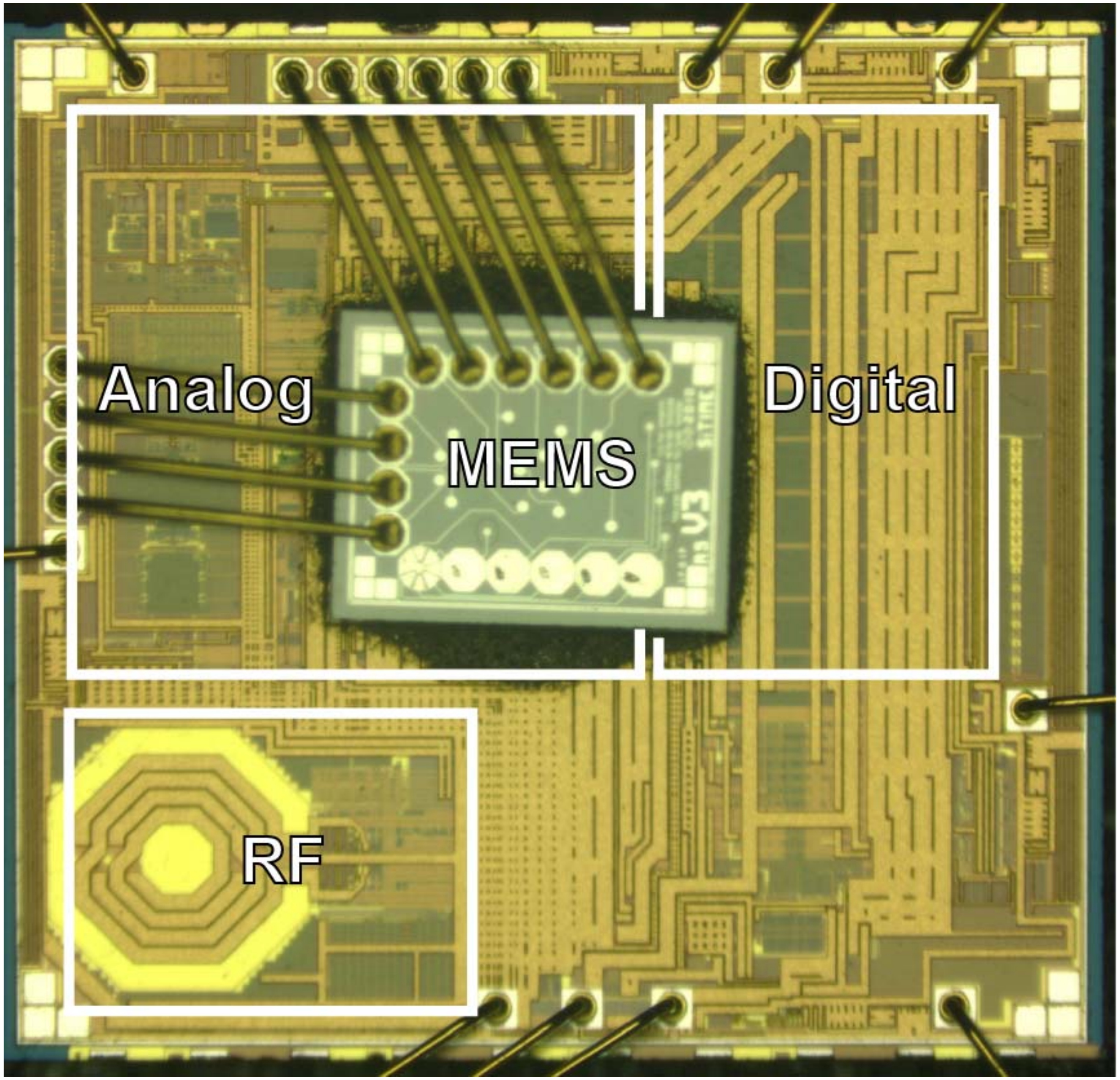
# *Fast and Accurate System Level Simulation of Time-Based Circuits using CppSim and VppSim*

*IEEE Distinguished Lecture*
*Lehigh Valley SSCS Chapter*

**Michael H. Perrott**

**October 2013**

## *Modern Mixed Signal Circuit Design*

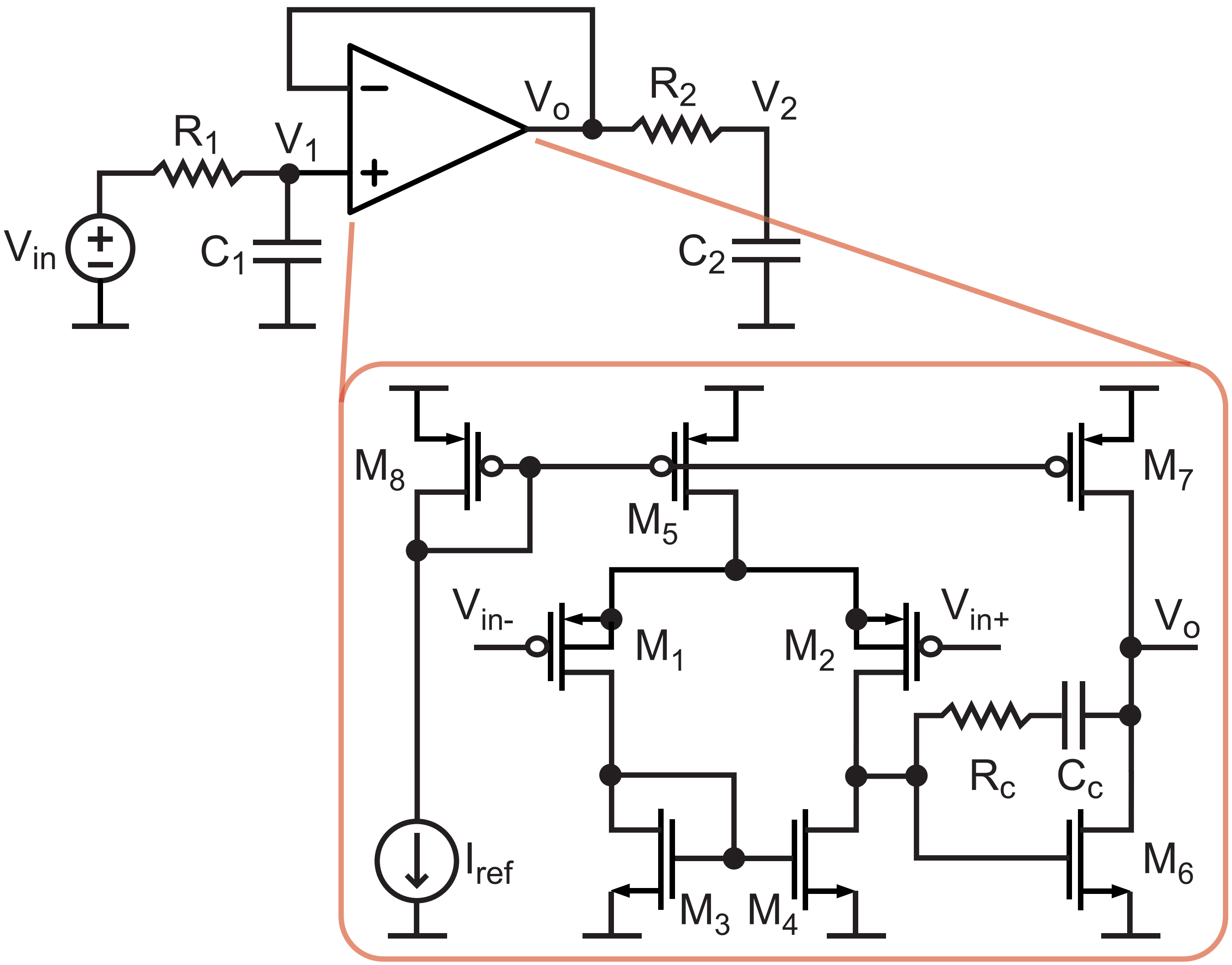



**A Programmable MEMS Oscillator**

- ***Analog***
  **Temperature sensor, ADC, oscillator sustaining circuit**
- ***Digital***
  **signal processing**
- ***RF***
  **clocking (2.5 GHz)**
- ***MEMS***
  **high Q resonator**

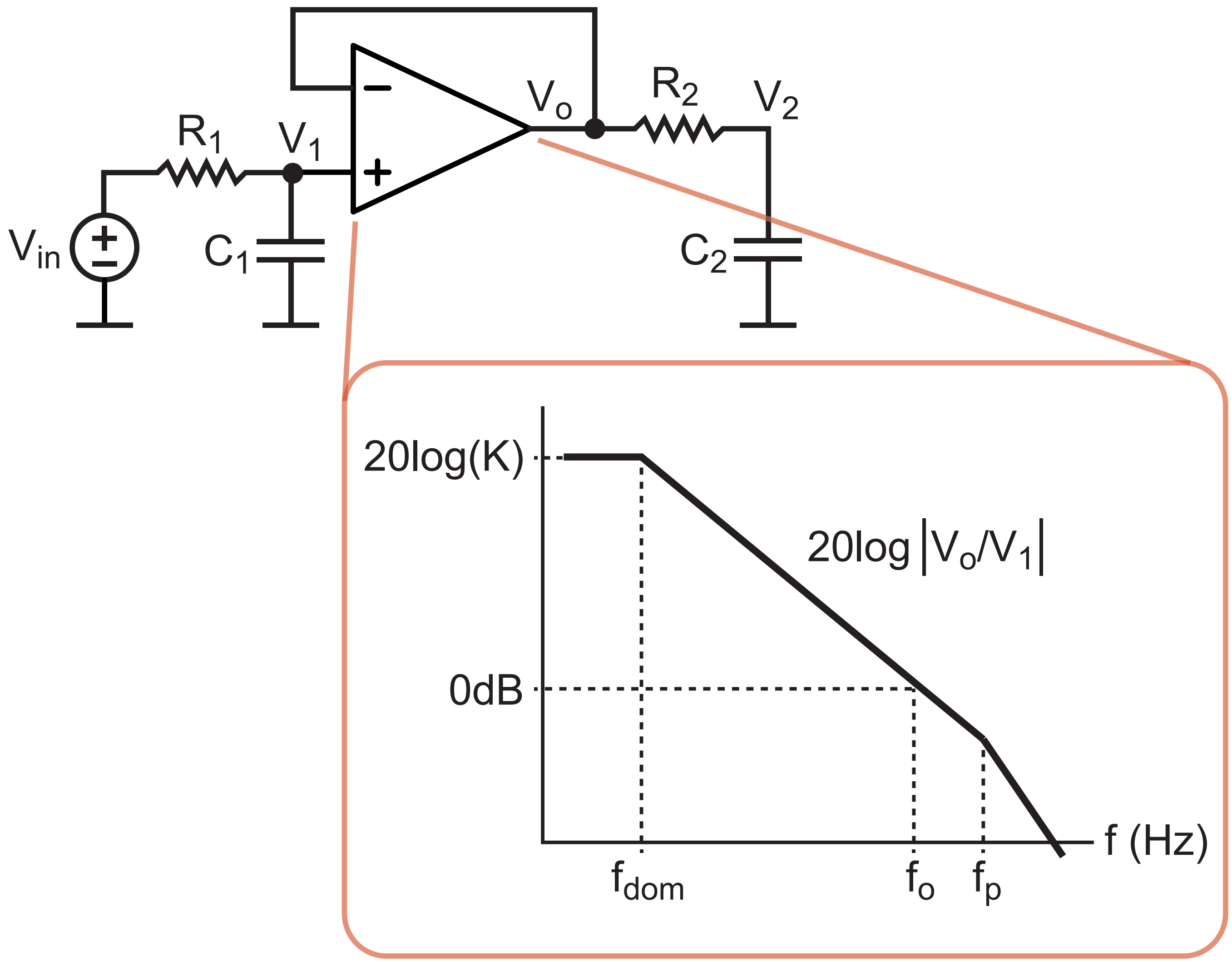
**System level design is critical**

# System-Level Modeling:  A Basic Example



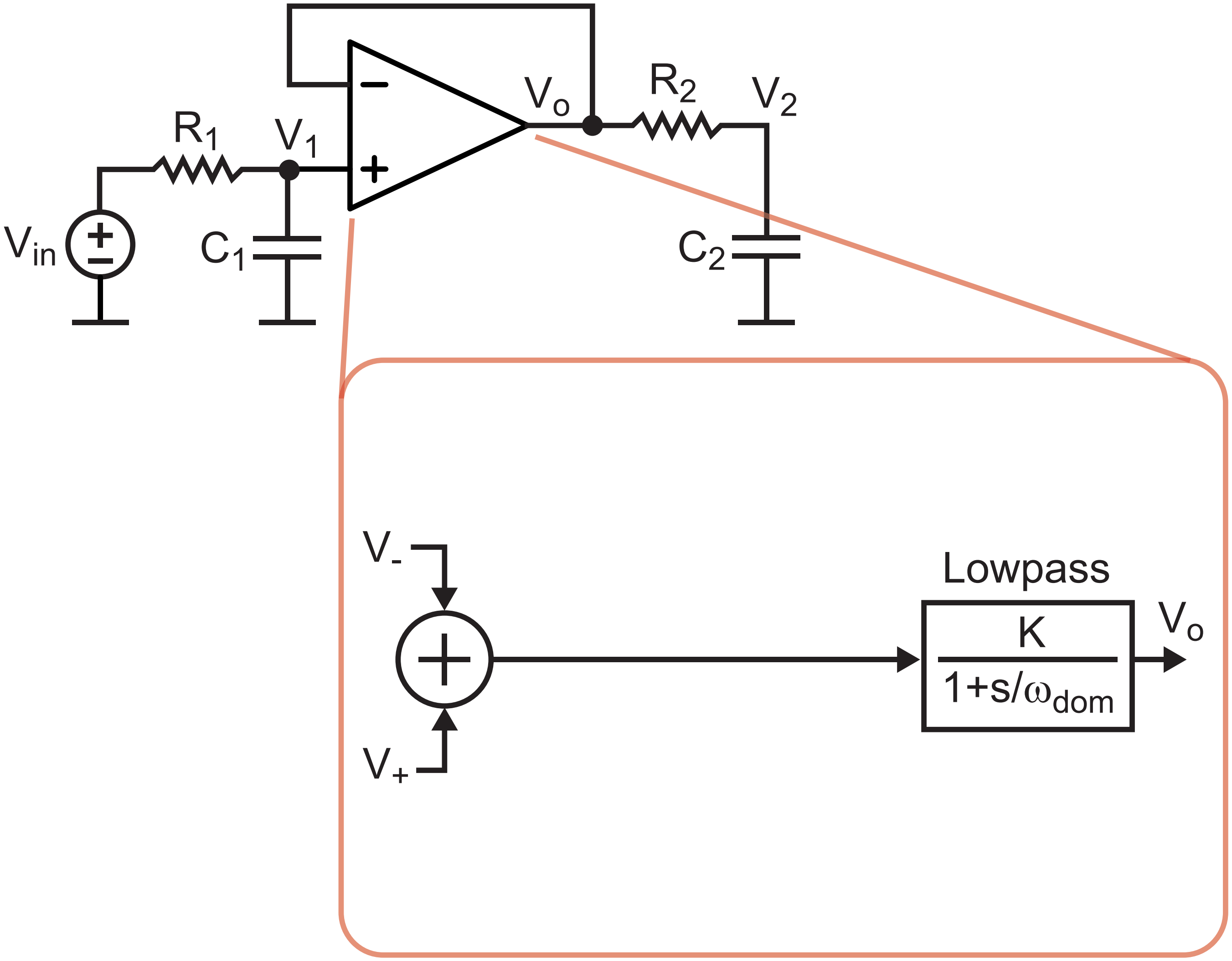

- Opamp is a nonlinear, transistor-level circuit
  - Device level representation mandates SPICE-level simulation

# *Opamps Often Modeled at Transfer Function Level*



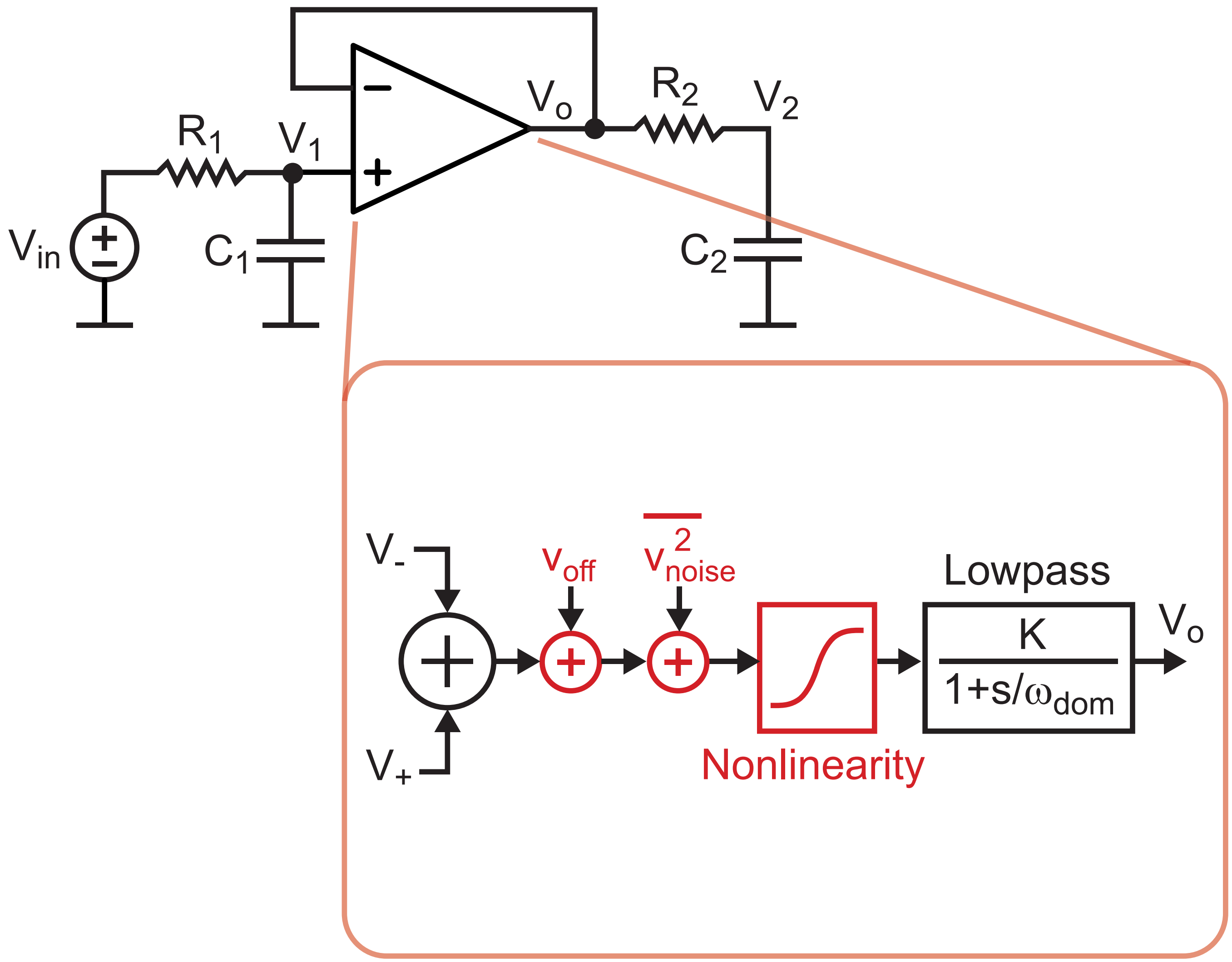

- **Works well for small perturbations about steady-state**
  - **Key parameters are gain and bandwidth**

