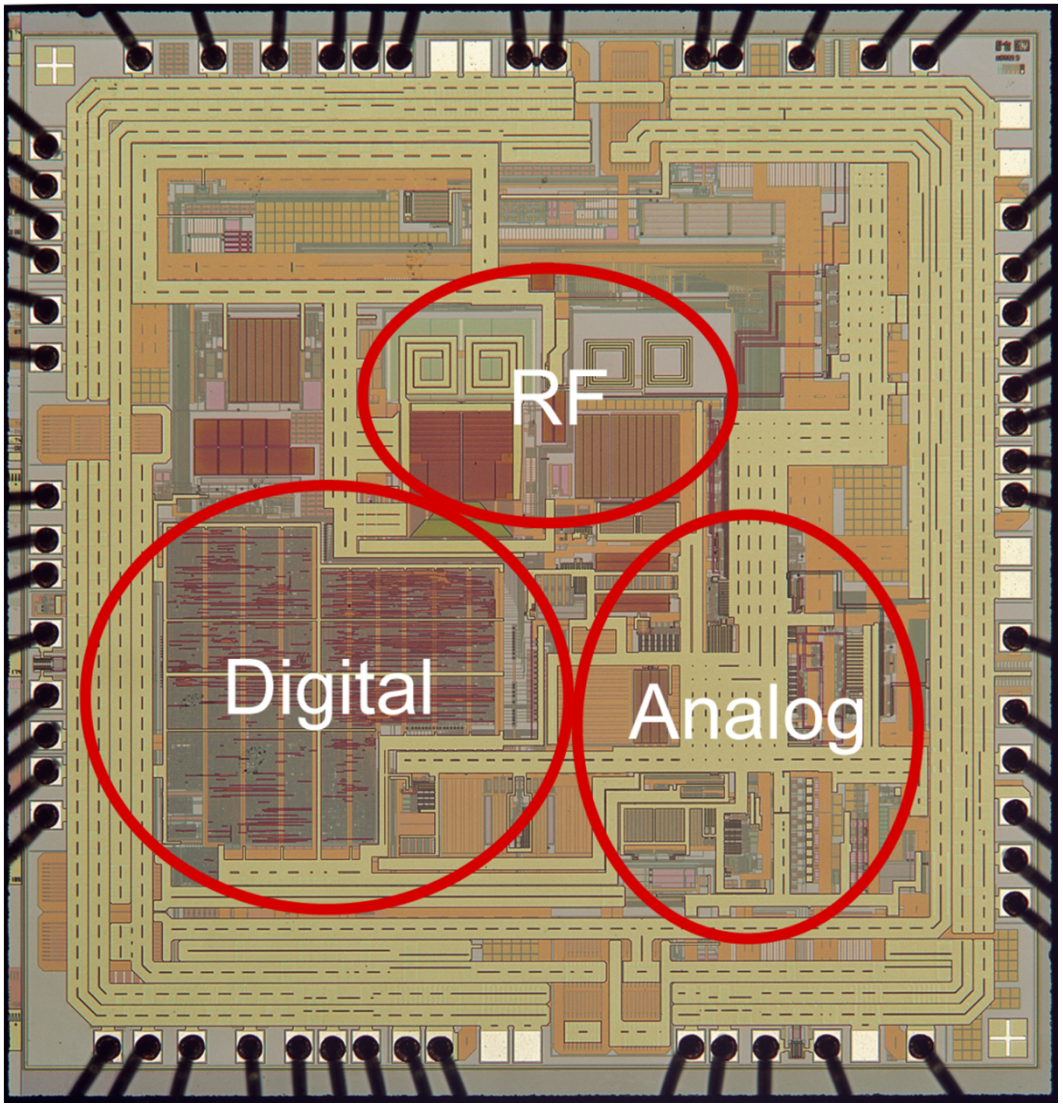
Fast and Accurate Behavioral Simulation of Time-Based Circuits using C++ and Standard Verilog

Dallas IEEE CAS Workshop 2010

Michael Perrott

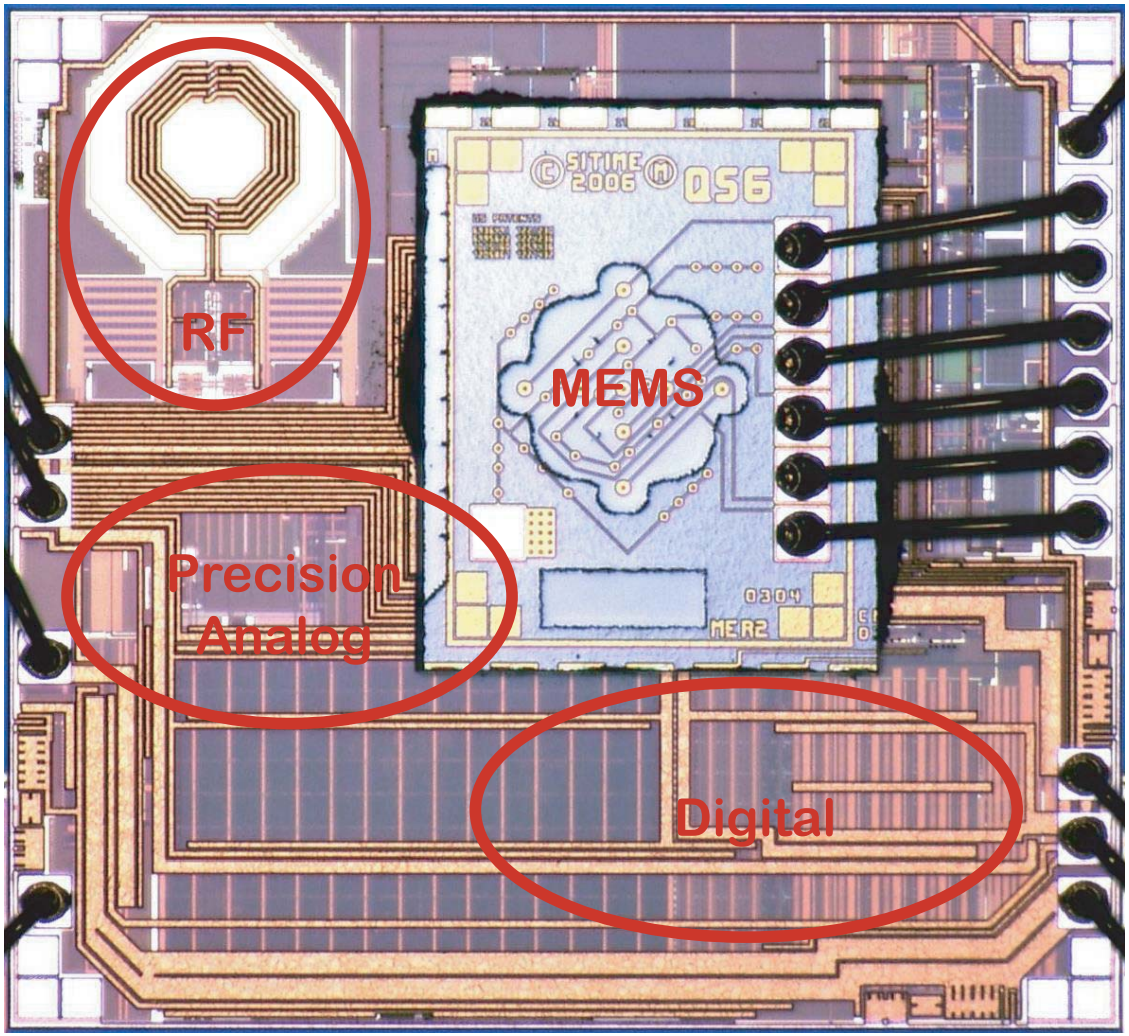
October 2010

A Modern "Analog" Custom IC



- A 2.5 Gb/s CDR for high speed links
  - *Analog*
    amplification and phase sensing
  - *Digital*
    filtering and calibration
  - *RF*
    clocking (2.5 GHz)
- How do we design such chips?
  - Standard methodologies do not provide a cohesive system approach

