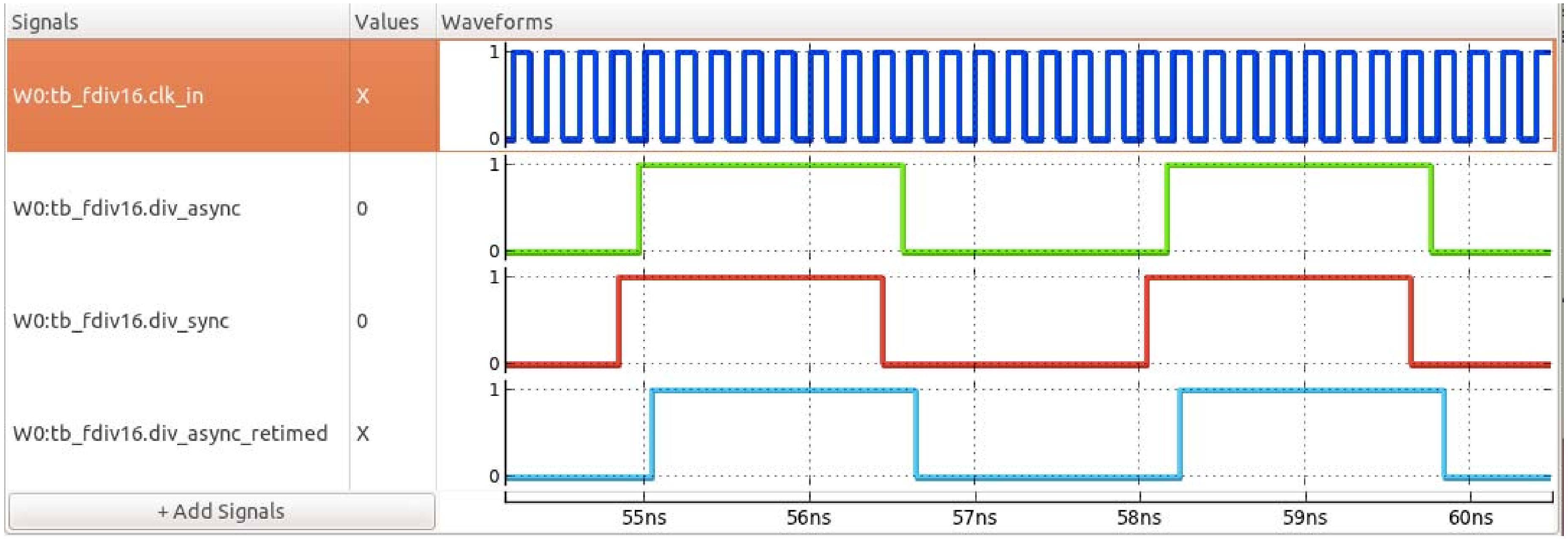
- Examine the two frequency divider models:
  - Synchronous divider: models/fdiv16_sync.sv
  - Asynchronous divider: models/fdiv16_async.sv
- Run the testbench in sim/tb_fdiv16:



What is the difference between the two divider outputs?

Exercise (cont.)

- Modify the asynchronous divider model so that its clock-to-Q delay is reduced
  - Add a retiming FF as shown in the previous slide
  - Answer: sim/tb_fdiv16/answer/fdiv16_async_retimed.sv



Signals
Values
Waveforms
W0:tb_fdiv16.clk_in
X
W0:tb_fdiv16.div_async
0
W0:tb_fdiv16.div_sync
0
W0:tb_fdiv16.div_async_retimed
X
+ Add Signals
55ns
56ns
57ns
58ns
59ns
60ns