# A Simple Block Diagram Model of Opamp



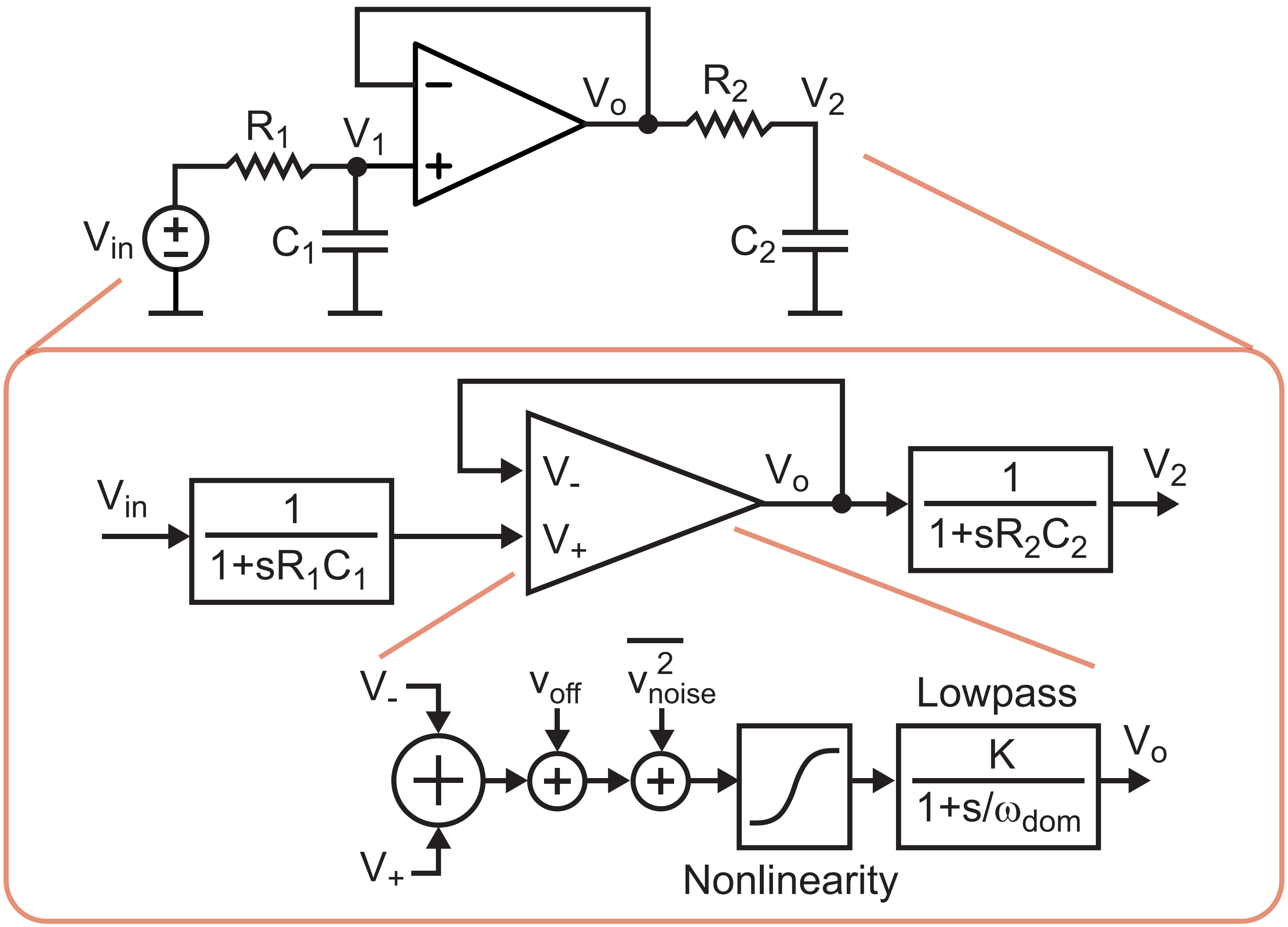

- **Approximates first order behavior of opamp**

# *Inclusion of Second Order Effects*



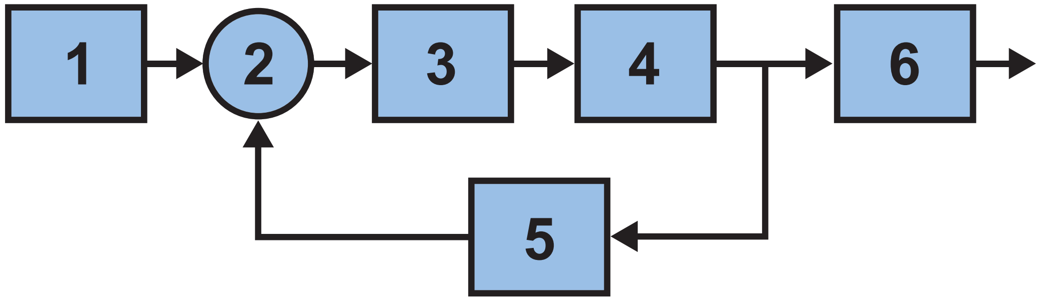

- **Offset, noise, and nonlinearity of front end-differential pair**
  - **Parasitic poles are also easy to add as additional blocks**

# Overall Block Diagram Model



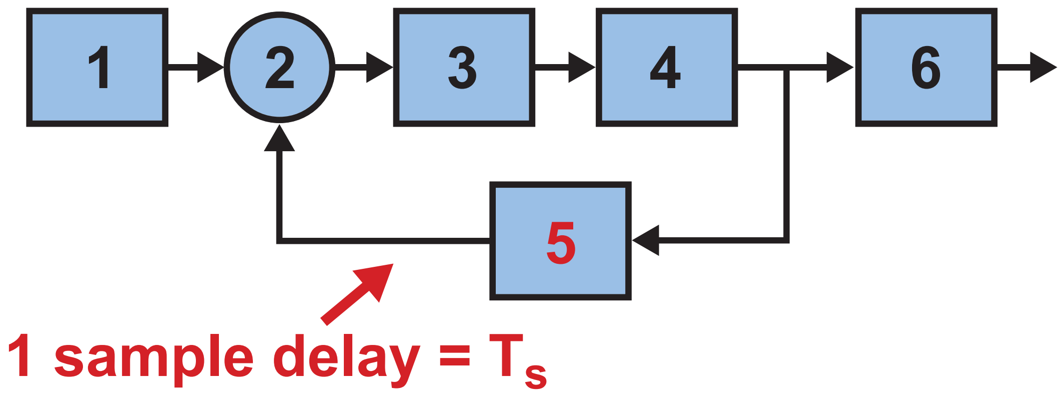

- **Unilateral flow through blocks allows fast simulation**
  - **Compute block outputs one at a time for each time step**

# *Advantages of Block-by-Block Computation*




- **Simple, fast computational structure**
  - **Simply perform computation for each block one at a time for each time step**
    - Extends to hierarchical design quite easily
- **High level of system complexity can be handled**
  - **Overall computational load is simply the sum of the computation required for each block**
  - **Contrast with SPICE whose computational load grows exponentially with the number of elements**

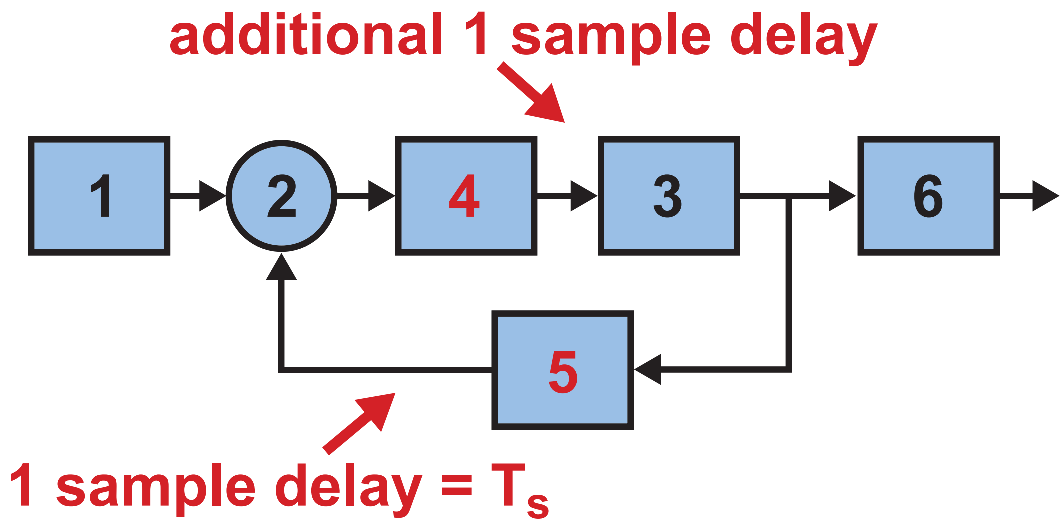
## *The Issue of Delay with Block-by-Block Computation*




- **Minimum possible delay within a feedback loop is one sample period**
  - **Example: Block 2 will not receive updated value from Block 5 until next time sample**
  - **For unity gain crossover frequency $f_o$ and delay $T_s$:**
    - Phase margin reduced by $f_o \cdot T_s \cdot 360^o$

**Time step of simulation must be small compared to bandwidth of feedback loops being simulated**

## *The Issue of Block Order*

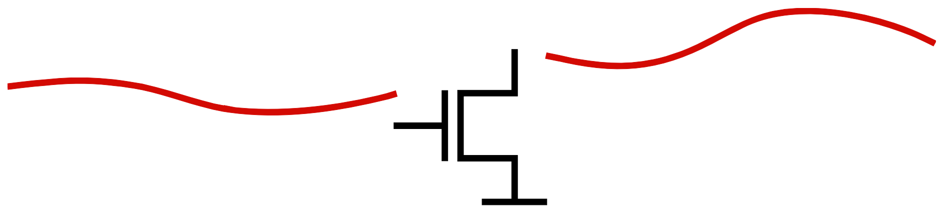



- **Poor ordering of blocks leads to additional delay within feedback loops**
  - **Issue is made worse if blocks computed concurrently**
    - Leads to one sample delay *per block*
- **Block-by-block computation requires additional algorithm to achieve minimum delay ordering**

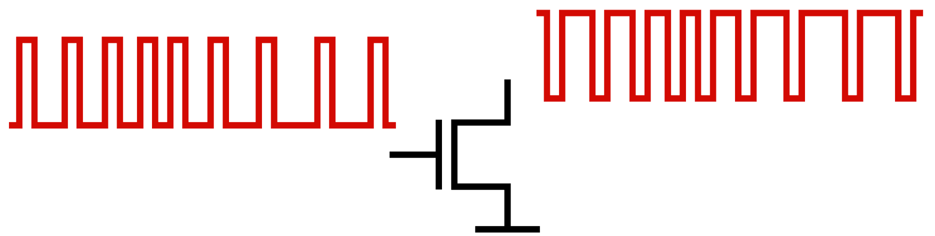
**CppSim provides automatic minimum delay ordering and allows user specified ordering**

# *Time-Based Circuits*

- **Traditional analog circuits utilize voltage and current with bandwidth constrained signaling**



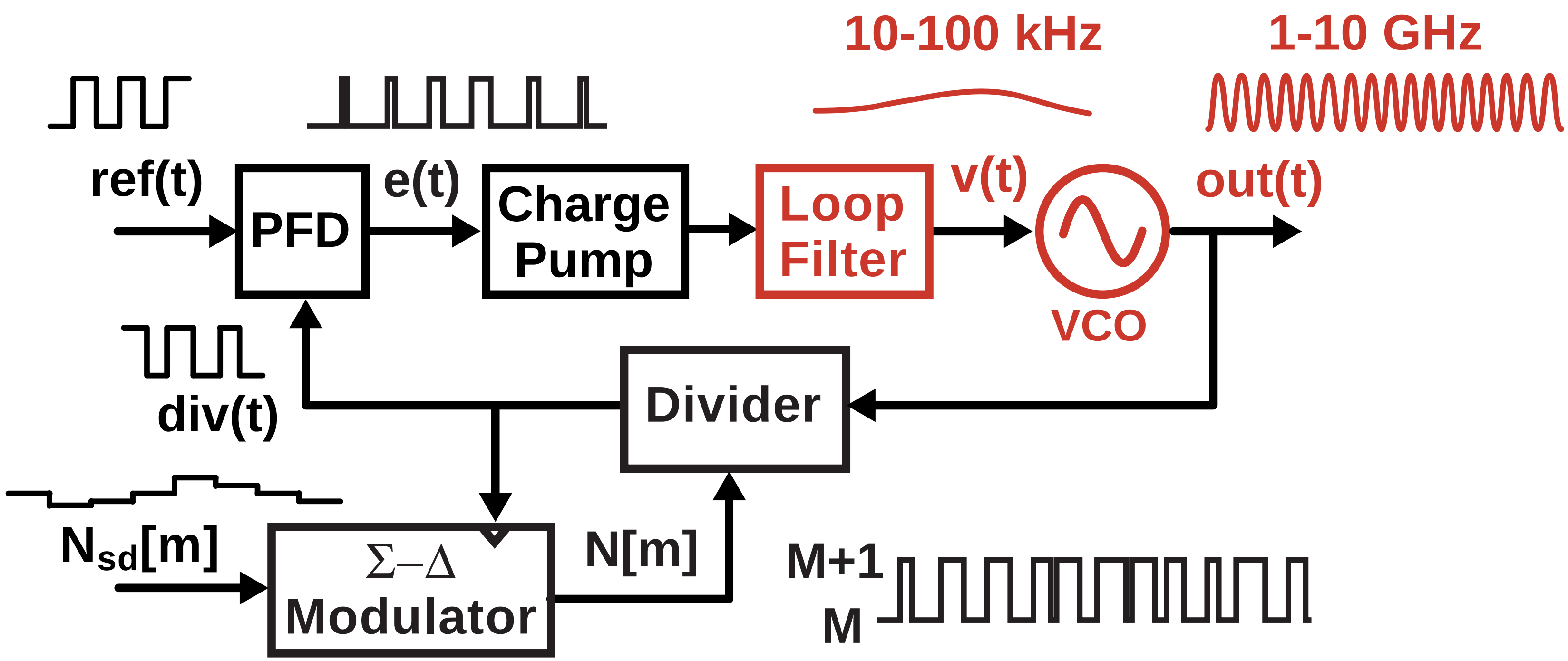
- **Time-based circuits utilize the timing of edges produced by “digital” circuits**



  - **Modern CMOS processes are offering faster edge rates and lower delay through digital circuits**

**High bandwidth of time-based circuits creates challenges for high speed simulation**

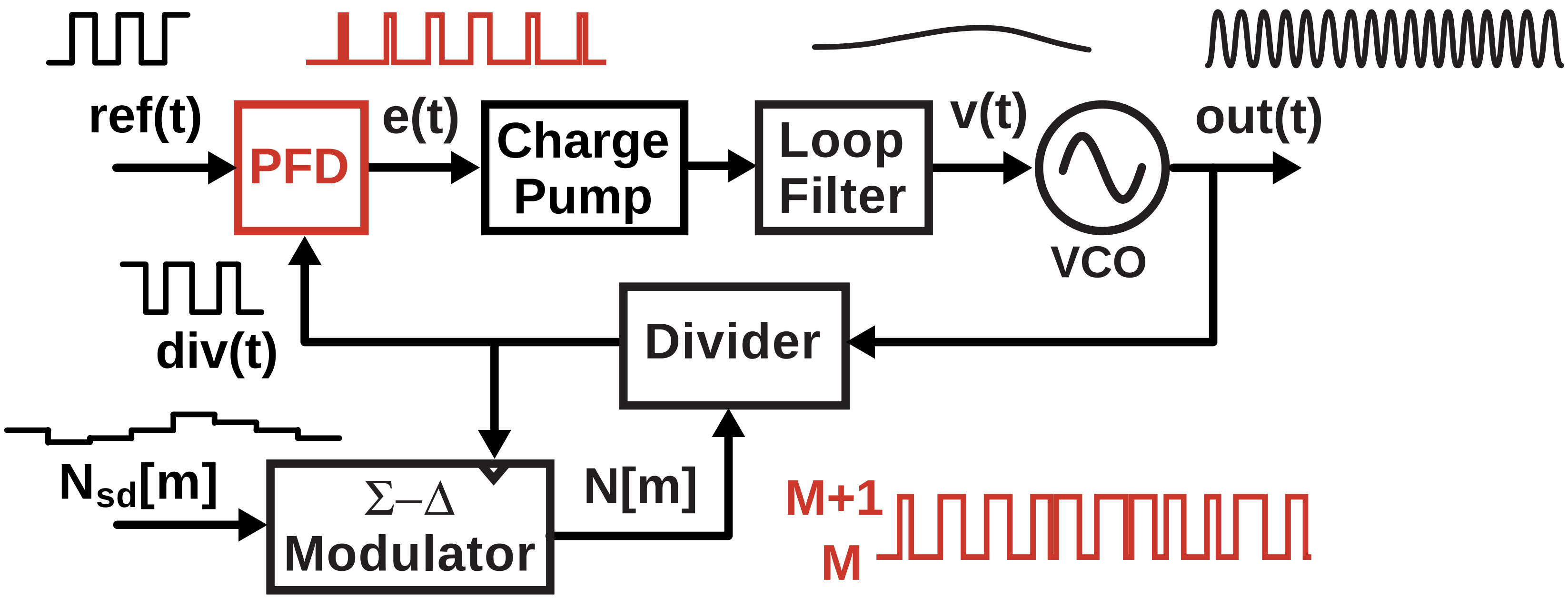
# *A Common Time-Based Circuit*




- Consider a fractional-N synthesizer as a prototypical time-based circuit
  - High output frequency ⇨ High sample rate
  - Long time constants ⇨ Long time span for transients

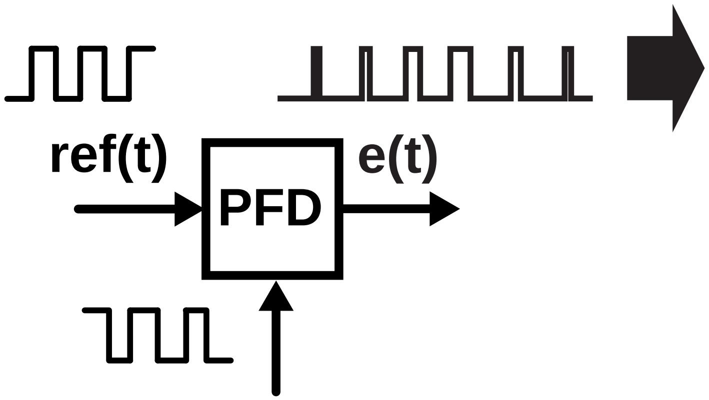
**Large number of simulation time steps required**

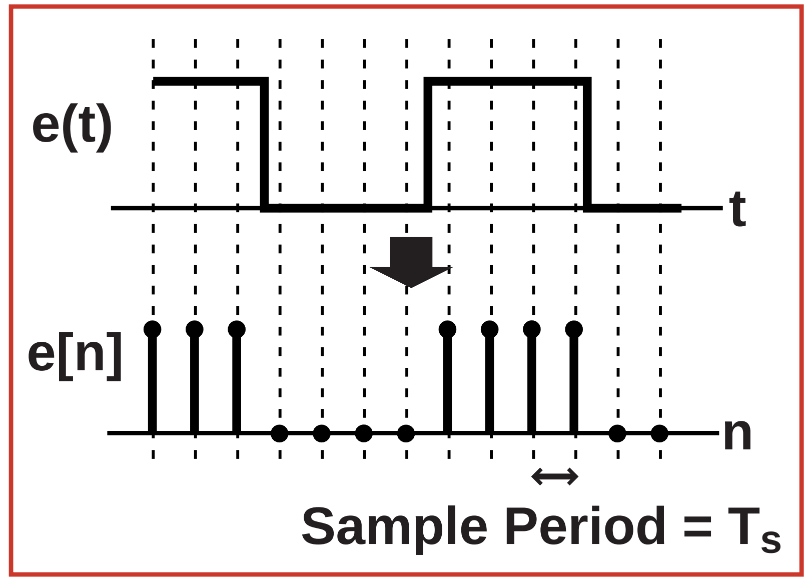
# *Continuously Varying Edges Lead to Accuracy Issues*




- PFD output has very high bandwidth
  - Difficult to achieve high accuracy within a conventional discrete-time or SPICE level simulator
- Non-periodic dithering of divider complicates matters
  - Periodic, steady-state methods do not apply

# *Consider A Classical Constant-Time Step Method*

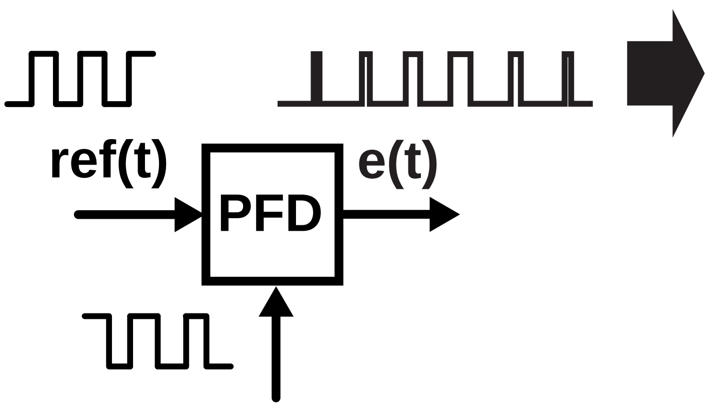


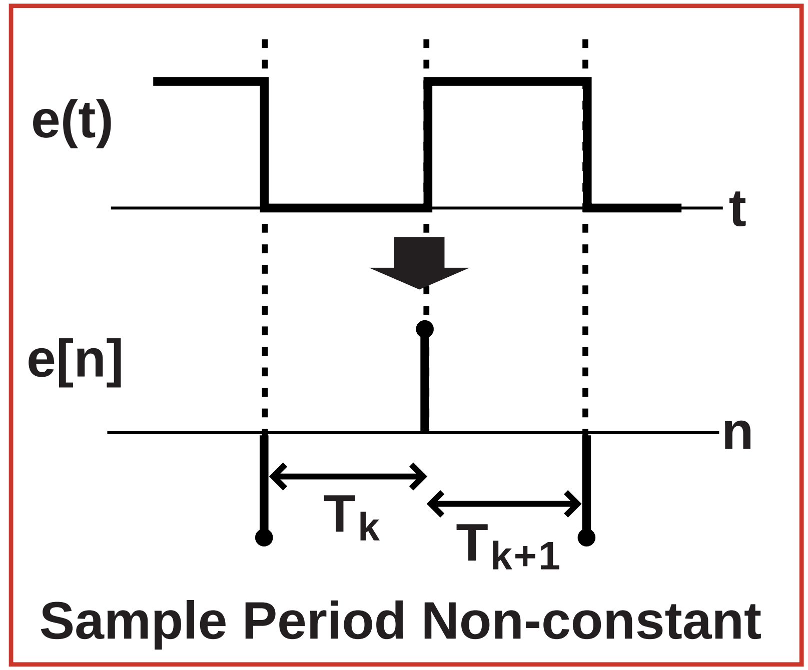




(Johns and Martin,
Analog Integrated Circuit Design)

- Directly sample the PFD output according to the simulation sample period
  - Simple, fast, readily implemented in Matlab, Verilog, C++
- Issue – quantization noise is introduced
  - This noise can overwhelm the PLL noise sources we are trying to simulate