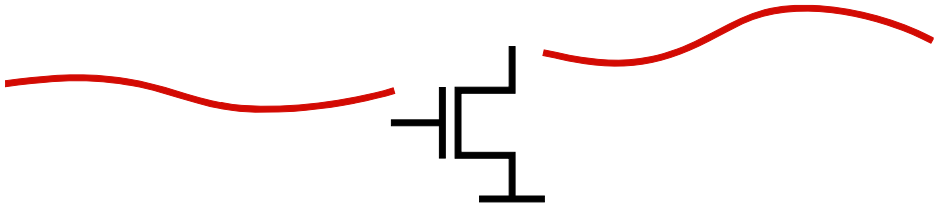
IC Design is Getting More Complex



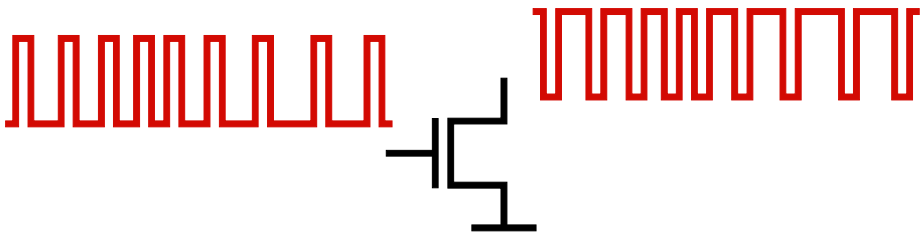
- **A Programmable MEMS Oscillator**
  - ***Analog***
    Temperature sensor, ADC, oscillator sustaining circuit
  - ***Digital***
    signal processing
  - ***RF***
    clocking (800 MHz)
  - ***MEMS***
    high Q resonator
- **System level design is critical**

Circuit Architectures are Changing

- **Traditional analog circuits utilize voltage and current**



  - **Modern CMOS processes have issues with voltage headroom, intrinsic gain ($g_m r_o$), and leakage**
- **Time-based circuits utilize the timing of edges produced by "digital" circuits**



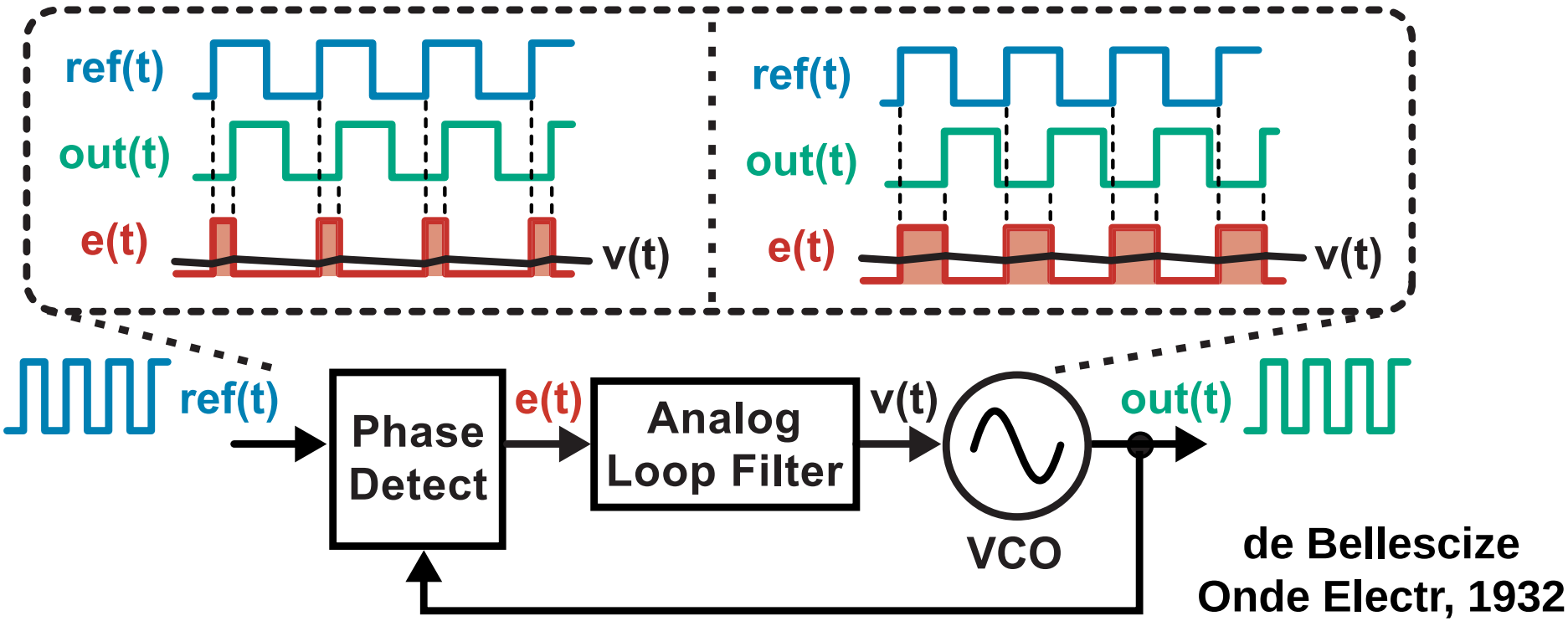
  - **Modern CMOS processes are offering faster edge rates and lower delay through digital circuits**

How do we design such circuits within an overall system context?

Focus: System Level Design of Time-Based Circuits

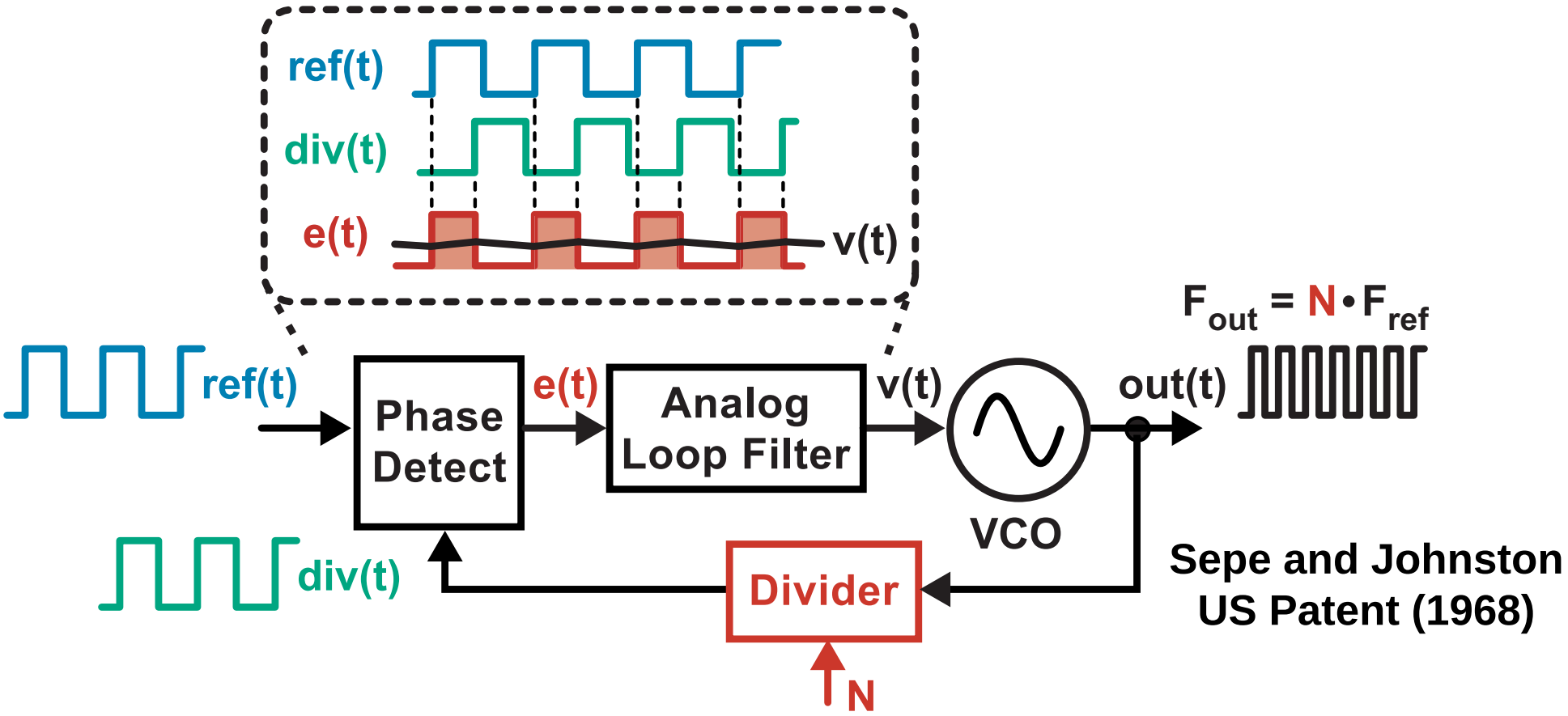
- We will begin by taking a closer look at time-based circuits
  - Background and some recent examples
  - Techniques for fast and accurate behavioral simulation
- We will then consider the issue of system level simulation
  - Where it fits within an overall IC design methodology
  - C++ versus Verilog as a system level simulator
  - Combined C++/Verilog simulation using CppSim/VppSim
- We will conclude with a short case study of a MEMS-based oscillator

Phase-Locked Loops are Classical Time-Based Circuits



- **VCO efficiently provides oscillating waveform with variable frequency**
- **PLL synchronizes VCO frequency to input reference frequency through feedback**
  - **Key block is phase detector**
    - Realized as *digital gates* that create pulsed signals