# *Alternative: Event Driven Simulation*

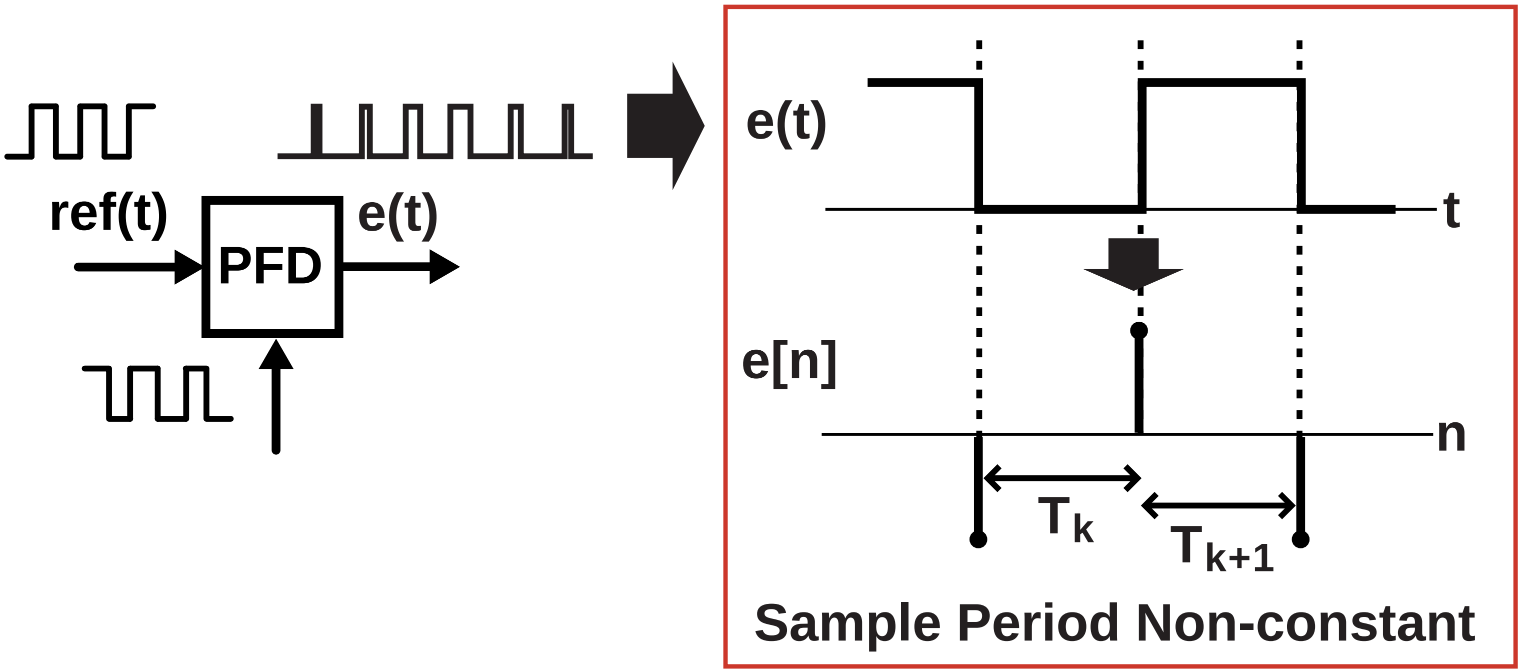



(Smedt et al, CICC ’98,
Demir et al, CICC ’94,
Hinz et al, Circuits and Systems ’00)

- **Set simulation time samples at PFD edges**
  - **Sample rate can be lowered to edge rate!**

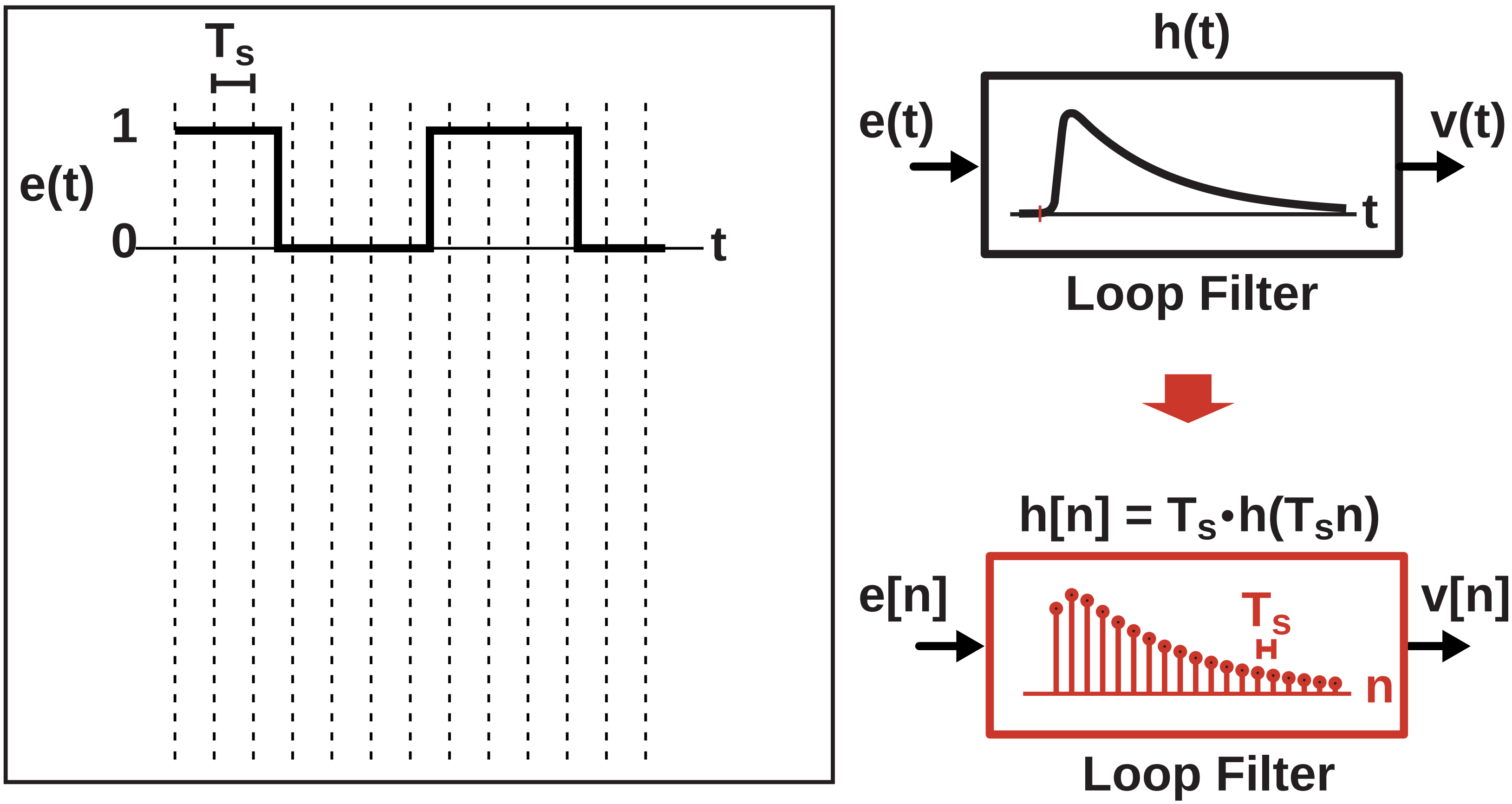
# *Issue: Non-Constant Time Step Brings Complications*




- **Filters and noise sources must account for varying time step in their code implementations**
- **Spectra derived from mixing and other operations can display false simulation artifacts**
- **Setting of time step becomes progressively complicated if multiple time-based circuits simulated at once**

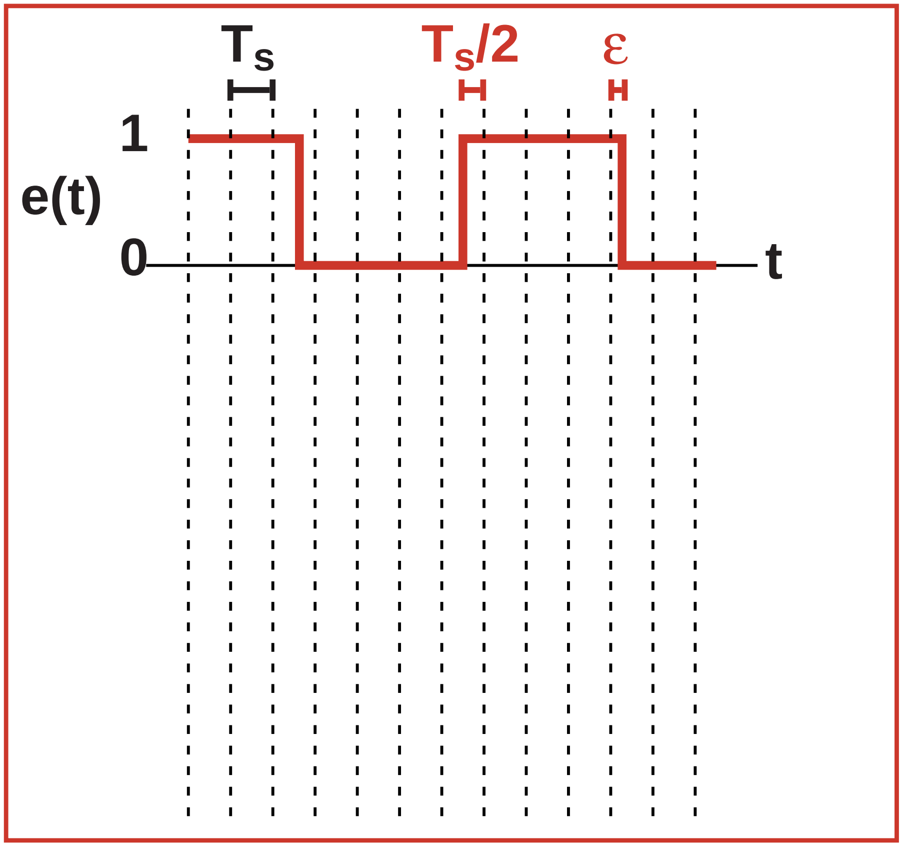
***Is there a better way?***

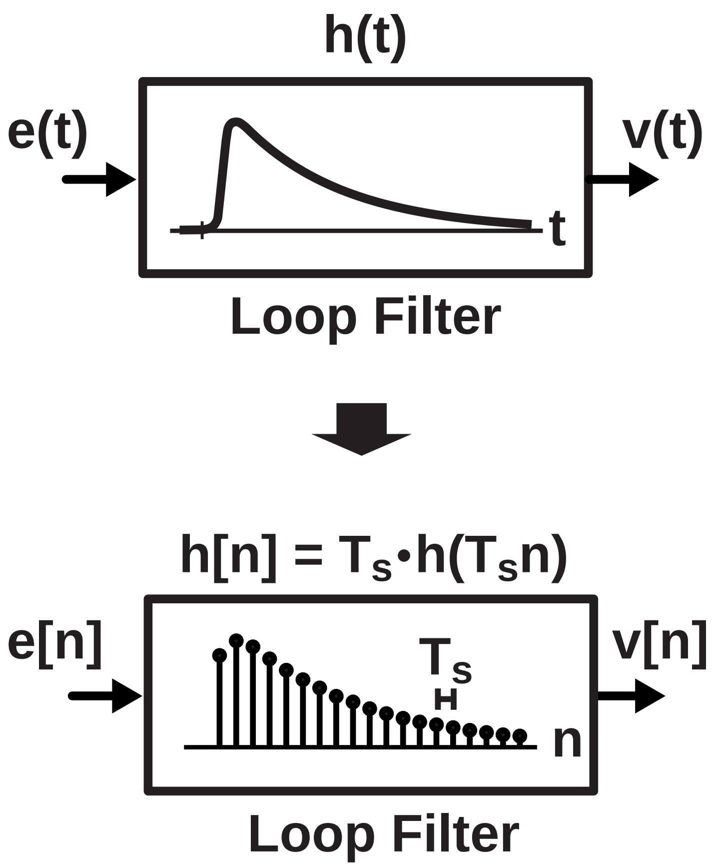
# *Proposed Approach: Use Constant Time Step*




- **Straightforward CT to DT transformation of filter blocks**
  - **Use bilinear transform or impulse invariance methods**
- **Overall computation framework is fast and simple**
  - **Simulator can be based on Verilog, Matlab, C++**

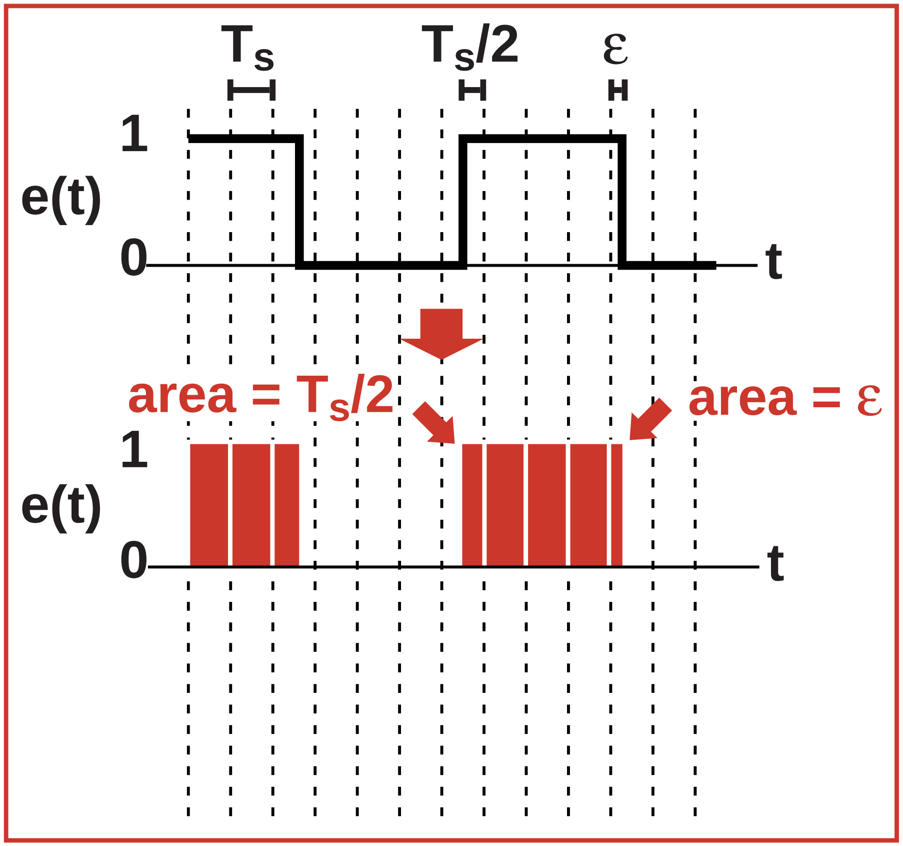
## *Problem: Quantization Noise at PFD Output*

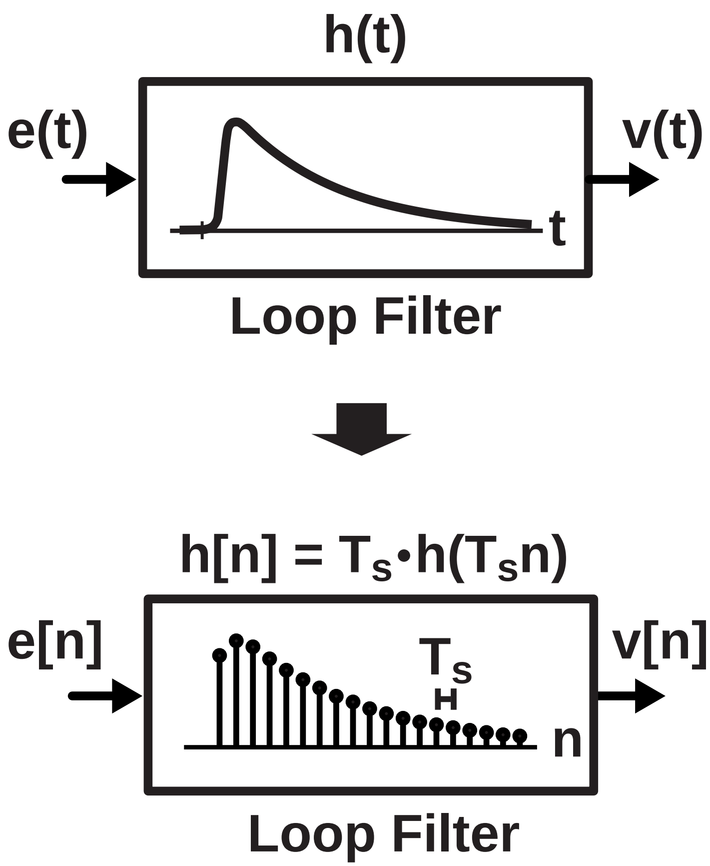



- **Edge locations of PFD output are quantized**
  - **Resolution set by time step: $T_S$**
- **Reduction of $T_S$ leads to long simulation times**

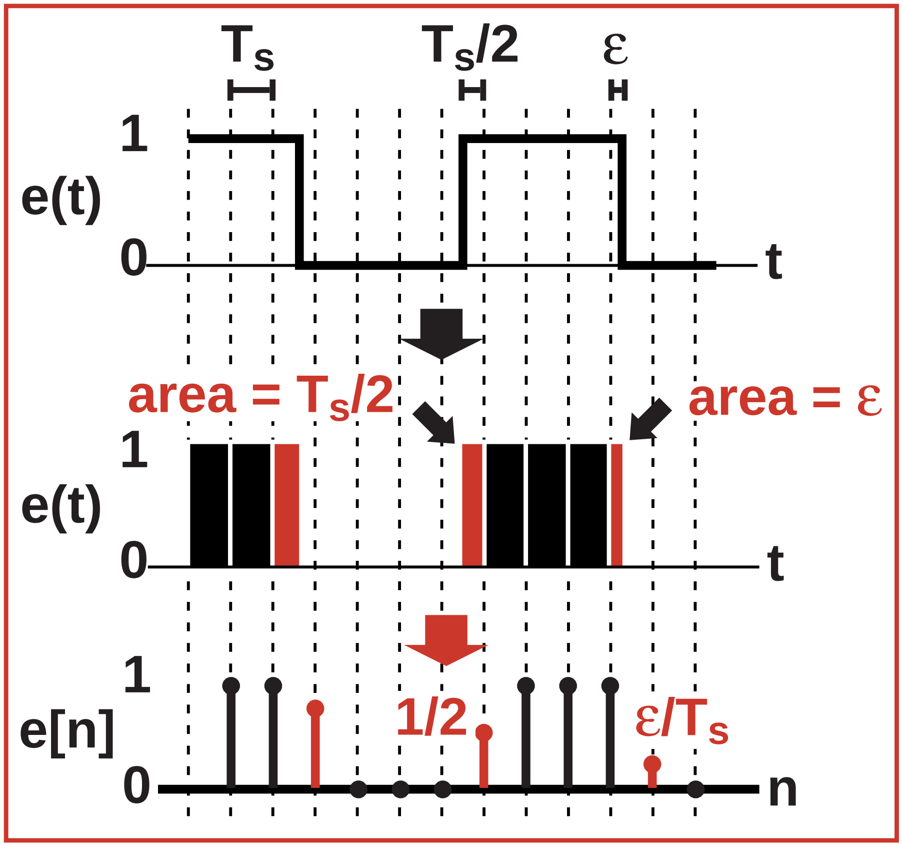
# *Proposed Approach: View as Series of Pulses*

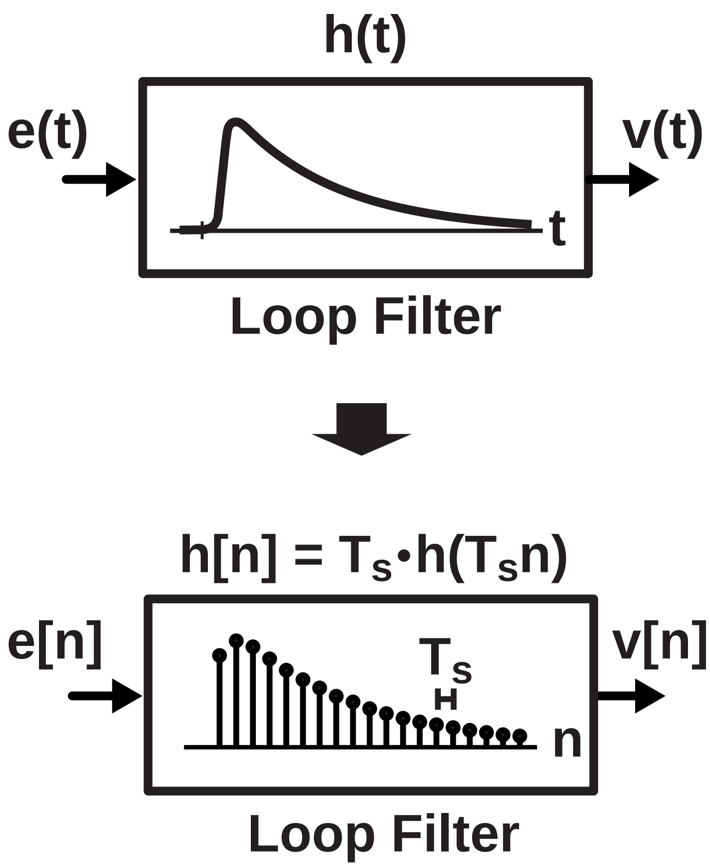



- Area of each pulse set by edge locations
- Key observations:
  - Pulses look like impulses to loop filter
  - Impulses are parameterized by their area and time offset