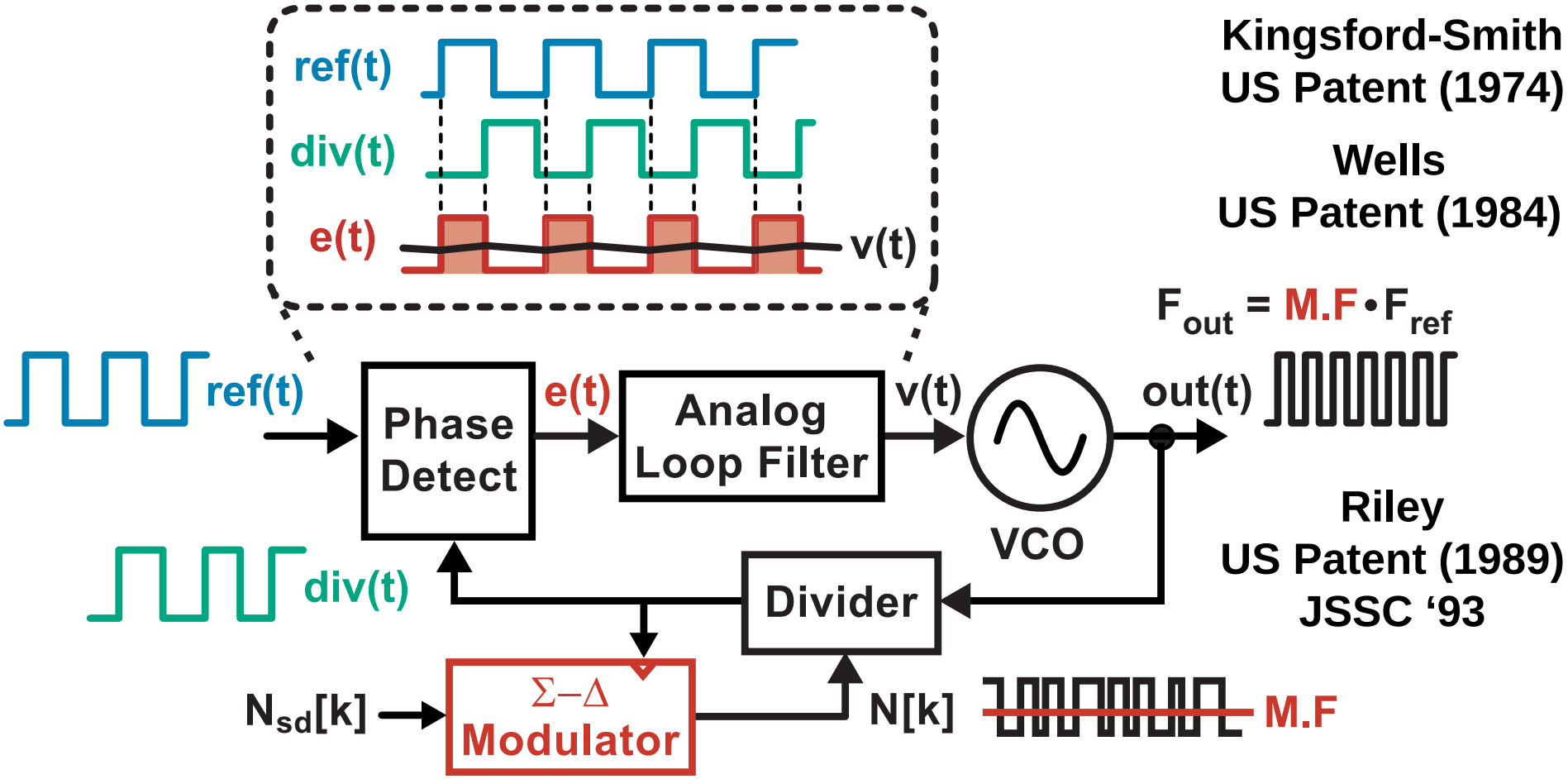
Integer-N Frequency Synthesizers



- Use digital counter structure to divide VCO frequency
  - Constraint: must divide by integer values
- Use PLL to synchronize reference and divider output

Analog output frequency is digitally controlled

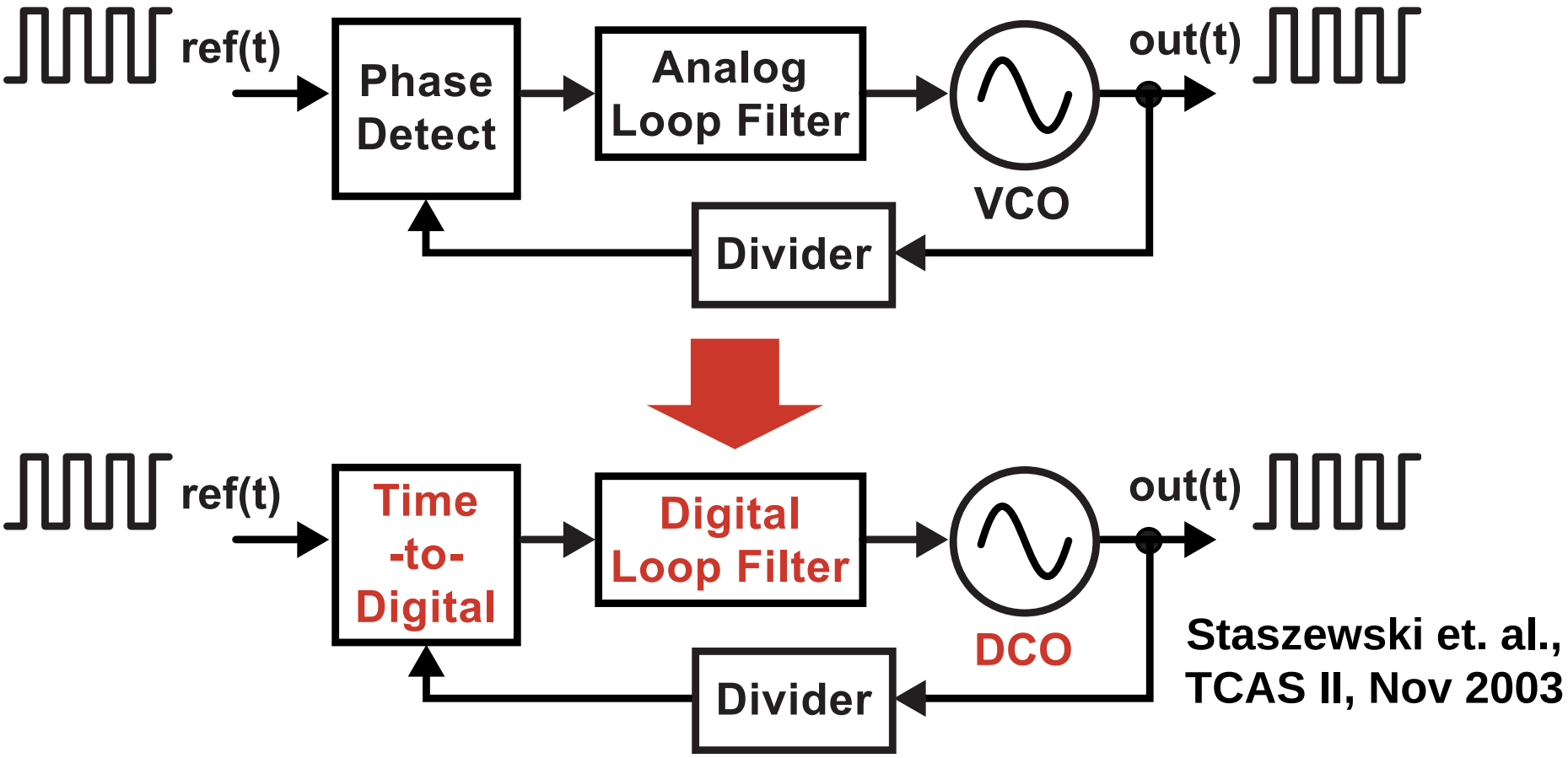
Fractional-N Frequency Synthesizers



- Dither divide value to achieve fractional divide values
  - PLL loop filter smooths the resulting variations