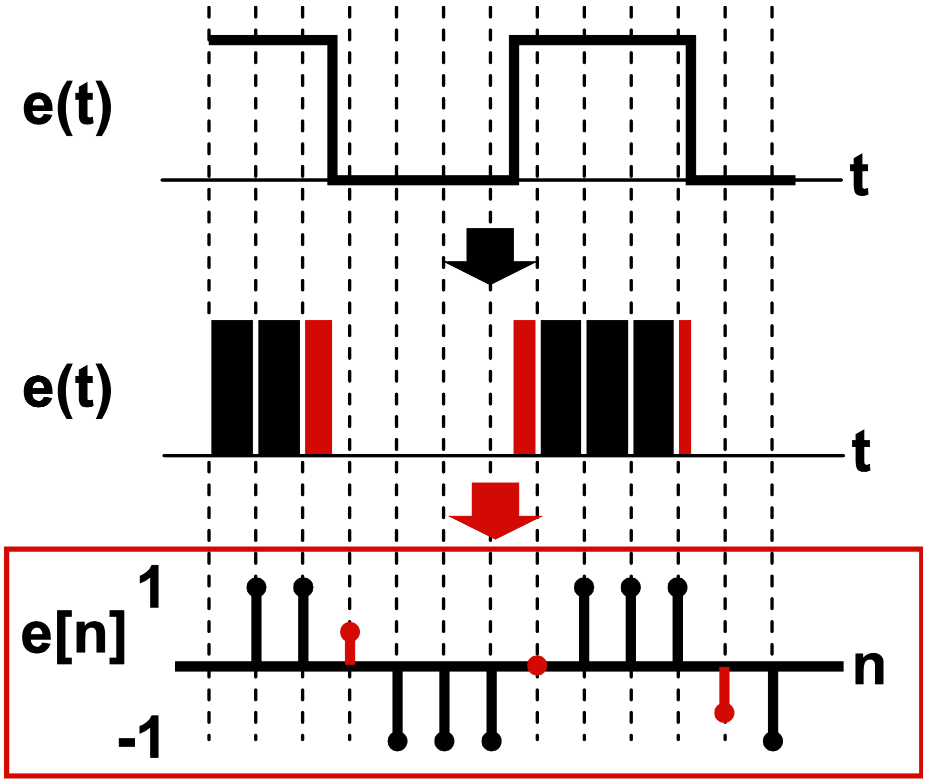
# *Proposed Area Conservation Method*



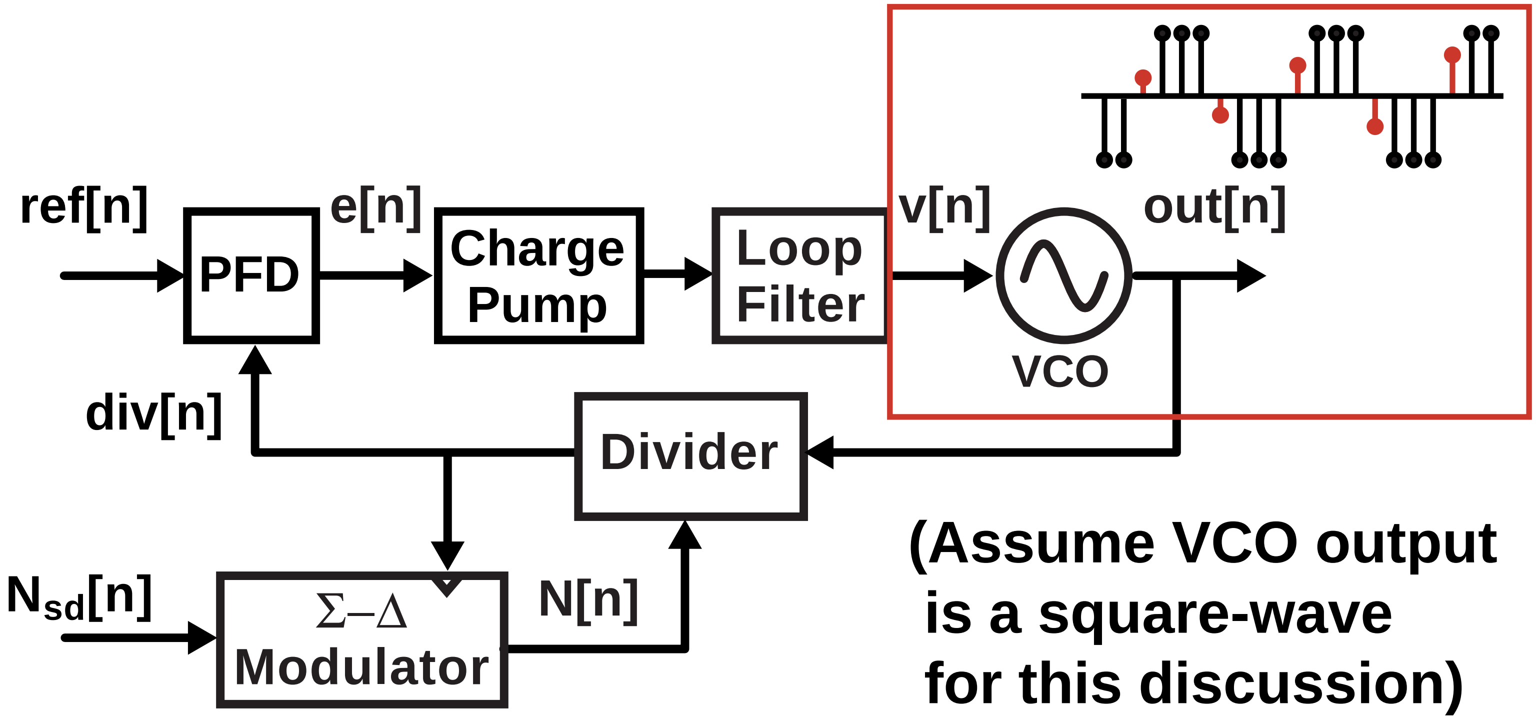

- Set e[n] samples according to pulse areas
  - Leads to very accurate results
  - Fast computation

# *Double_Interp Protocol*




- Protocol sets signal samples to -1 or 1 except for transitions
  - Transition values between -1 and 1 are directly related to the edge time location
  - Can be implemented in C++, Verilog, and Matlab/Simulink

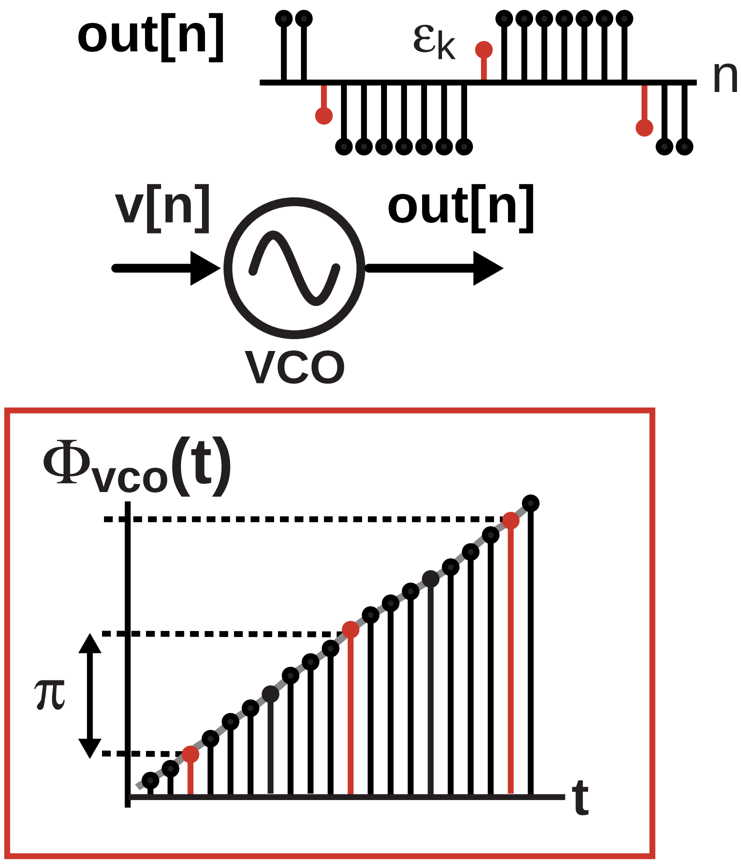
# *VCO is a Key Block for Double_Interp Encoding*




(Assume VCO output is a square-wave for this discussion)

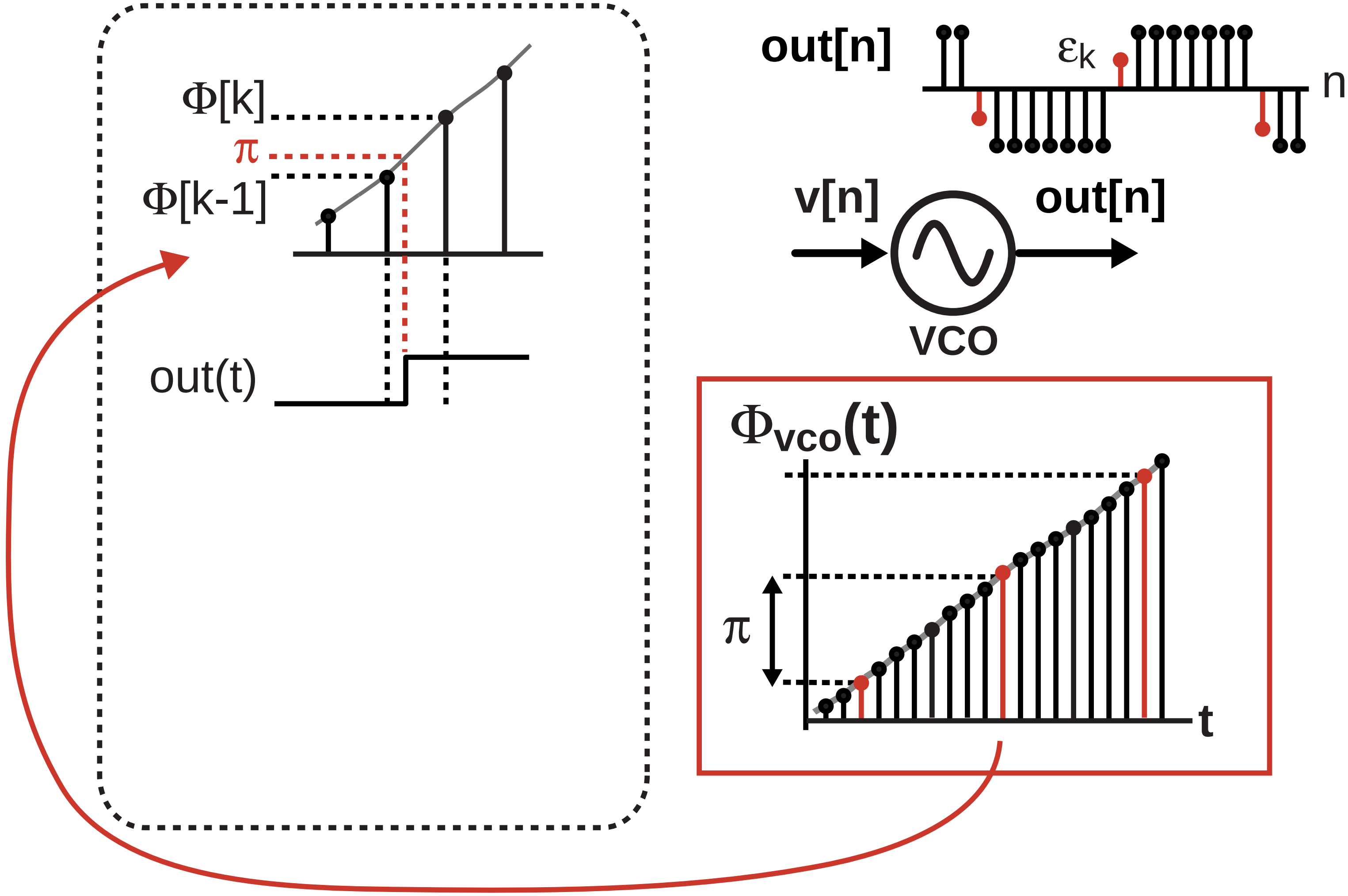
- The VCO block is the key translator from a bandlimited analog input to an edge-based waveform
  - We can create routines in the VCO that calculate the edge times of the output and encode their values using the double_interp protocol

# Calculation of Transition Time Values



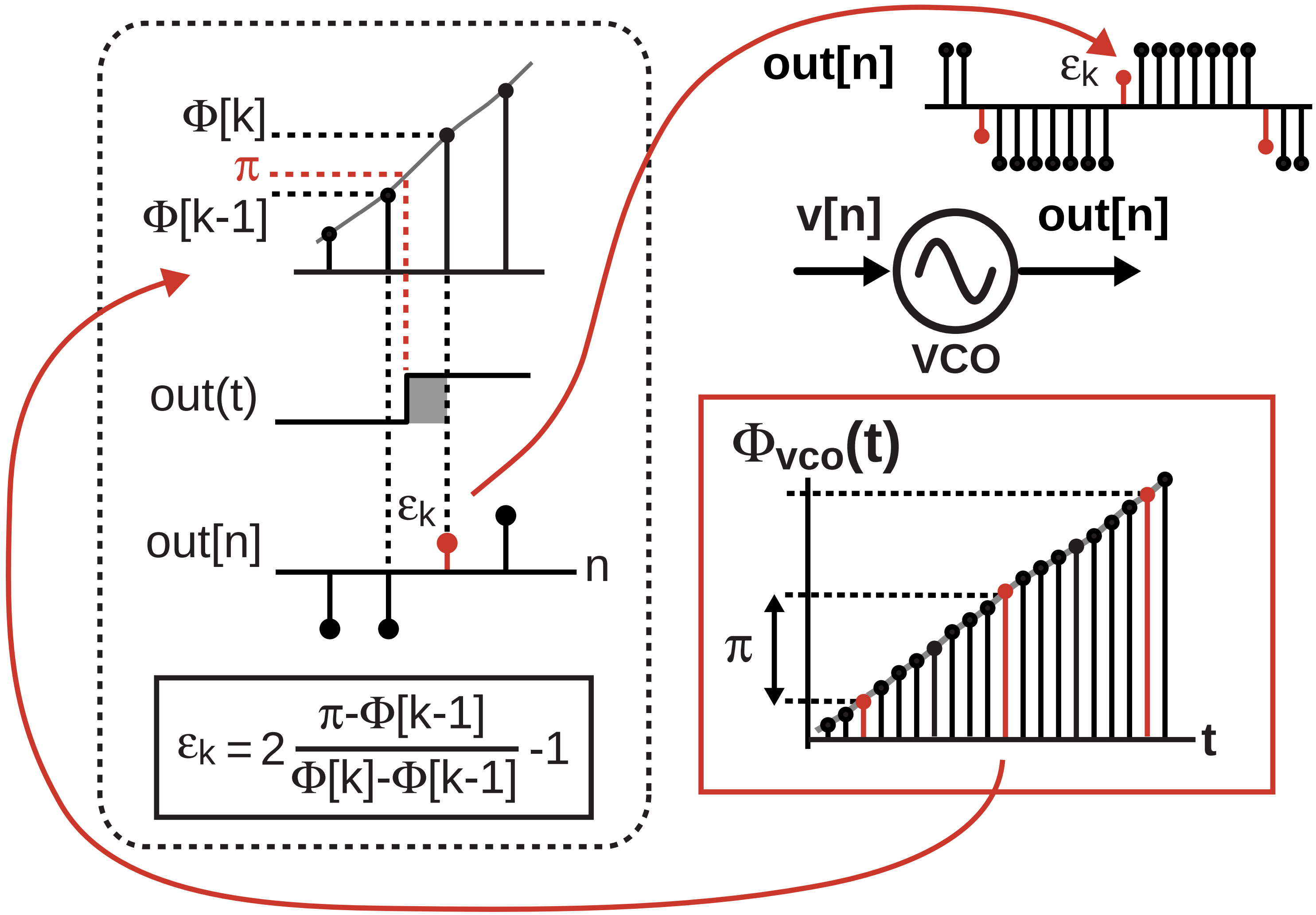

- **Model VCO based on its phase**

# *Calculation of Transition Time Values (cont.)*



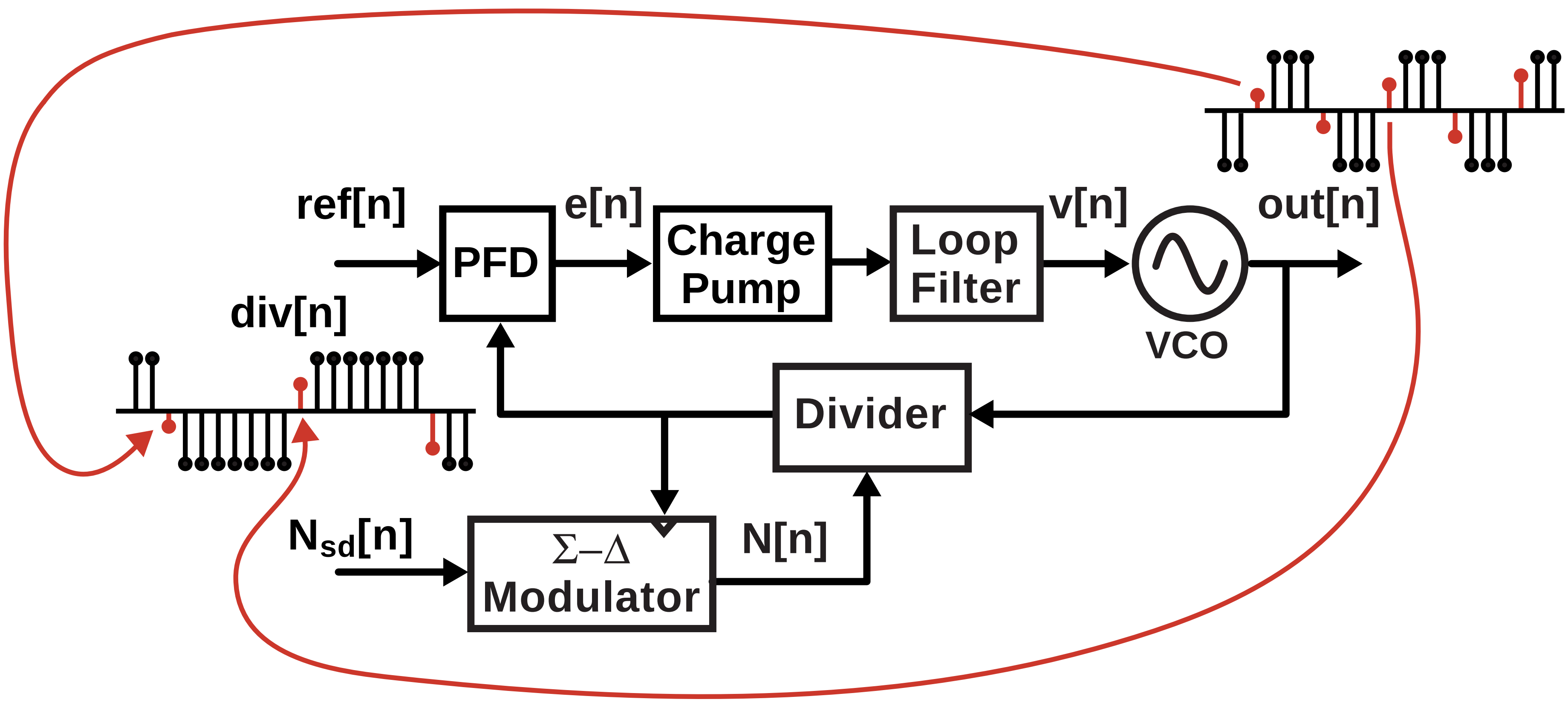

- **Determine output transition time according to phase**

# *Calculation of Transition Time Values (cont.)*



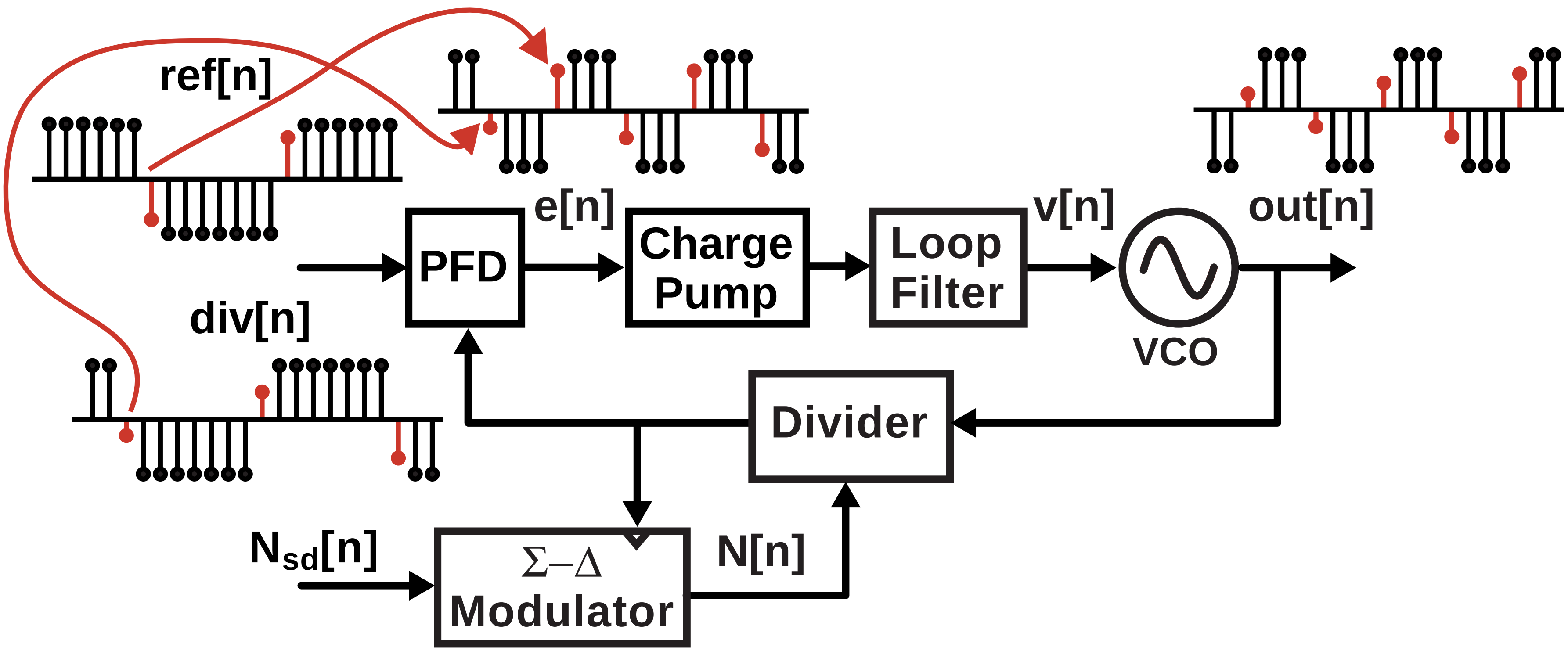

- **Use first order interpolation to determine transition value**

# *Processing of Edges using Double_Interp Protocol*



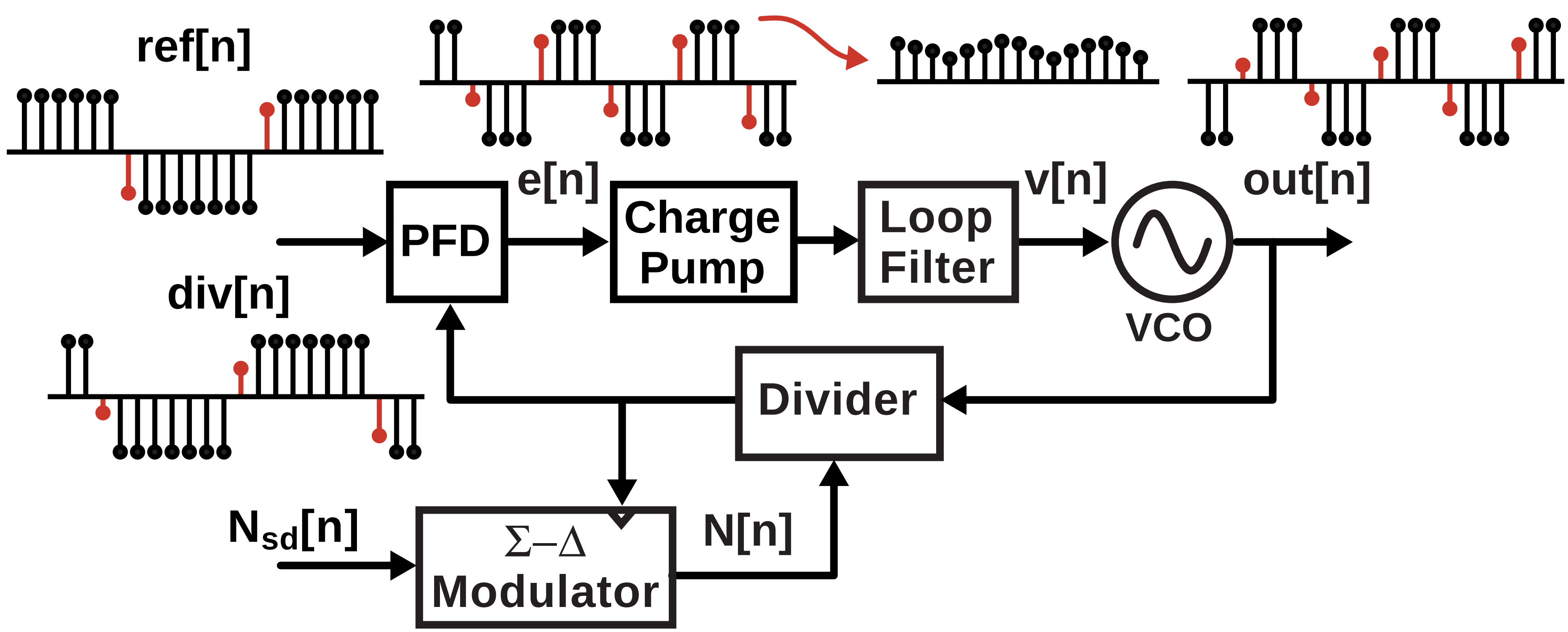

- **Frequency divider block simply passes a sub-sampling of edges based on the VCO output and divide value**

## *Processing of Edges using Double_Interp Protocol*



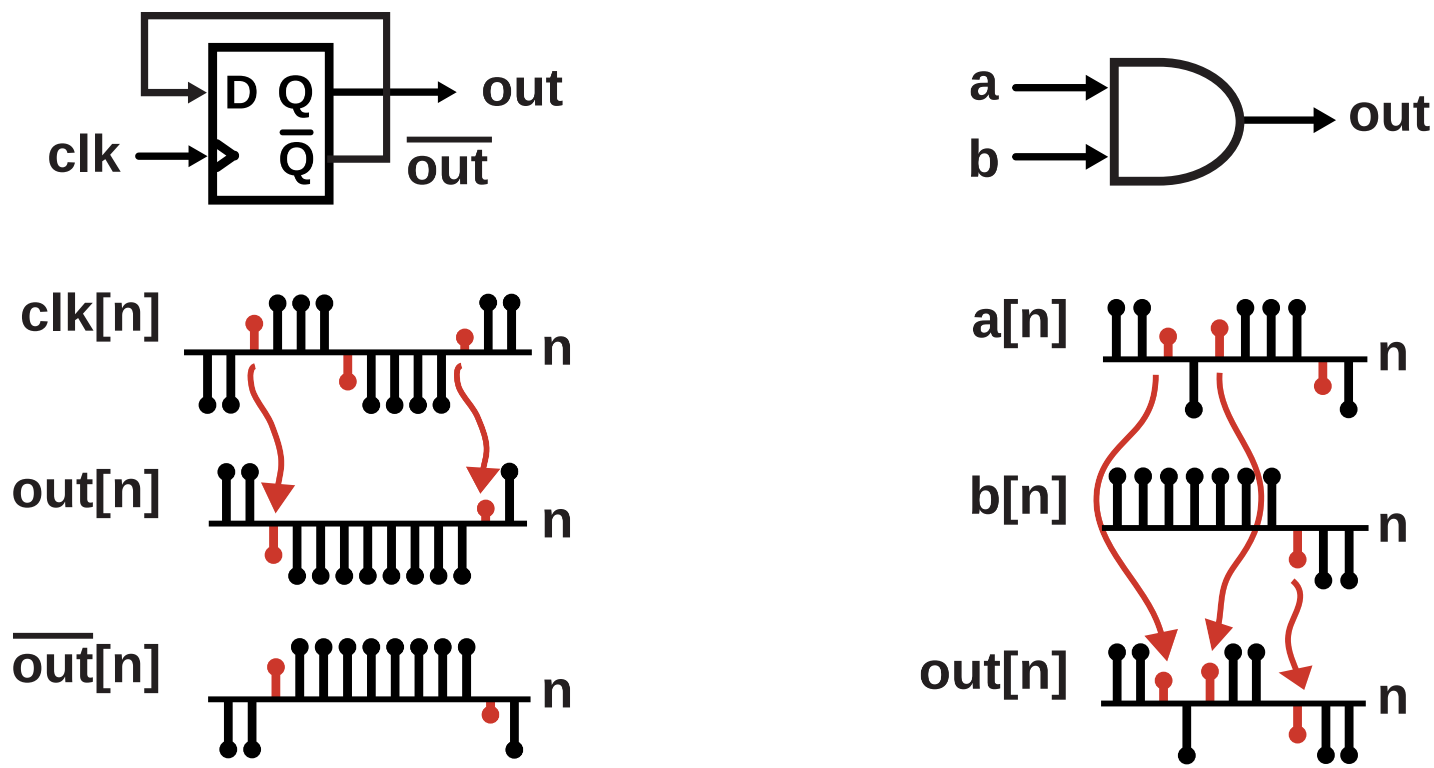

- Phase Detector compares edges times between reference and divided output and then outputs pulses that preserve the time differences

# *Processing of Edges using Double_Interp Protocol*



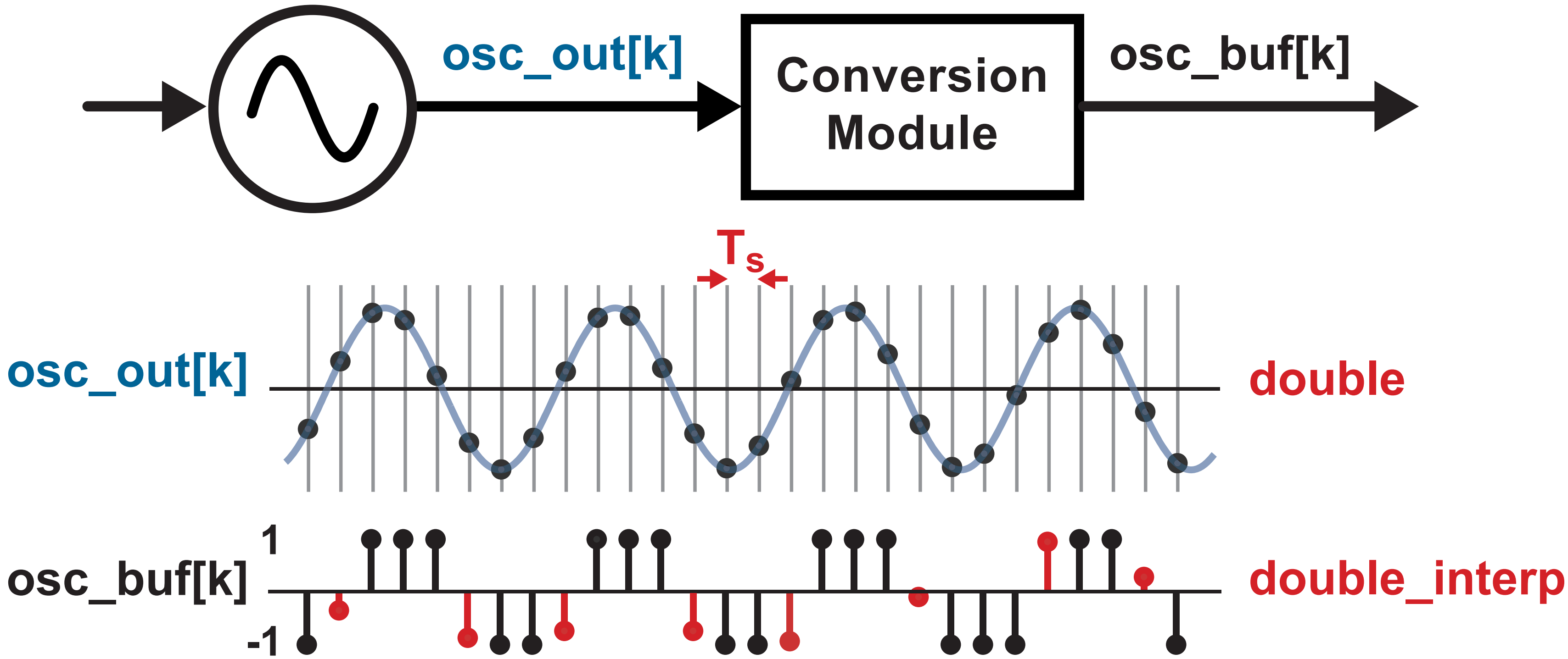

- Charge Pump and Loop filter operation is straightforward to model
  - Simply filter pulses from phase detector as discussed earlier

# *Using the Double_Interp Protocol with Digital Gates*



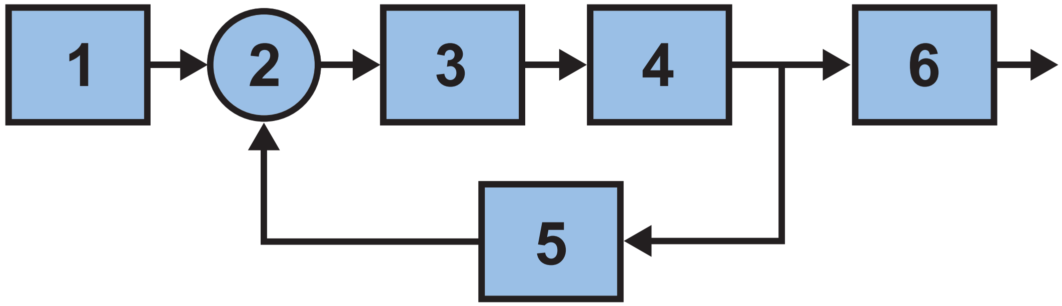

- **Relevant timing information contained in the input that causes the output to transition**
  - **Determine which input causes the transition, then pass its transition value to the output**

## *Using the Double_Interp Protocol with Sine Waves*



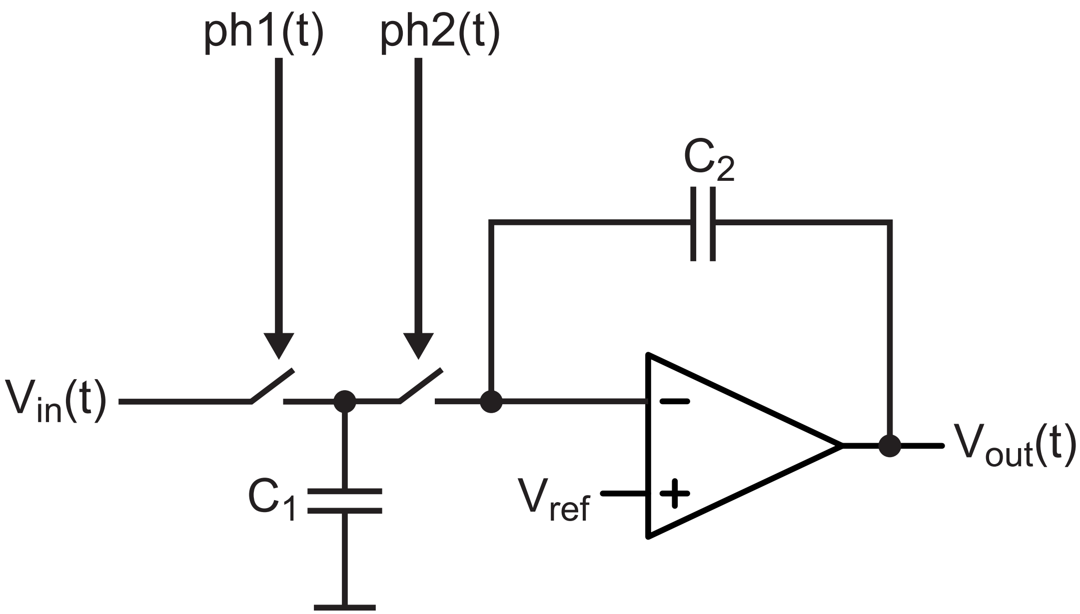

- **In some systems we must deal directly with sine waves**
  - **An explicit conversion module should be utilized**
    - We can convert to double_interp protocol using a similar interpolation technique as described earlier
  - **See gmsk_limitamp module within GMSK_Example library**
    - Used in module gmsk_pll_transmitter in the same library

# *Summary of Block-by-Block Computation Method*




- **Requires unilateral flow through blocks**
- **Impacts phase margin of feedback loops**
  - **Need $1/T_s$ >> bandwidth of feedback loop**
  - **Need proper ordering of blocks (automatic in CppSim)**
- **Constant time step simplifies simulation**
  - **Easier block descriptions**
  - **Frequency domain analysis become straightforward**
  - **Time-based signals handled with double_interp protocol**

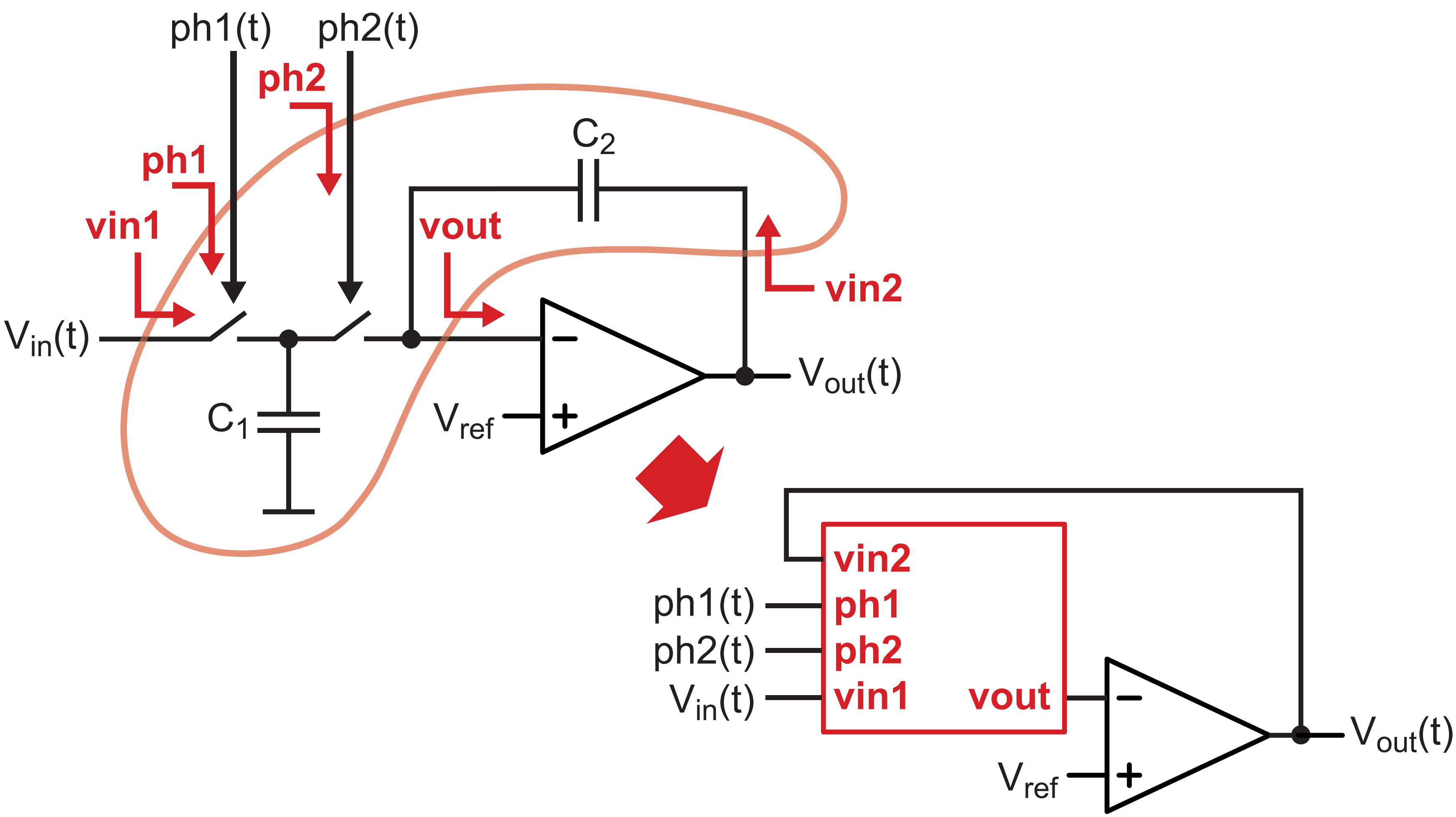
# *Simulation of Switched Capacitor Circuits*




- **Capacitor network with switches can be modeled with unilateral flow blocks, but many practical issues:**
  - **Very challenging for beginners, tedious for experts**
  - **Difficult to check correctness of model**
  - **Difficult to investigate alternative architectures**

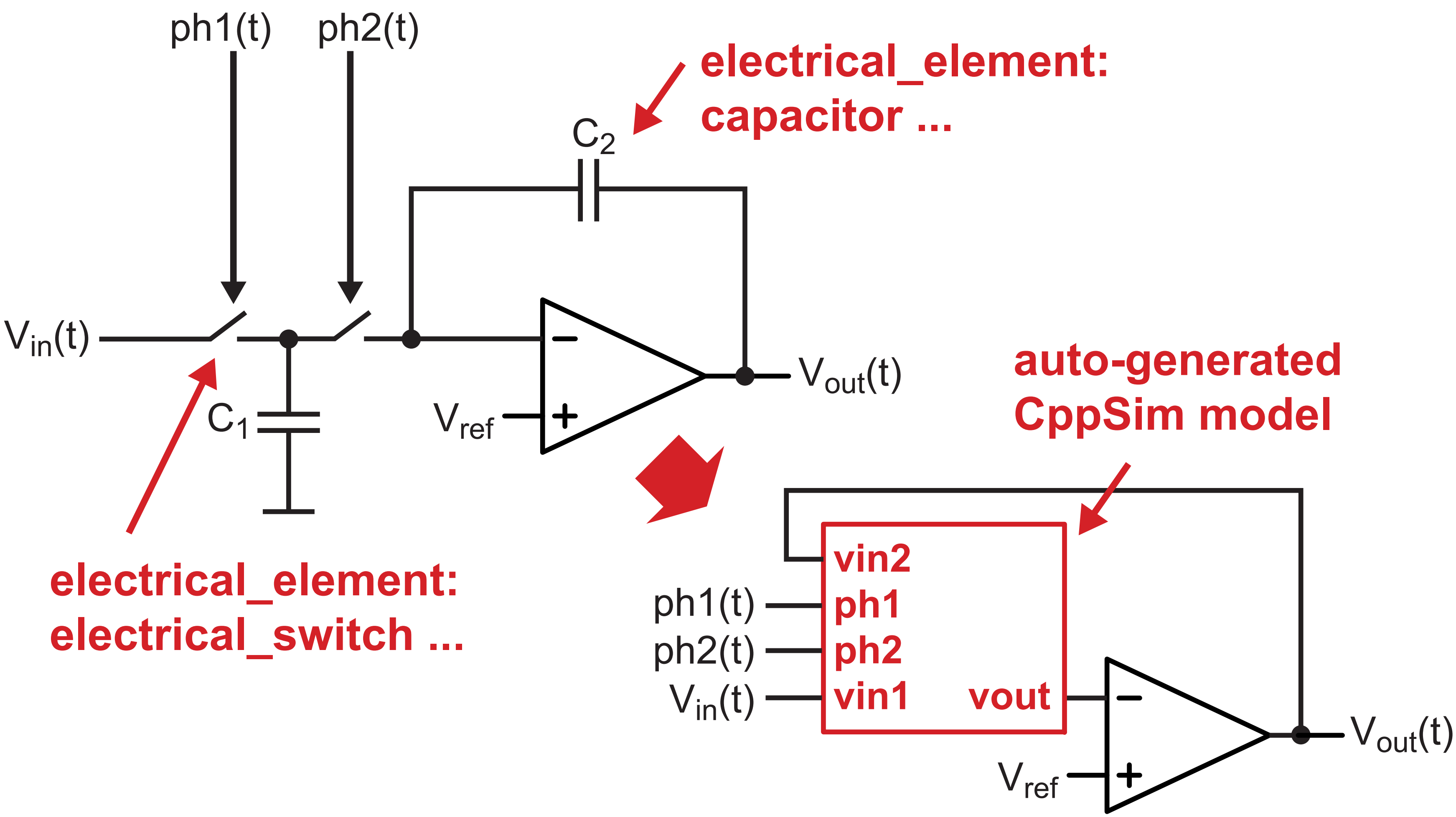
**We need a way to automate the modeling process…**