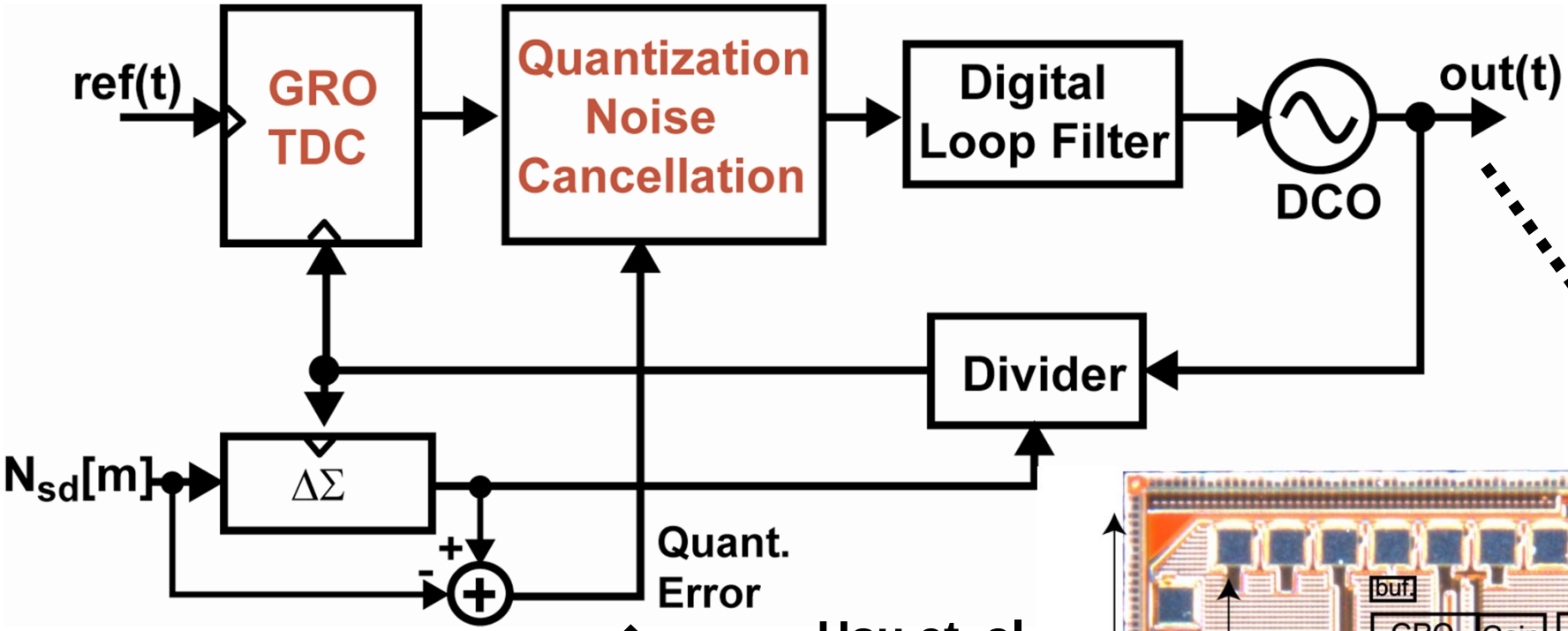
Very high frequency resolution is achieved

Recent Trend: Move to a More Digital Implementation

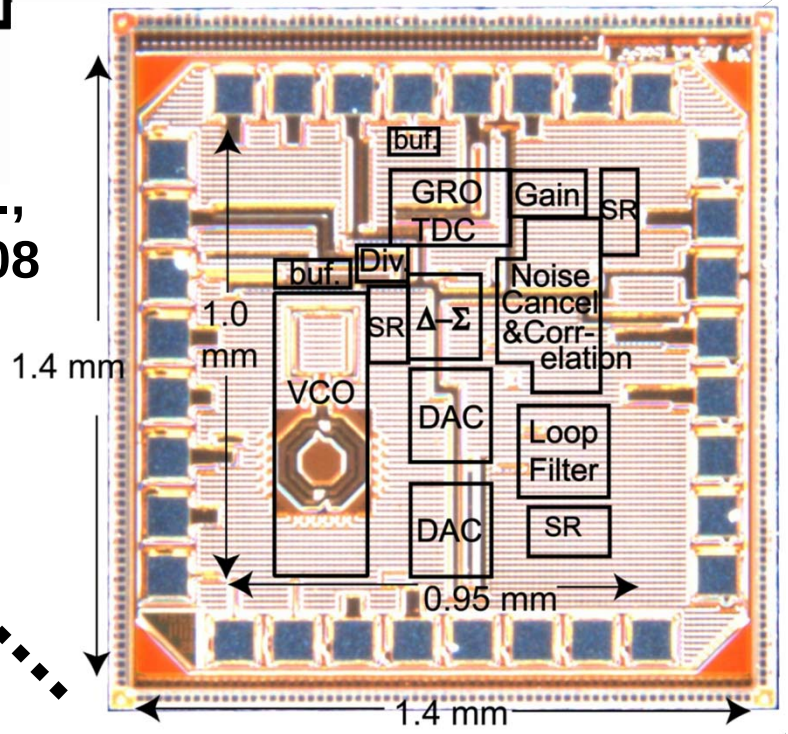


- Digital loop filter: compact area, digital flow
- Key insight: faster CMOS processes allow faster edges, lower delays, and overall improved time resolution
  - Allows us to leverage Moore's law for improving performance

Improved TDC and Noise Cancellation Lowers Jitter

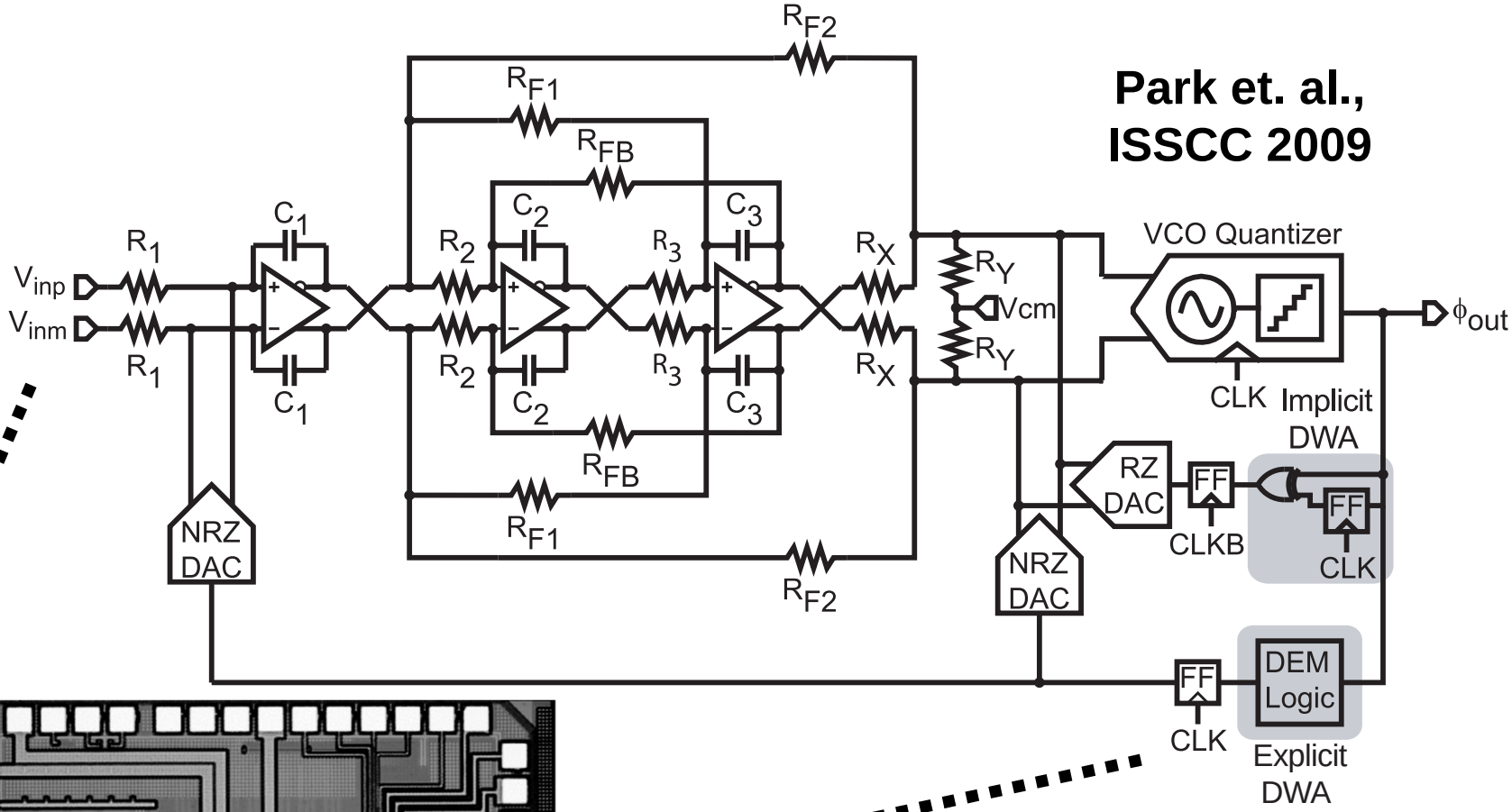


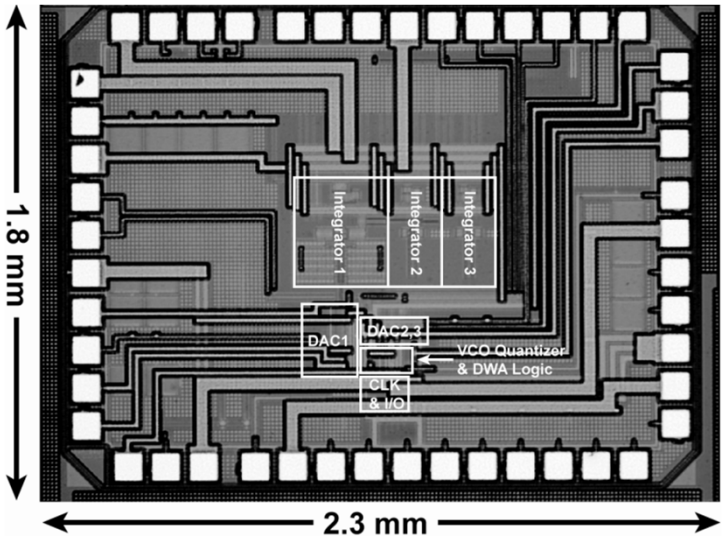
Hsu et. al., ISSCC 2008



- **< 250 fs (rms) integrated phase noise (1kHz to 40 MHz)**
- **Highly digital implementation**

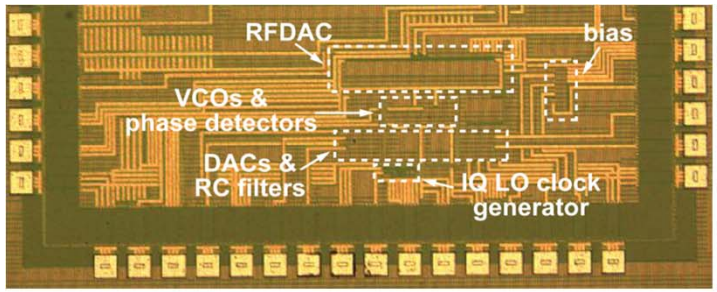
VCO-Based ADCs Use Time to Achieve High Resolution



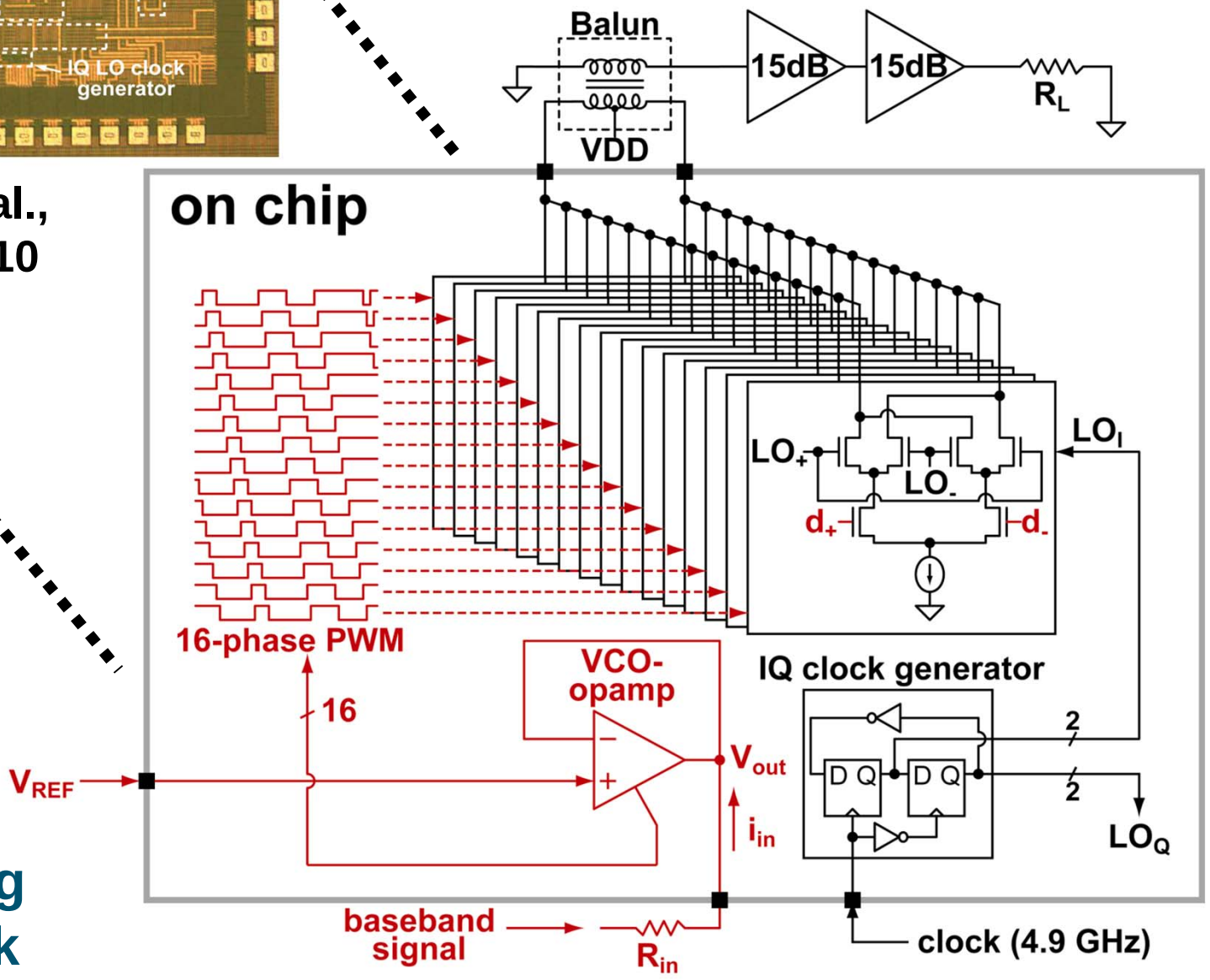


- **Peak SNDR of 78 dB with 20 MHz bandwidth**
- **Figure of merit: 330 fJ/step**