# *Automatic Unilateral Model Generation*



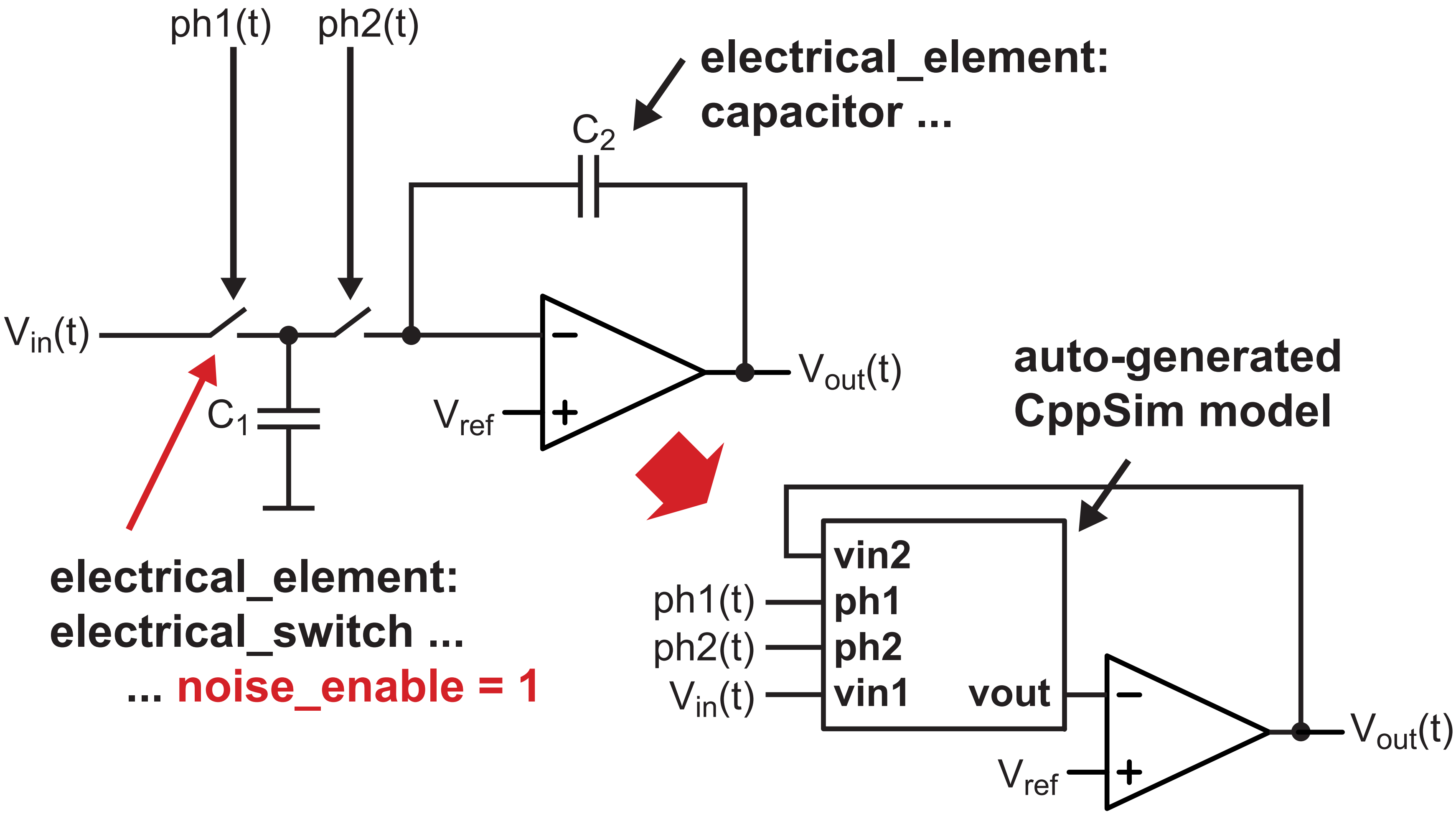

- **A linear network with switches can be represented as a state-space model with switch dependent matrices**
  - **An equivalent unilateral flow block is created**

# *CppSim Approach to Linear Networks with Switches*



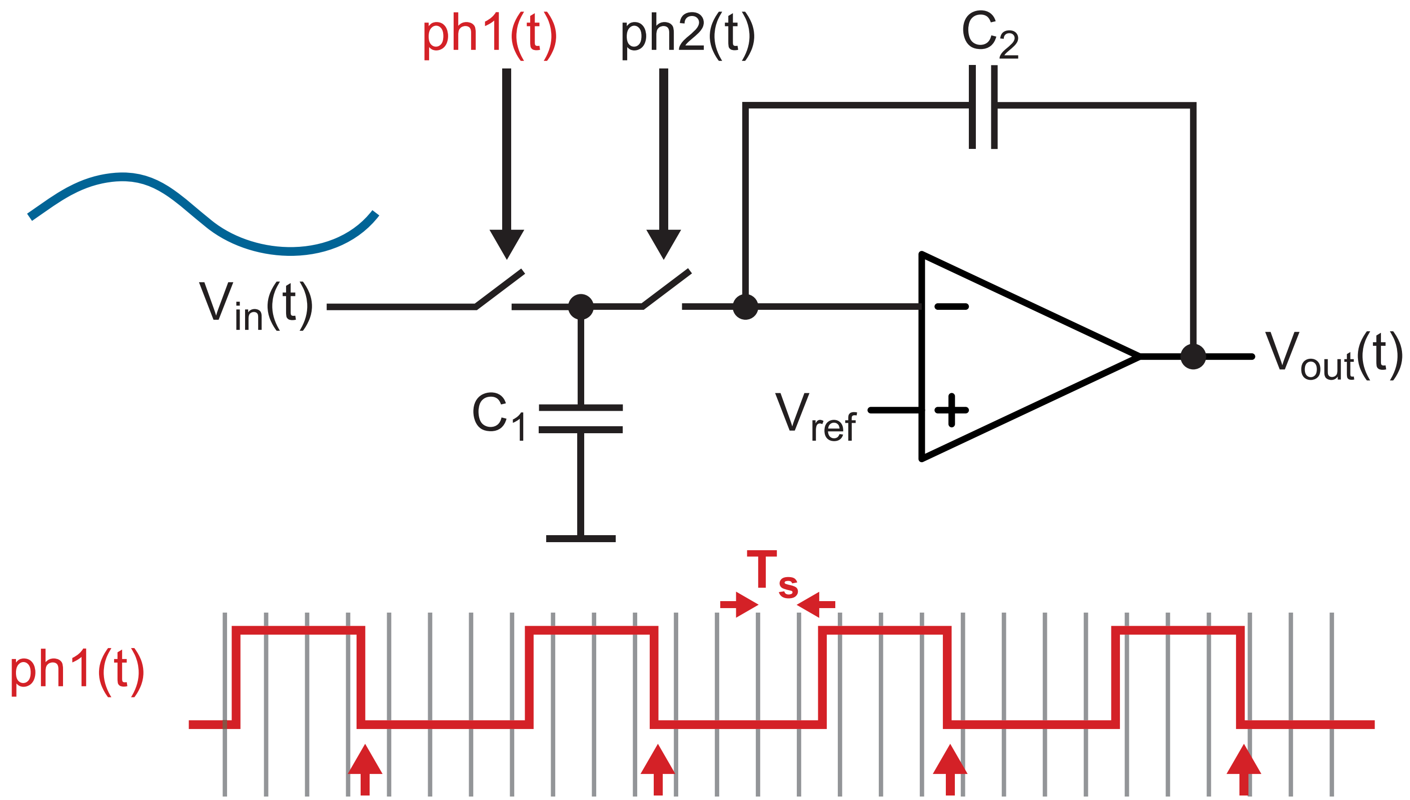

- **User specifies the CppSim model for linear elements, switches, and diodes using electrical_element: command**
  - **Draw the schematic and CppSim takes care of the rest!**

# *Transient Noise Analysis is Supported*



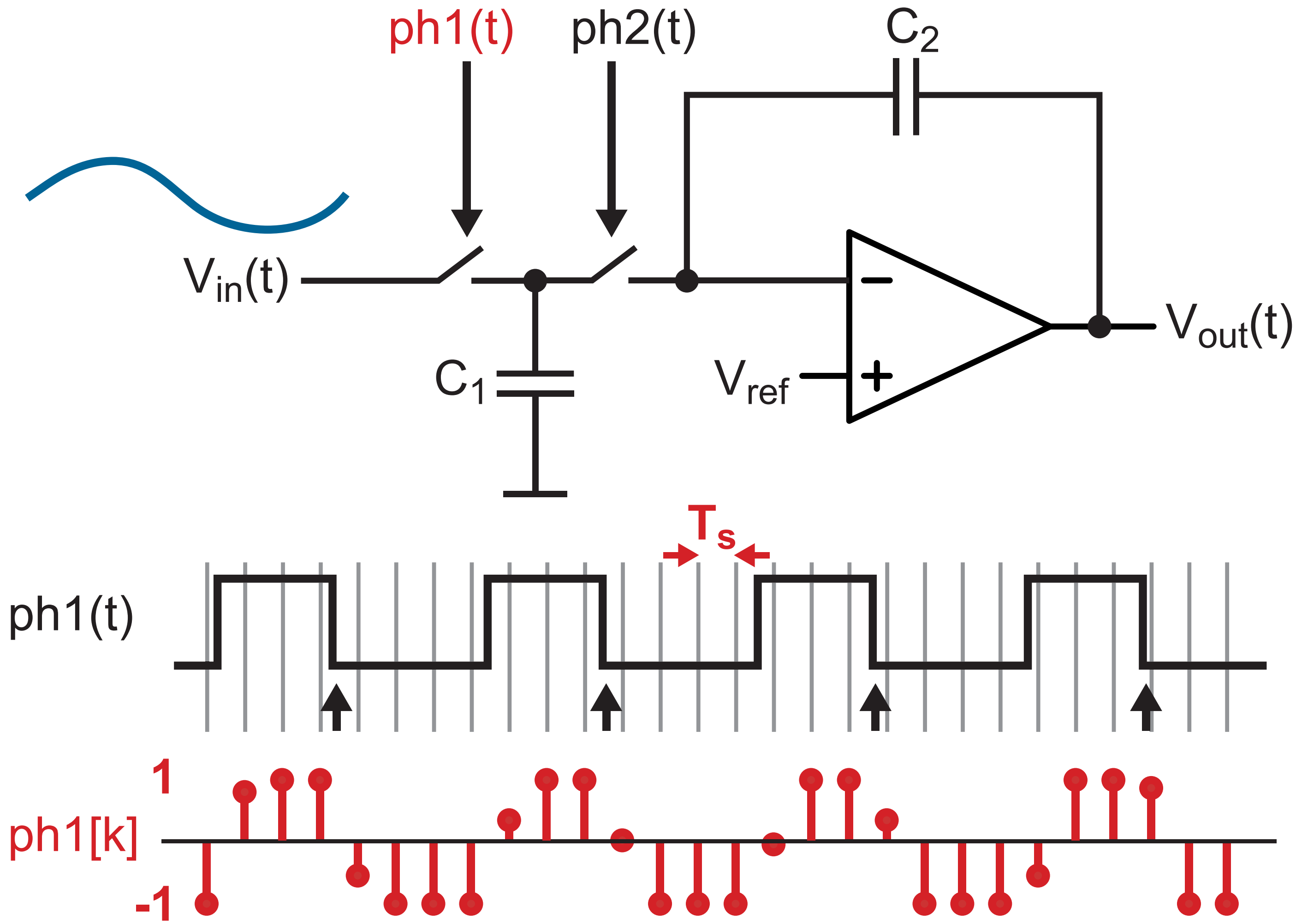

- Resistors, switches, voltage/current thermal + 1/f noise
- For kT/C noise, need adequately small time step, $T_s$
  - Accuracy requires $1/T_s$ > 20*bandwidth of switch settling time

# *Time Based Signals with Electrical Elements*



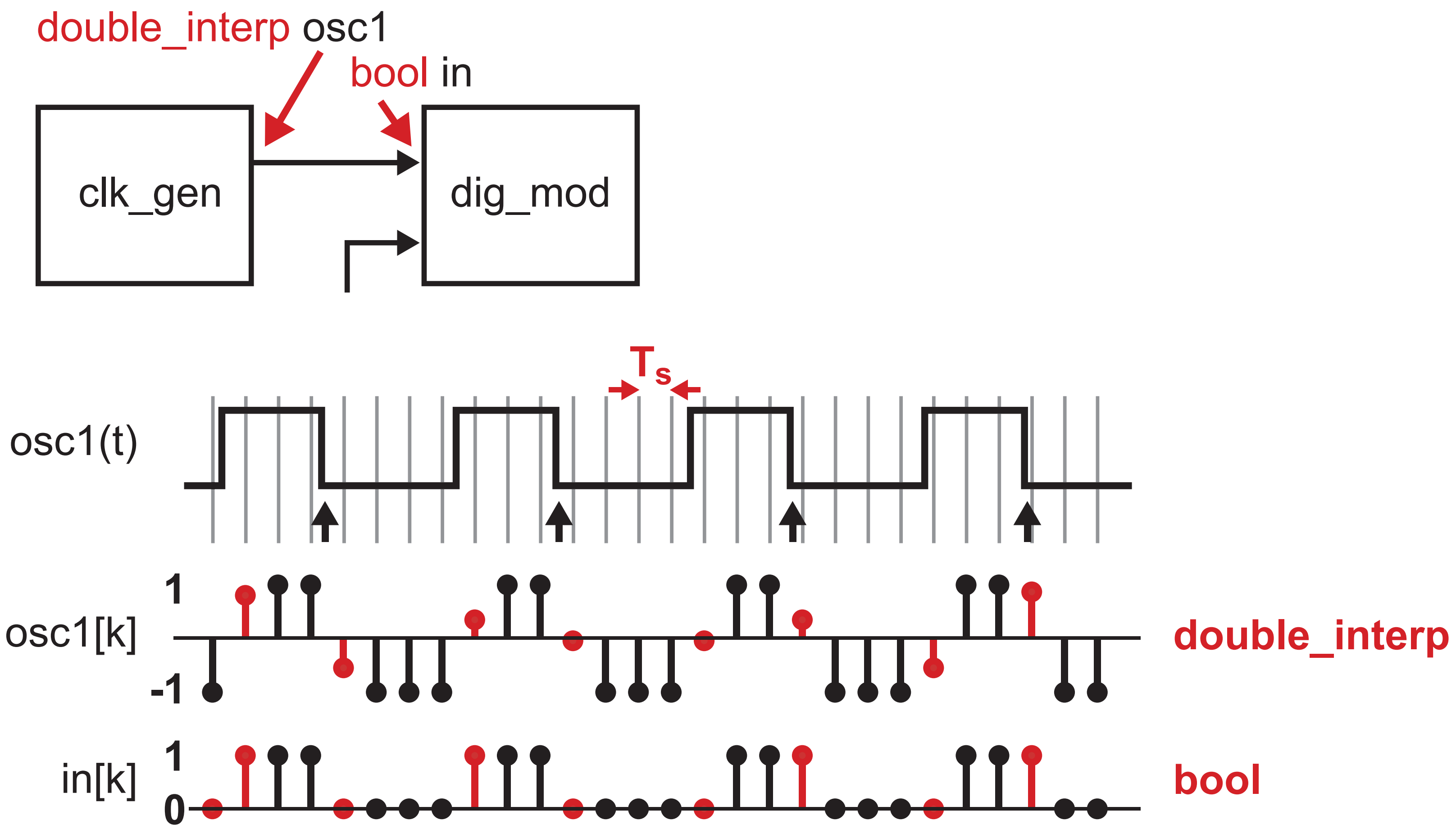

- **Constant time step of CppSim could lead to quantization effects on sample times of clock edges**
  - **Would result in sampling errors of input waveform**

## *Leverage Double_Interp Protocol*



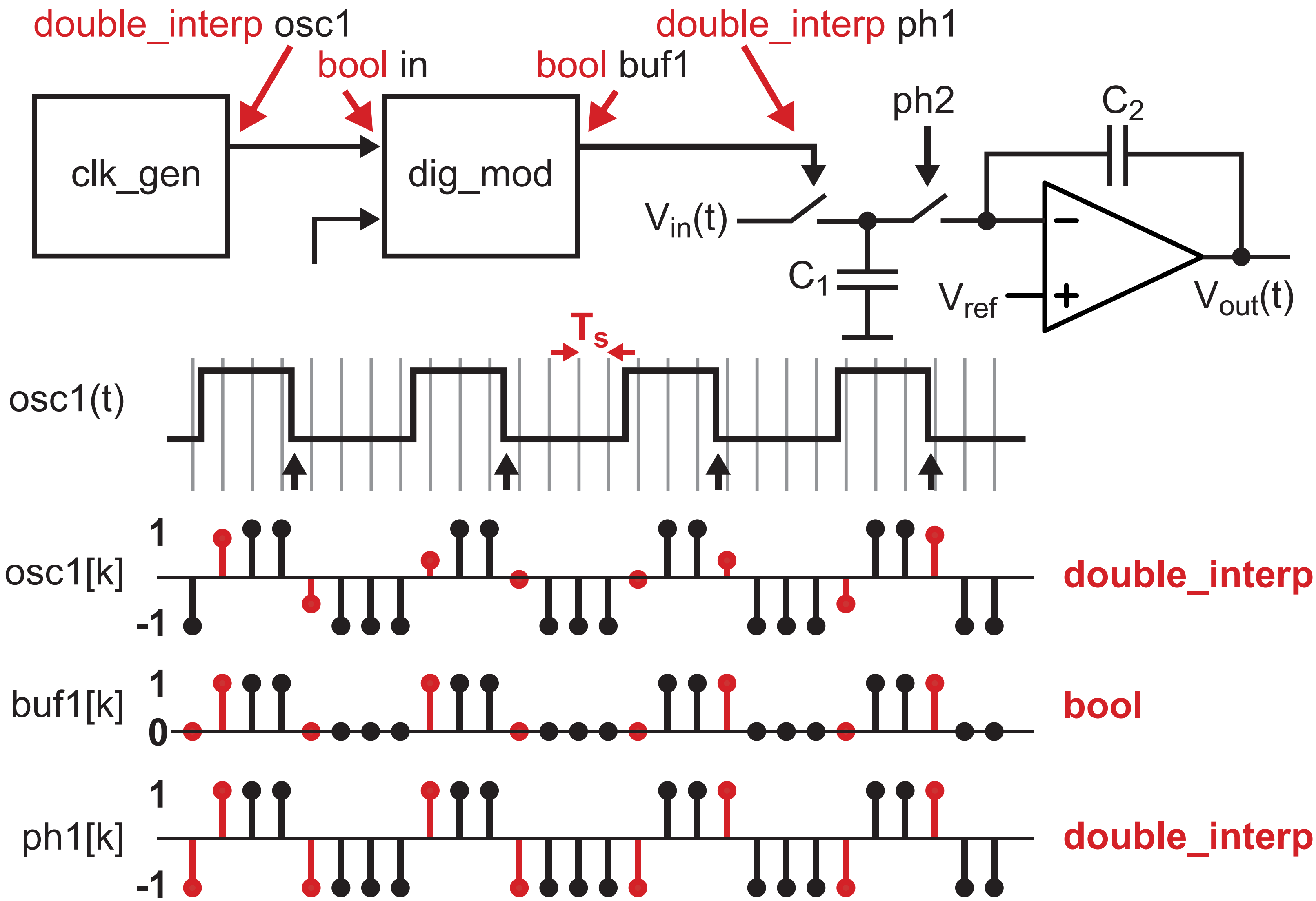

- **Electrical switches within CppSim require double_interp signals for the control nodes**
  - **Good timing accuracy achieved despite constant time step**

# *Feeding Bool Input with Double_Interp Signal*



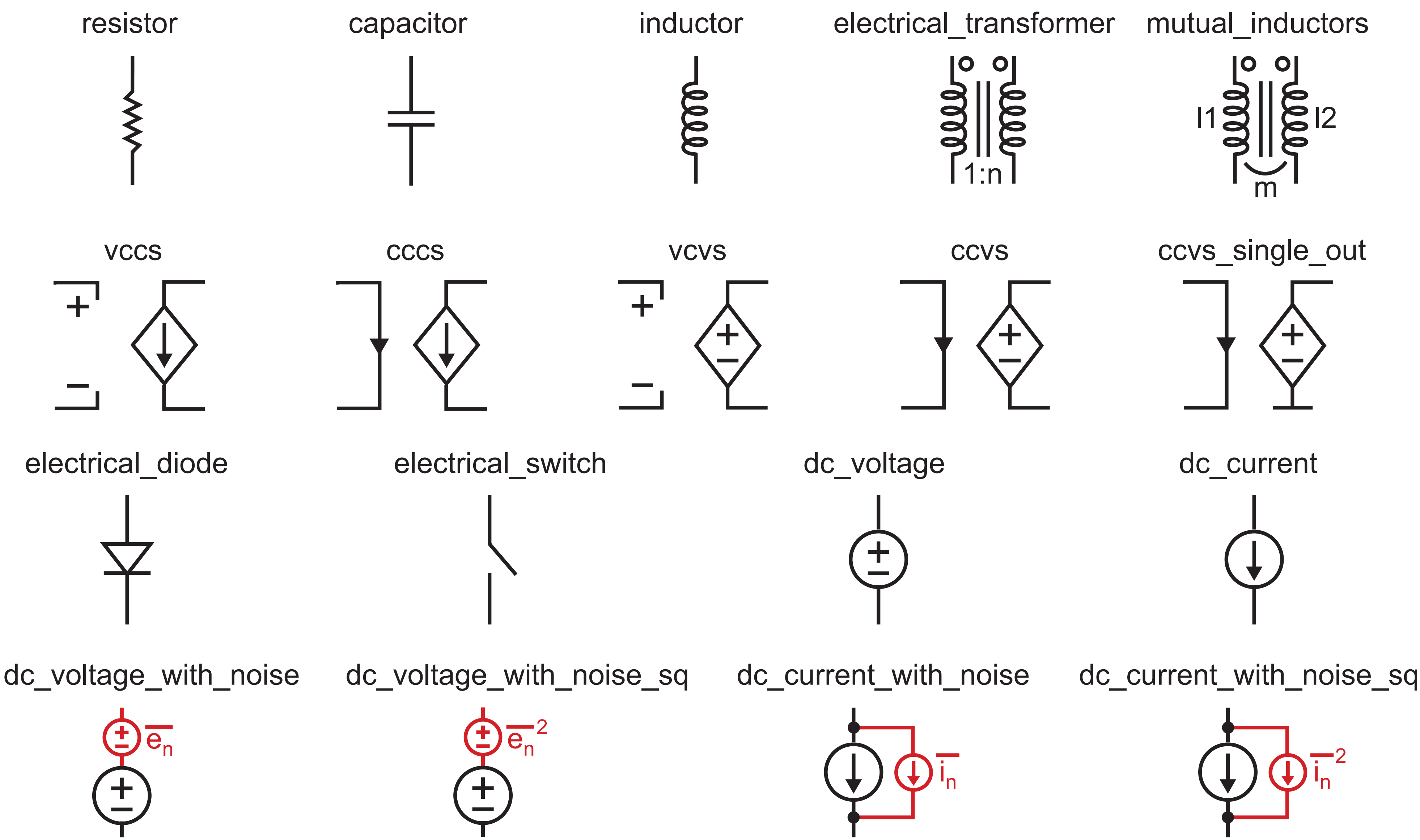

- Conversion module automatically inserted
  - -1,1 signaling converted to 0,1 signaling
  - High resolution edge timing information is lost