Mult-Phase PWM Enables Efficient RF modulator



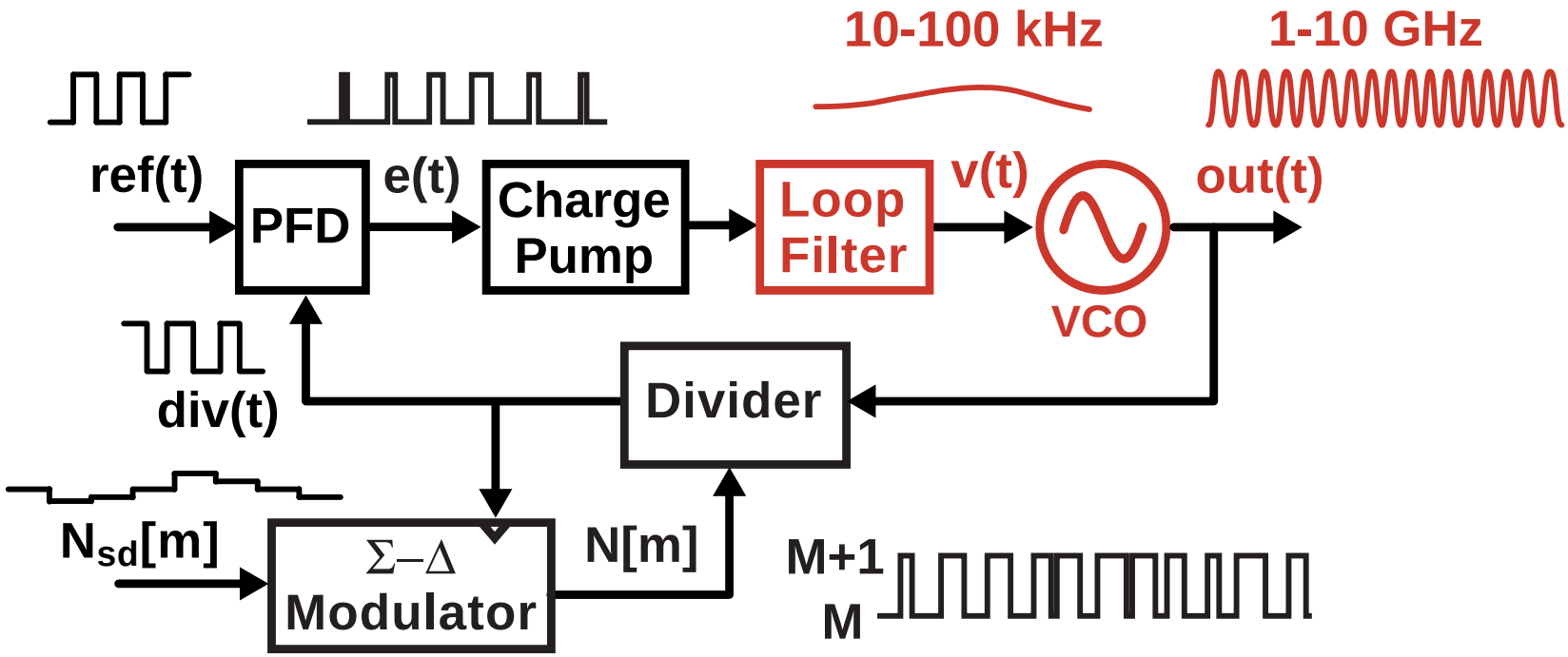
Park et. al.,
RFIC 2010



- Meets 802.11g spectral mask

Issues with Behavioral Simulation of Time-Based Circuits

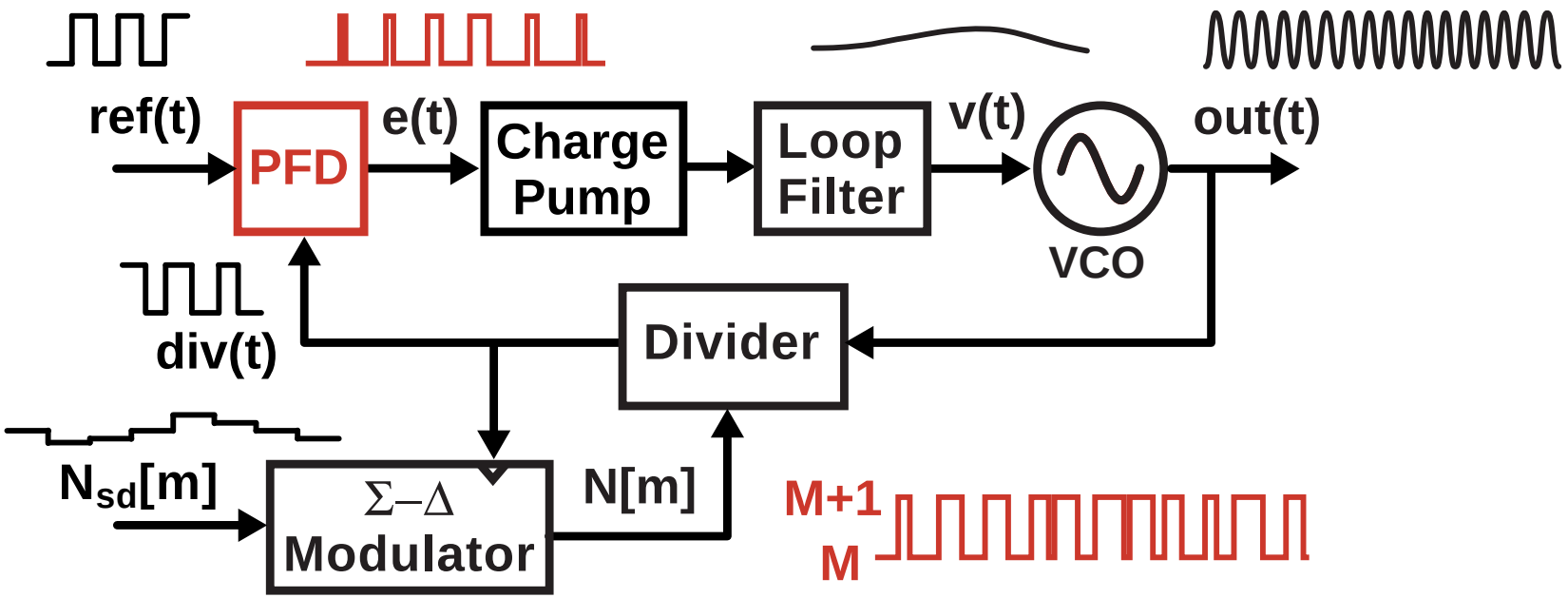
- **Consider a fractional-N synthesizer as a prototypical time-based circuit**



- **High output frequency ⇨ High sample rate**
- **Long time constants ⇨ Long time span for transients**

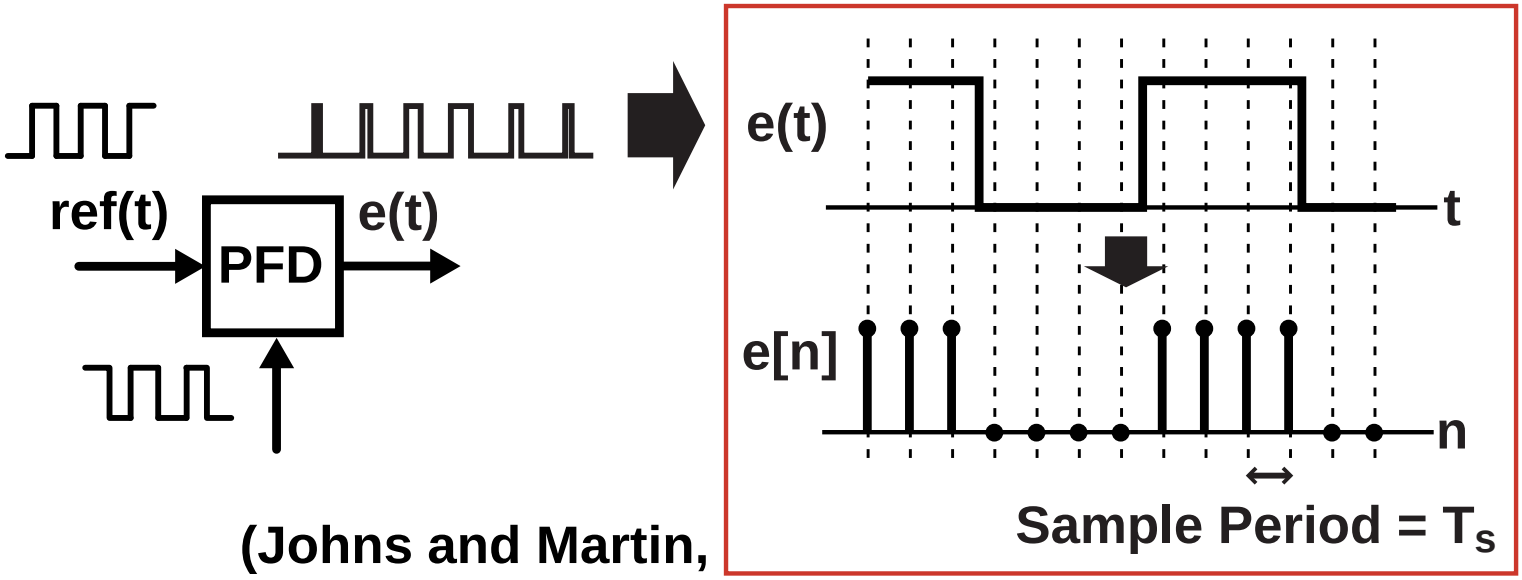
Large number of simulation time steps required

Continuously Varying Edge Times Create Accuracy Issues



- PFD output is not bandlimited
  - PFD output must be simulated in discrete-time
- Phase error is inaccurately simulated
- Non-periodic dithering of divider complicates matters

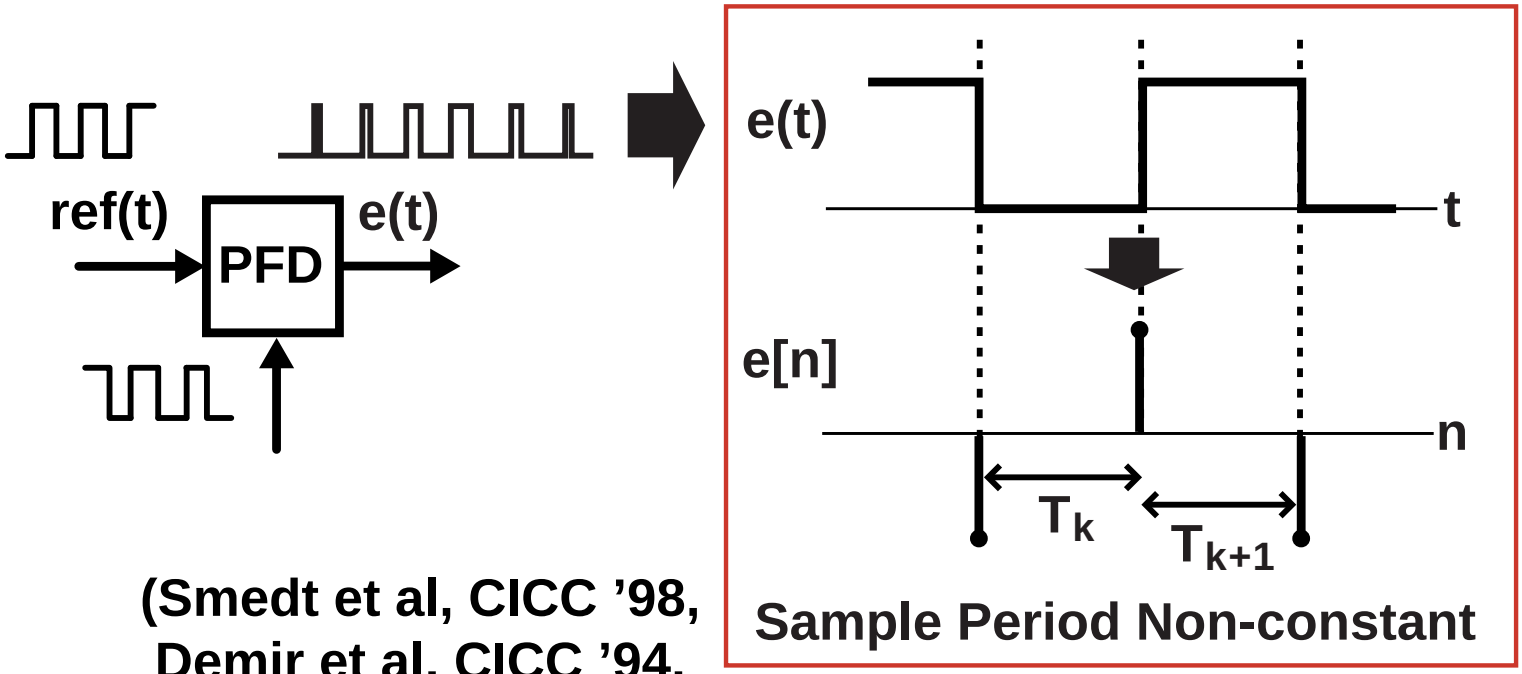
Example: Classical Constant-Time Step Method



(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 overwhelms the PLL noise sources we are trying to simulate

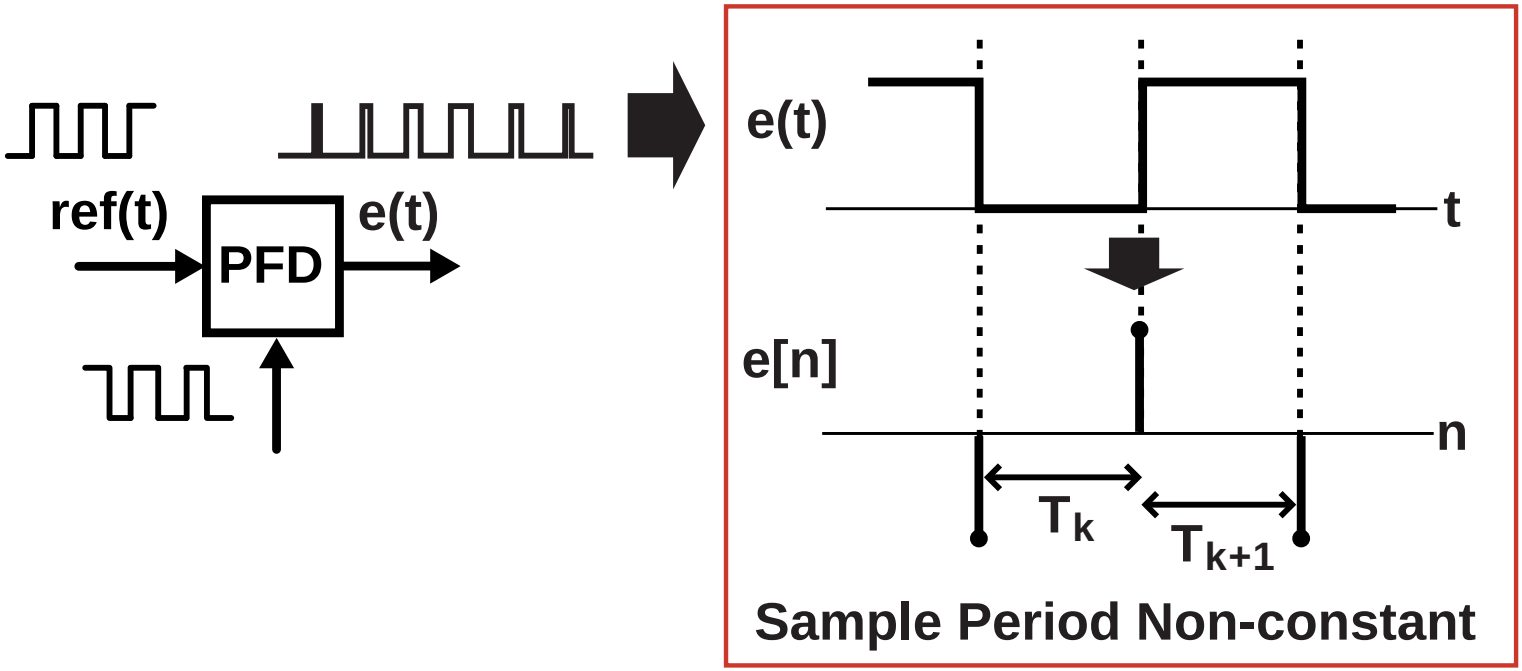
Alternative: Event Driven Simulation



**(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!**

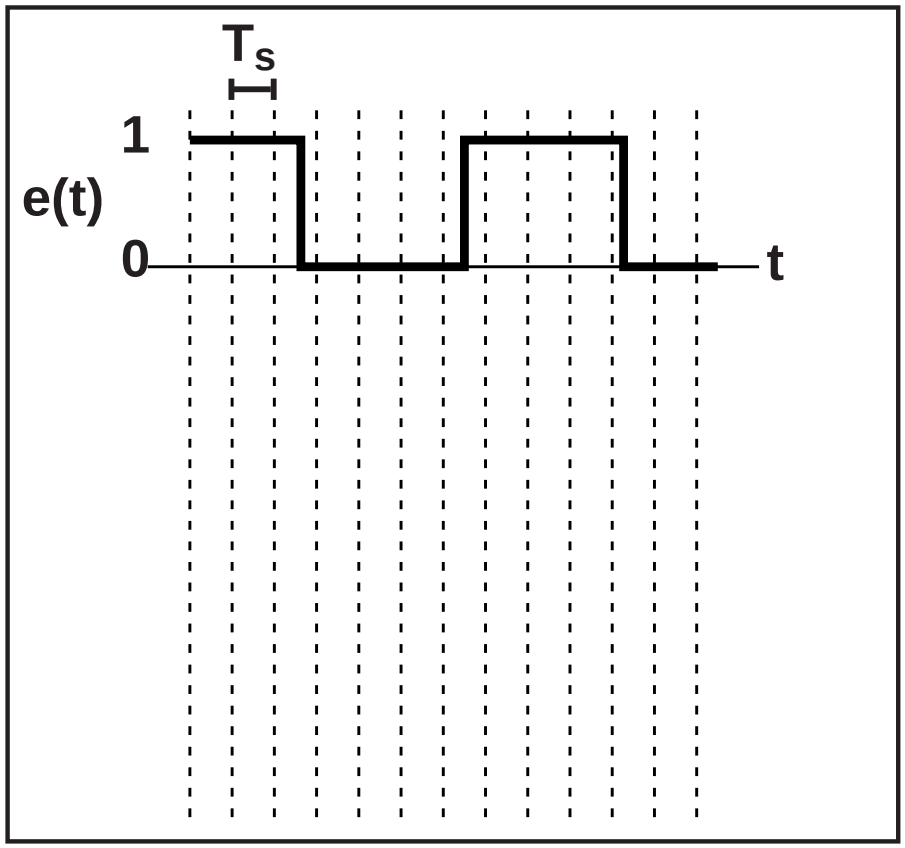
Issue: Non-Constant Time Step Brings Complications