# *Feeding Double_Interp Input with Bool Signal*

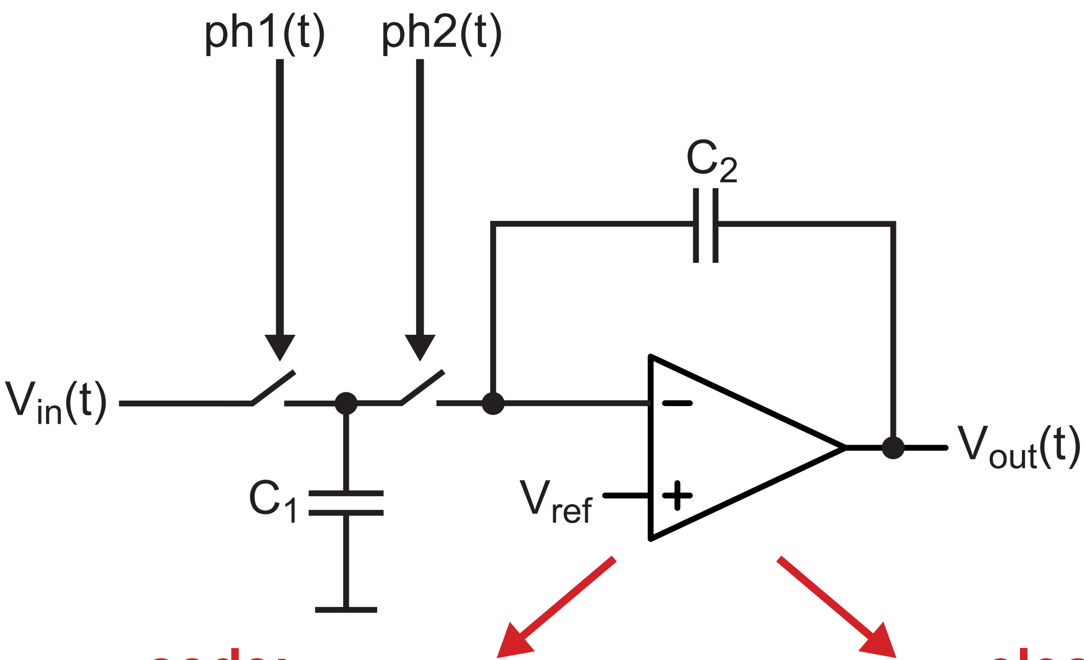



- **Automatic translation of 0,1 signaling to -1,1 signaling**
  - **Loss of timing information causes quantization noise!**

# Supported Electrical Elements in CppSim



# *CppSim Code Versus Electrical Element Modules*



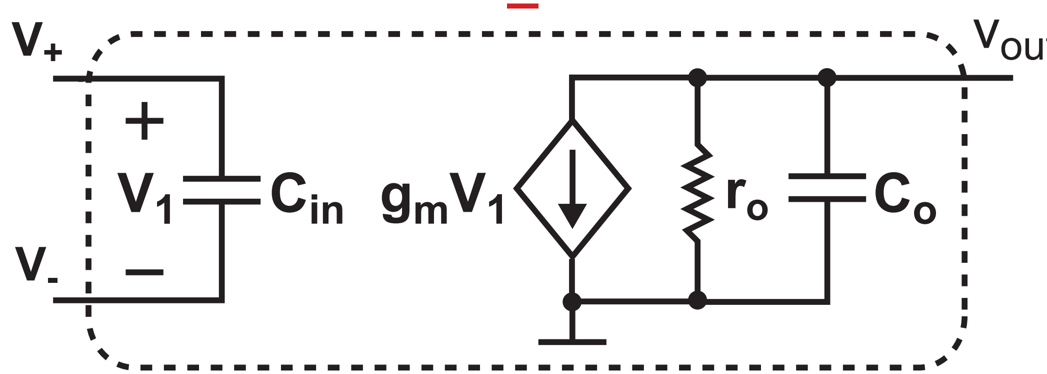

**code:**

**Filter filt1("K","1+1/wo*s",...)**

**⋮**

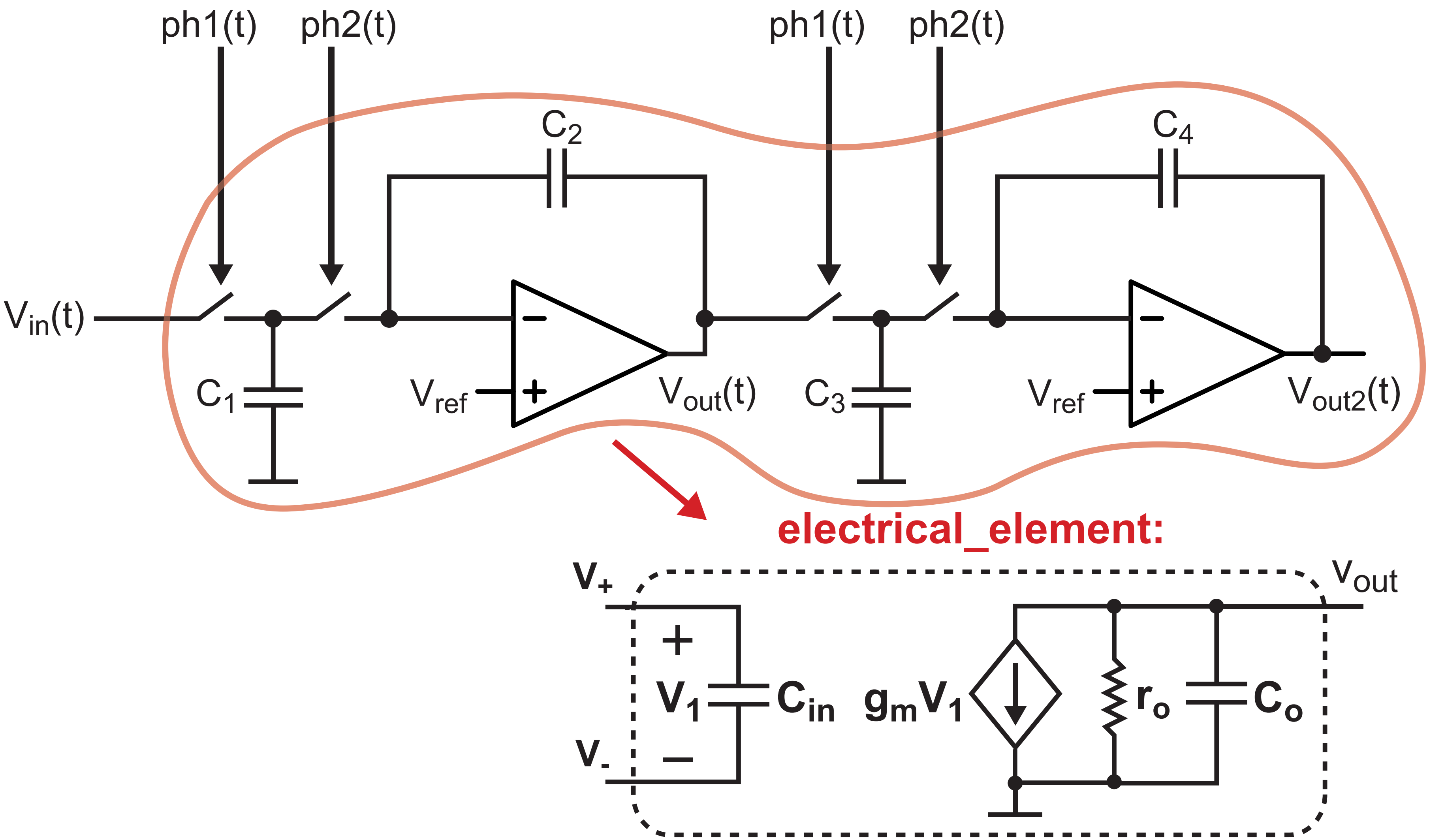
**vout = filt1(vinp-vinm)**

**electrical_element:**



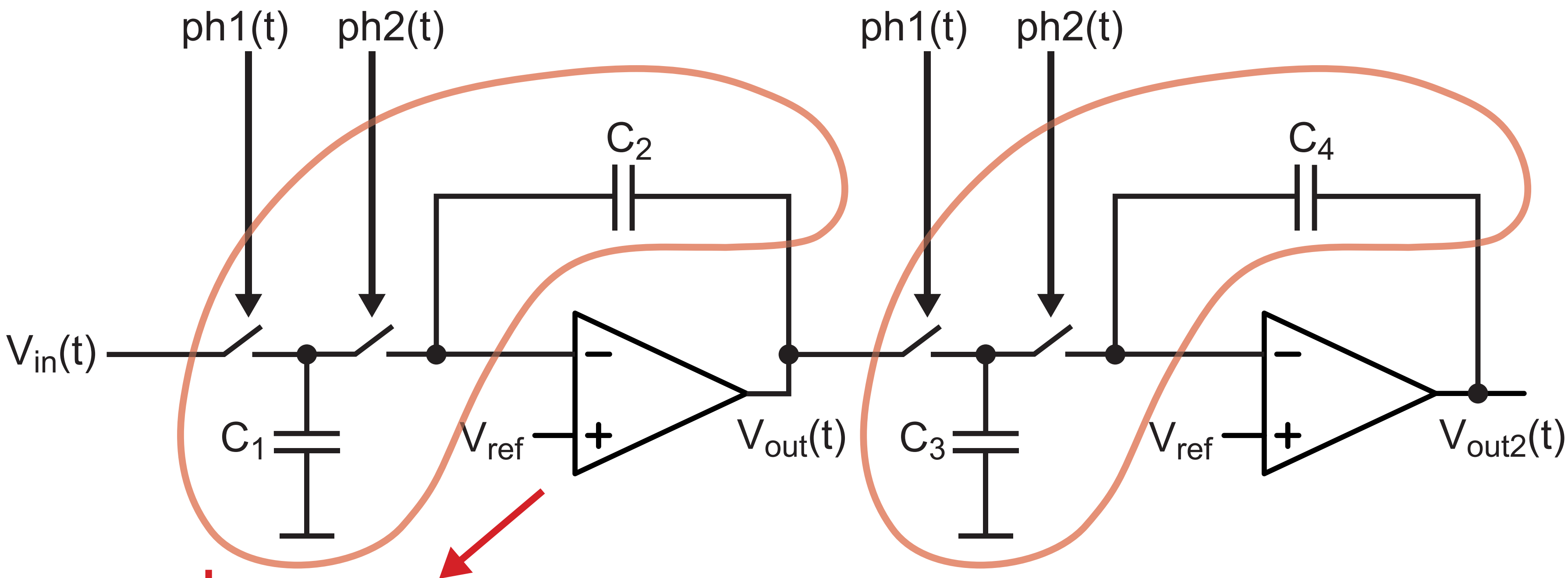

- **Which approach is best for circuit blocks such as opamps?**

# *Complexity Issue with Electrical Element Modules*




- **State-space calculations increase as (number of elements)$^2$**
  - **Large networks dramatically slow down simulation speed**

# *Code Modules Allow De-Coupling Between Networks*




**code:**

**Filter filt1("K","1+1/wo*s",...)**

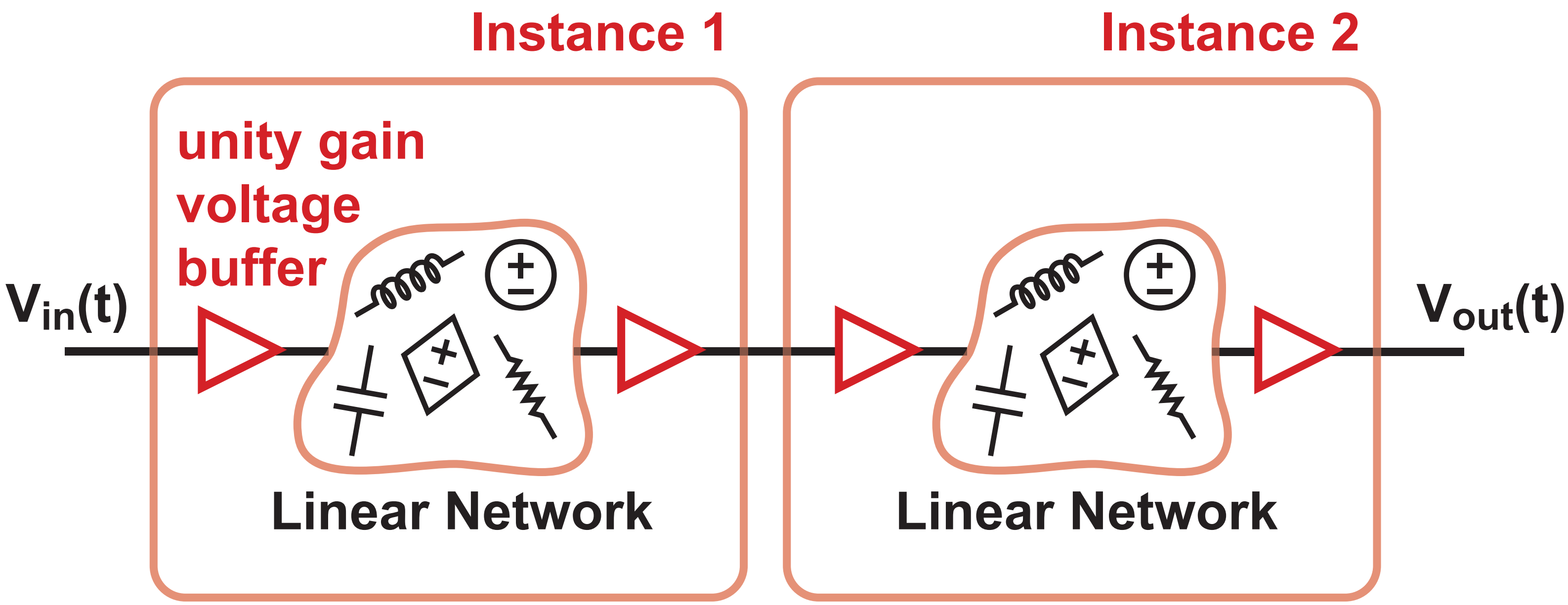
**⋮**

**vout = filt1(vinp-vinm)**

- **Code modules are not sensitive to loading**
  - **Allows CppSim to automatically separate into sub-networks**

**Code modules preferred to achieve fast simulation speed**

# *Impact of Hierarchy on Electrical Element Networks*

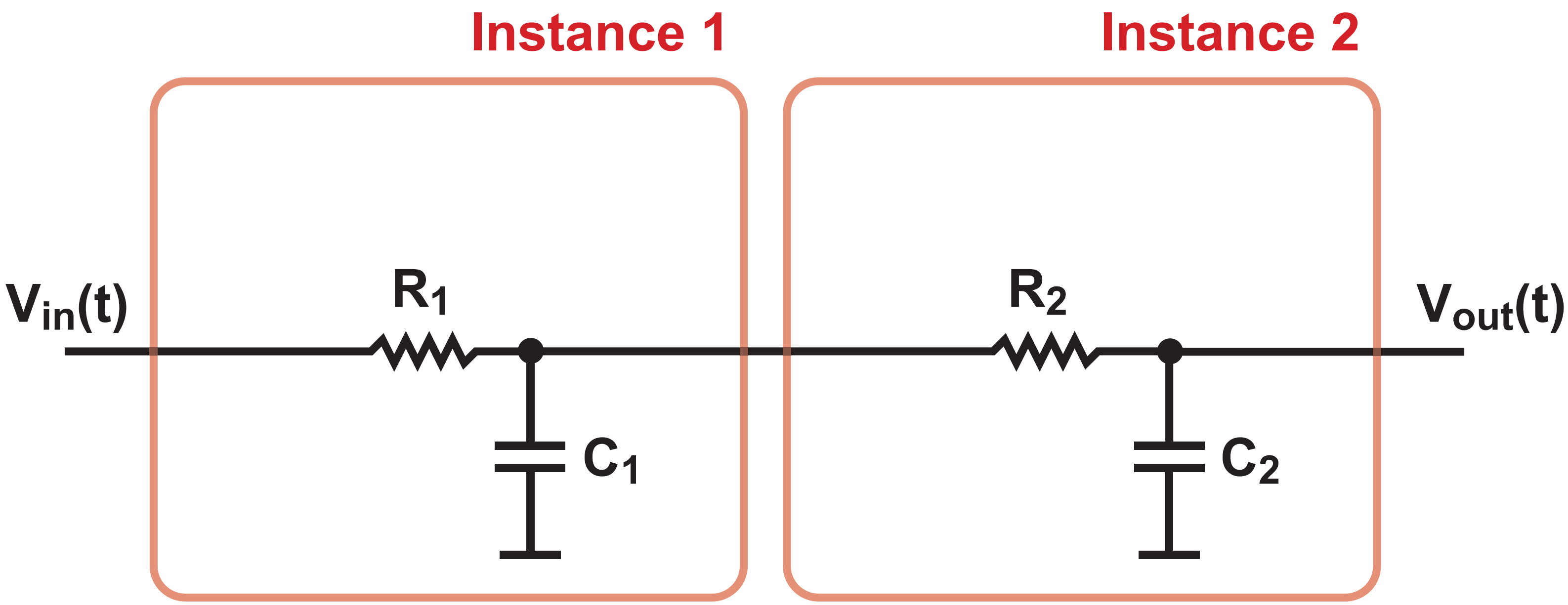



- **CppSim implicitly inserts unity gain voltage buffers at all inputs and outputs of instances**
  - **Allows hierarchical simulation structure of overall system to be retained**
  - **De-couples networks at instance level to discourage creation of large state-space models**

# *Example: A Second Order RC Network*

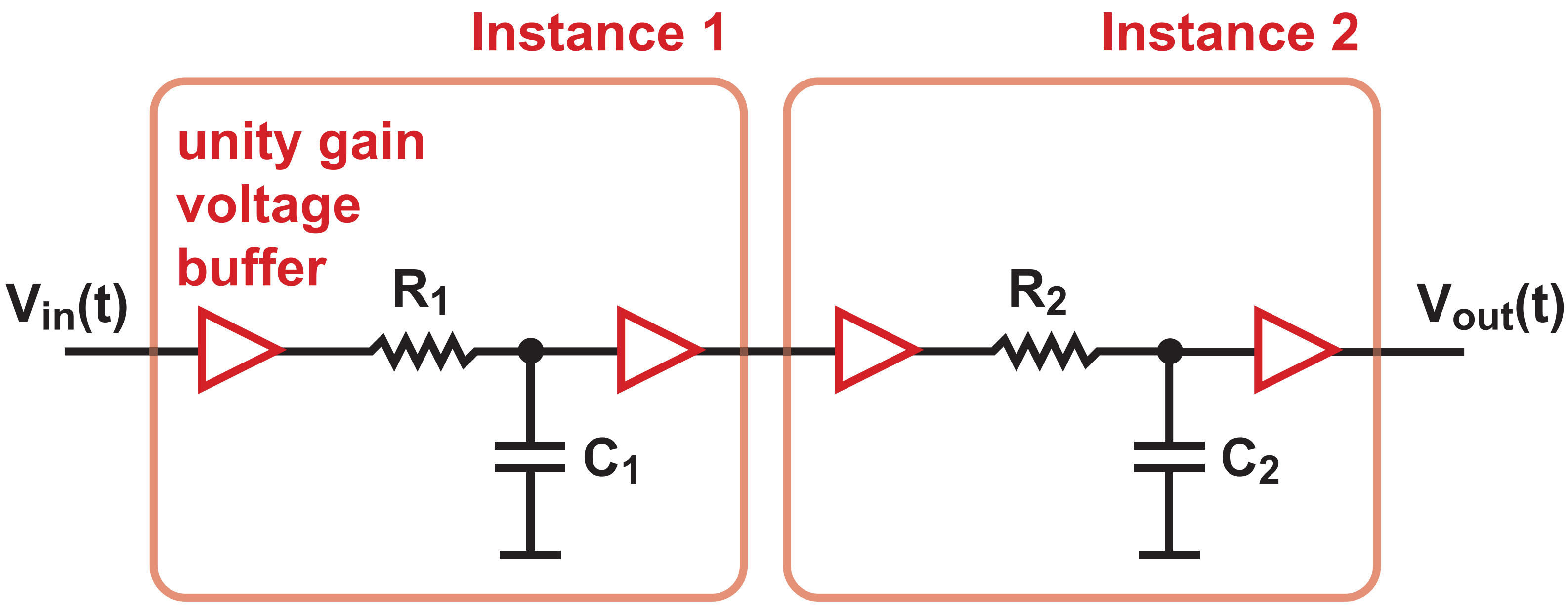

- **Resulting transfer function is *NOT* simply the cascade of two identical RC filters**
  - **Actual pole locations are influenced by mutual coupling of the two first-order RC networks**

## *Cascade of First Order RC Networks as Instances*




- This would appear to be the same as cascading the RC networks at the same level of hierarchy…

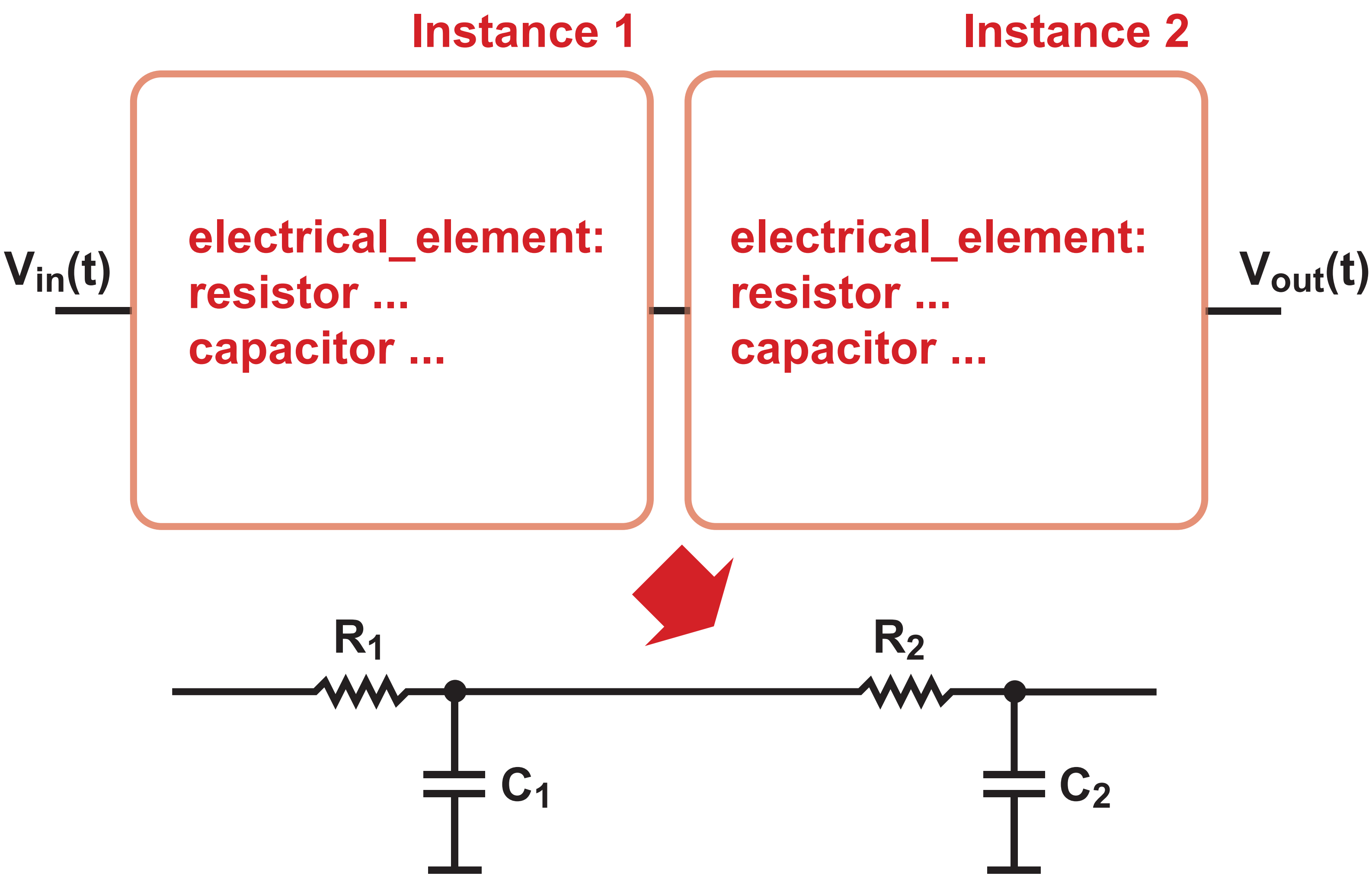
# *Recall Unity Gain Voltage Buffer Insertion*




- CppSim implicitly adds unity gain voltage buffers
  - Resulting transfer function is actually the cascade of two identical RC filters

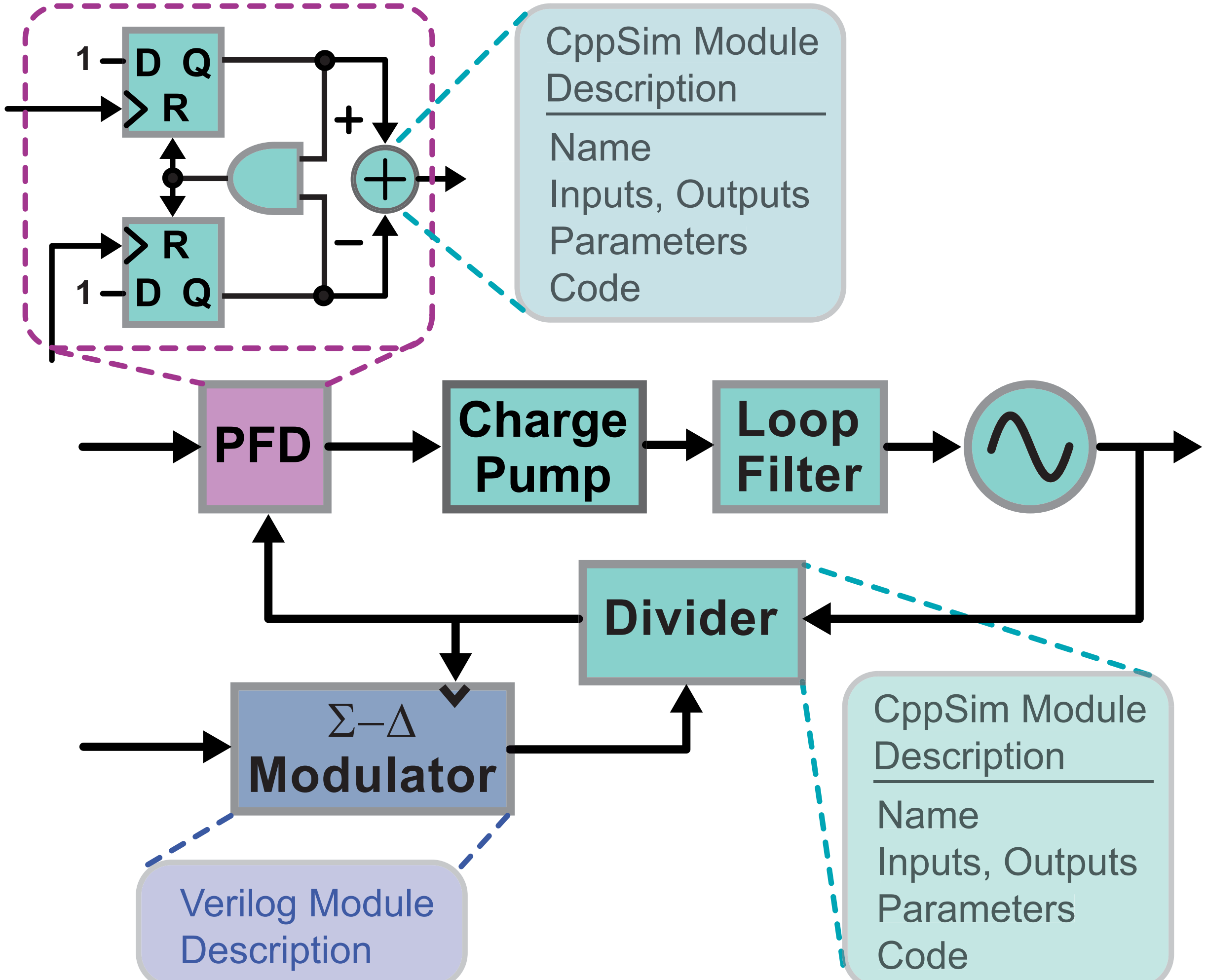
**How do you achieve network coupling with hierarchy?**

# *Electrical Element Modules Form Coupled Networks*




**CppSim allows one level of hierarchy for coupled networks**

# *Schematic Based Simulation using CppSim/VppSim*




- Schematic
  - Provides hierarchical description of *system topology*
- Code blocks
  - Specify *module behavior* using templated C++ code or Verilog code

- Designers graphically develop system based on a library of C++/Verilog symbols and code
  - Easy to create new symbols with accompanying code

## *CppSim versus VppSim*

- **CppSim**
  - **C++ is the simulation engine**
    - Verilog code translated into C++ classes using Verilator
  - **Best option when system simulation focuses on analog performance with digital support**
- **VppSim**
  - **Verilog is the simulation engine**
    - C++ blocks accessed through the Verilog PLI
  - **Best option when system simulation focuses on digital verification with C++ stimulus**

**Constant time step approach allows seamless connection between C++ and Verilog models**

# VppSim Example: Embed CppSim Module in NCVerilog

## CppSim module

```
module: leadlagfilter
parameters: double fz, double fp,
                  double gain
inputs: double in
outputs:  double out
static_variables:
classes: Filter filt("1+1/(2*pi*fz)s",
             "C3*s + C3/(2*pi*fp)*s^2",
             "C3,fz,fp,Ts",1/gain,fz,fp,Ts);
init:
code:
filt.inp(in);
out = filt.out;
```

## Resulting Verilog module

```
////// Auto-generated from CppSim module //////
module leadlagfilter(in, out);
  parameter fz = 0.00000000e+00;
  parameter fp = 0.00000000e+00;
  parameter gain = 0.00000000e+00;
  input in;
  output out;

  wreal in;
  real in_rv;
  wreal out;
  real out_rv;

  assign out = out_rv;

  initial begin
      assign in_rv = in;
    end

  always begin
      #1
      $leadlagfilter_cpp(in_rv,out_rv,fz,fp,gain);
    end
endmodule
```

# *EdgeDetect() versus timing_sensitivity: for VppSim*

## EdgeDetect (simplified)

```
////// Auto-generated from CppSim module //////
module dig_mod(a,b,clk,y,r);


  always begin
     #1
     $dig_mod_cpp(a,b,clk,y,r);
  end
endmodule
```

- PLI routine is called *every* time step
  - Dramatically slows down VppSim!

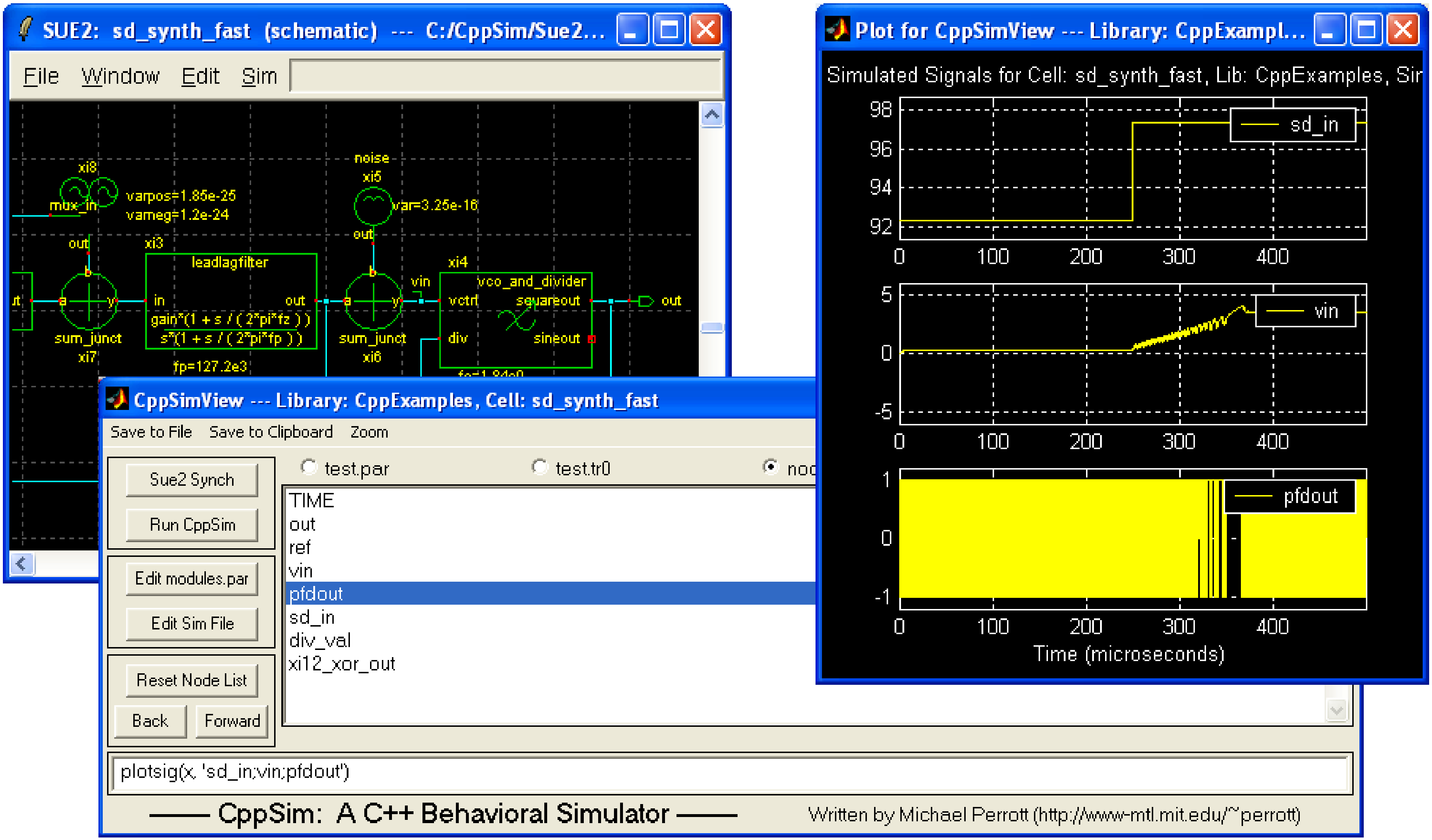
## timing_sensitivity:

```
////// Auto-generated from CppSim module //////
module dig_mod(a,b,clk,y,r);


  always@(posedge clk) begin
     $dig_mod_cpp(a,b,clk,y,r);
  end

endmodule
```

- PLI routine is only called on positive clk edges
  - Much less impact on simulation speed

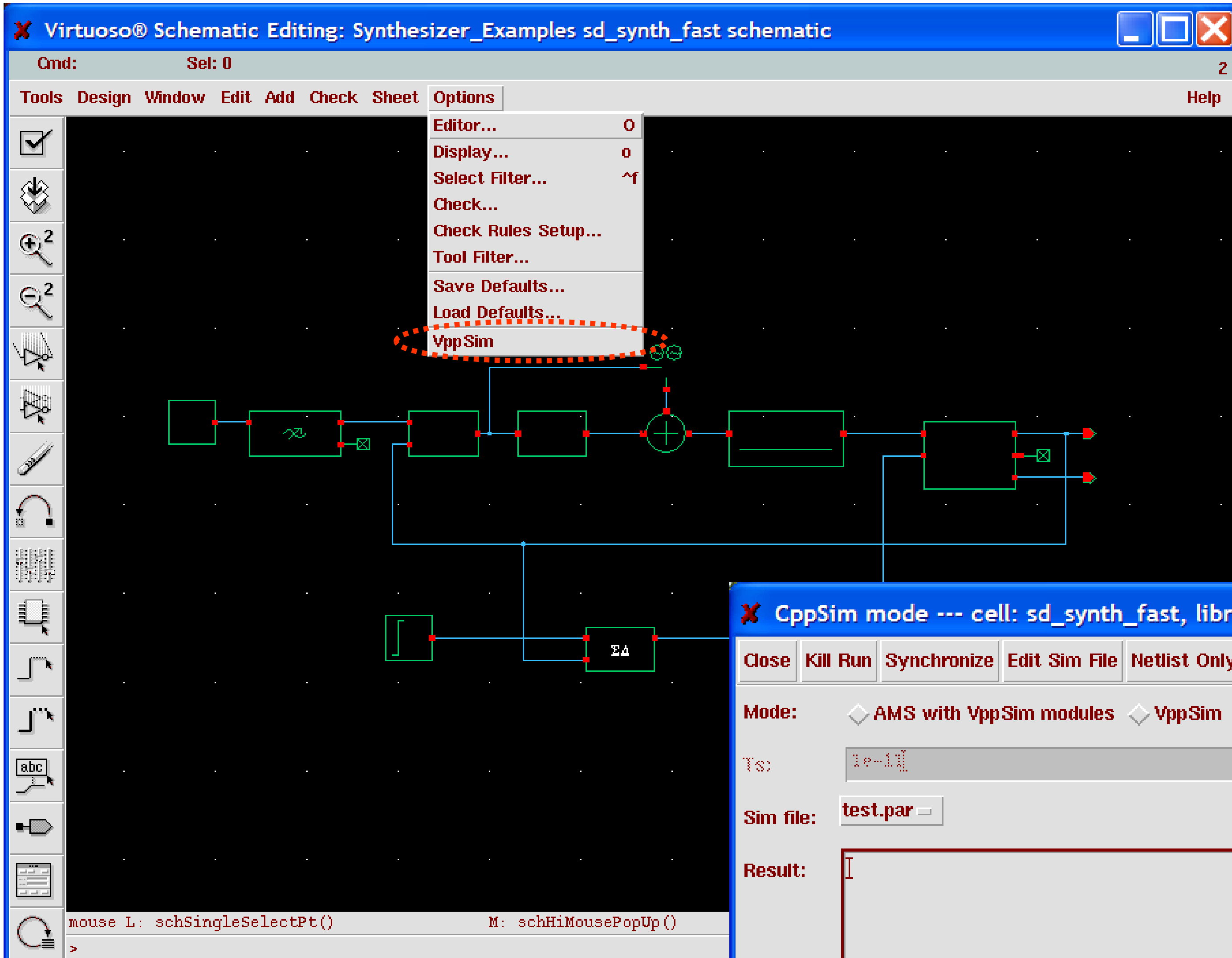
**Use timing_sensitivity: unless you need to perform computation during every time step (Note: no penalty for EdgeDetect method in CppSim)**

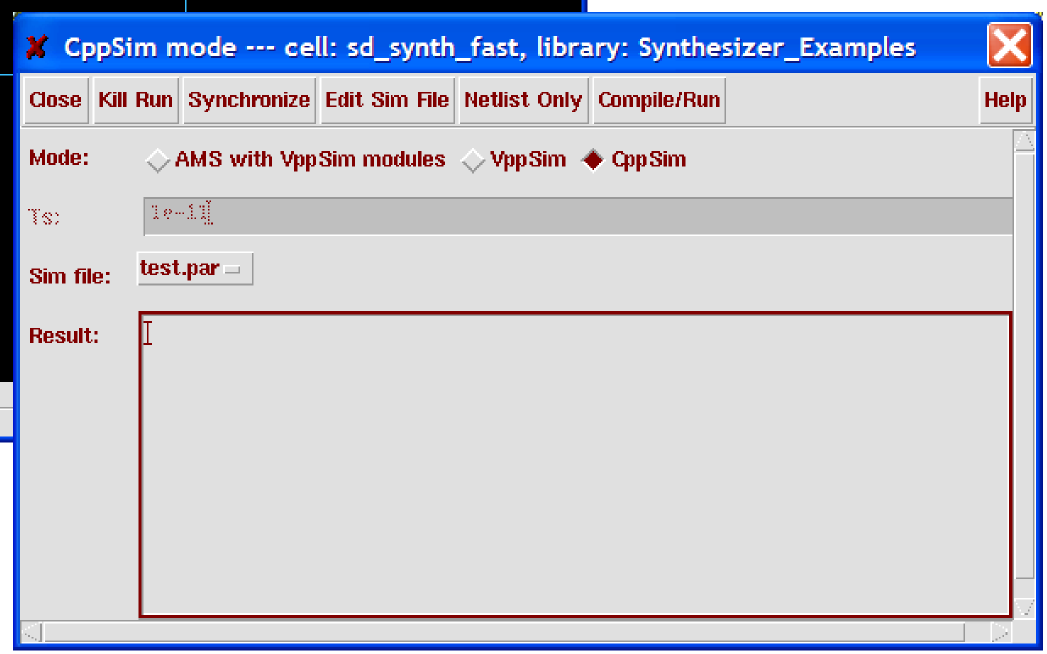
# Screenshot of CppSim/VppSim (Windows Version)




**Readily Interfaces with Matlab and GTKWave**

# Screenshot of CppSim/VppSim (Cadence Version)




**Interfaces with Matlab, GTKWave, and SimVision**

# Free Download at www.cppsim.com

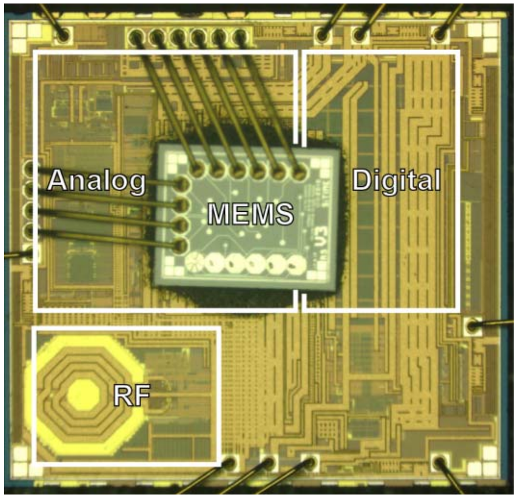
## *Many Tutorials Available for CppSim/VppSim*

- **Wideband Digital fractional-N frequency synthesizer**
- **VCO-based Analog-to-Digital Convertor**
- **GMSK modulator**
- **Decision Feedback Equalization**
- **Optical-Electrical Downversion and Digitization**
- **OFDM Transceiver**
- **C++/Verilog Co-Simulation**

➡ **See http://www.cppsim.com**

# *Example Benchmarks for a Full Chip Simulation*

## Tabulated simulation times for a MEMS-based oscillator:




- **SPICE-level model**
  - **Checking of floating gate, over-voltage, startup of bandgap and regulators, etc.**
    - **Spectre Turbo: 2 microseconds/day**
    - **BDA: 8 microseconds/day**
- **Architectural model using CppSim**
  - **Examination of noise and analog dynamics**
    - **2.8 milliseconds/hour**
- **Verification model using VppSim**
  - **Validation of digital functionality in the context of analog control and hybrid digital/analog systems**
    - **7 milliseconds/minute**

## *Conclusion*

- **CppSim is designed for high productivity and versatility**
  - **Easy to create your own code blocks**
    - Use existing modules to see examples, but don't limit yourself to what is available
  - **Allows very detailed modeling of complex circuits**
    - You are not confined to an overly simplified model
  - **Invites a scripted approach to running simulations**
    - Excellent integration with Matlab/Octave
  - **Runs in Windows, Mac OS X, or within Cadence**
    - Has been used to simulate entire ICs in Cadence

- **Extensive 10 year track record of enabling new circuit architectures with first chip success**