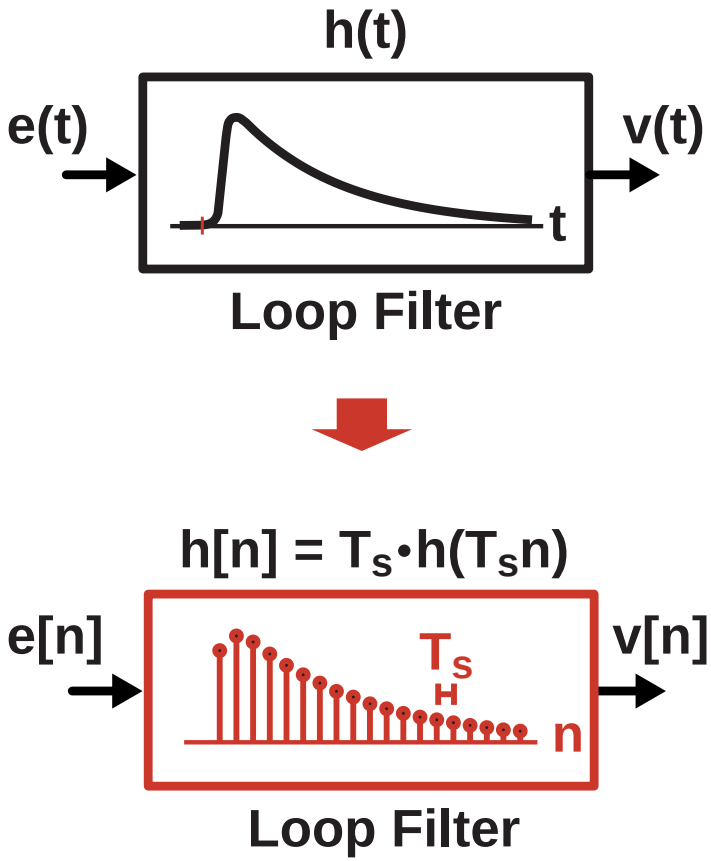


- **Filters and noise sources must account for varying time step in their code implementations**
- **Spectra derived from mixing and other operations can display false simulation artifacts**
- **Setting of time step becomes progressively complicated if multiple time-based circuits simulated at once**

Is there a better way?

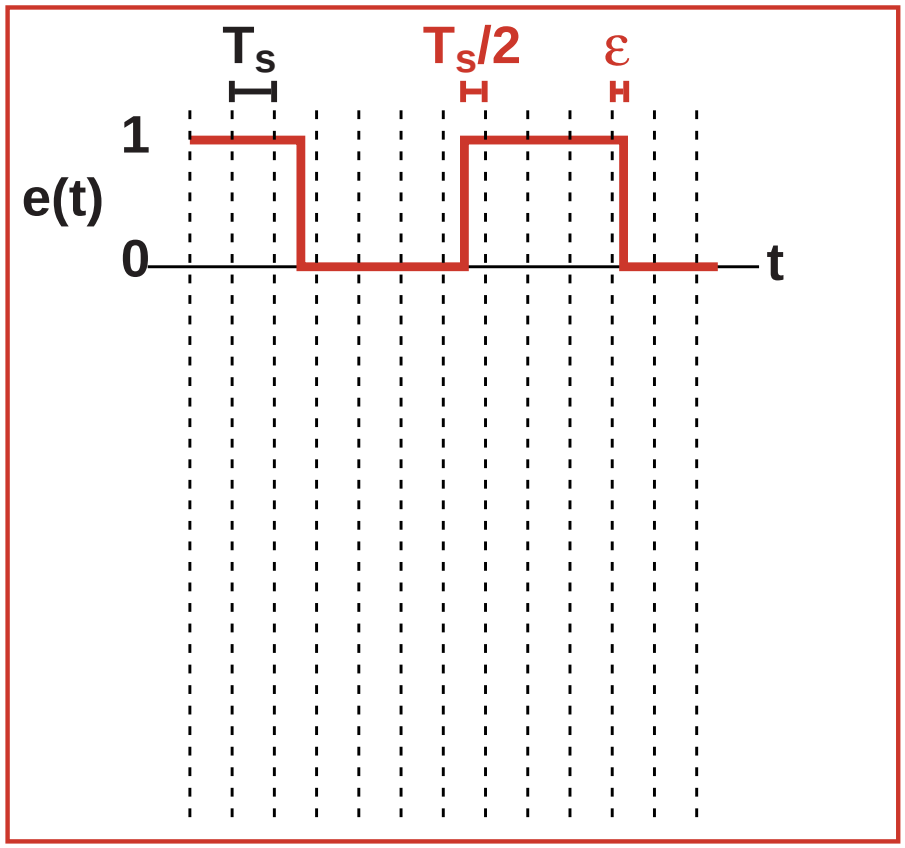
Proposed Approach: Use Constant Time Step

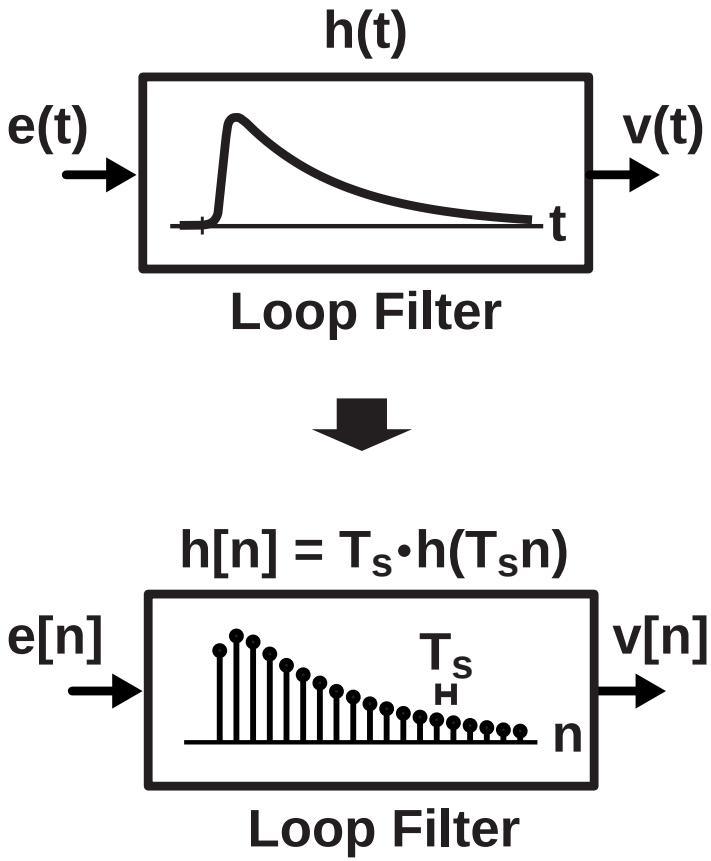




- Straightforward CT to DT transformation of filter blocks
  - Use bilinear transform or impulse invariance methods
- Overall computation framework is fast and simple
  - Simulator can be based on Verilog, Matlab, C++

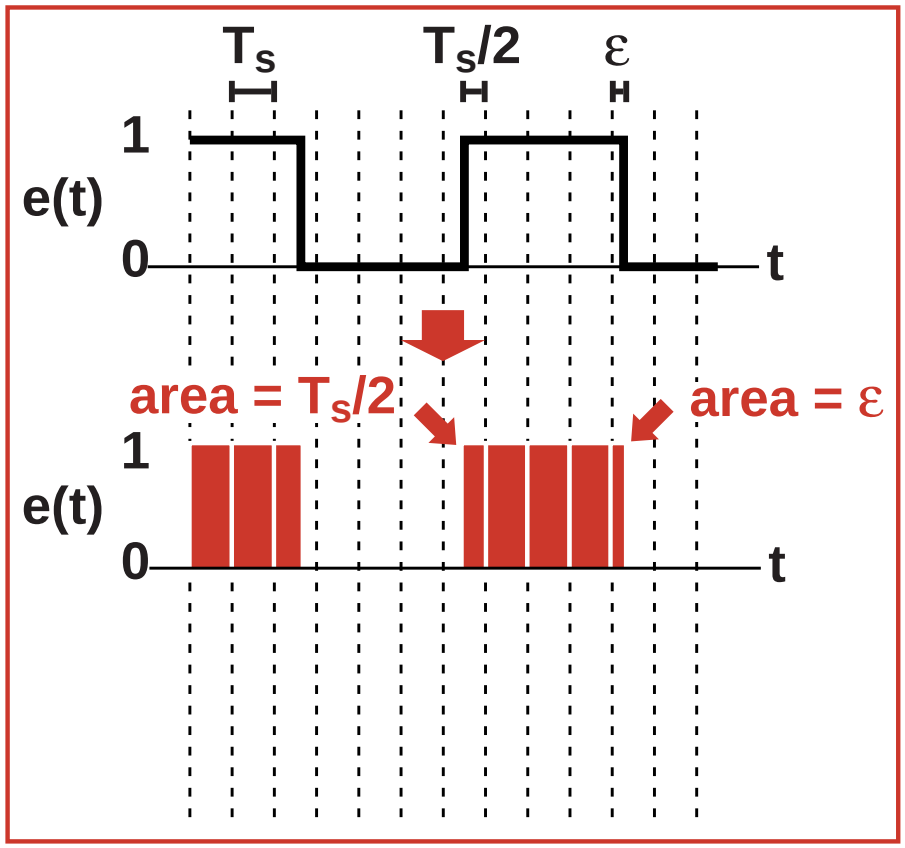
Problem: Quantization Noise at PFD Output

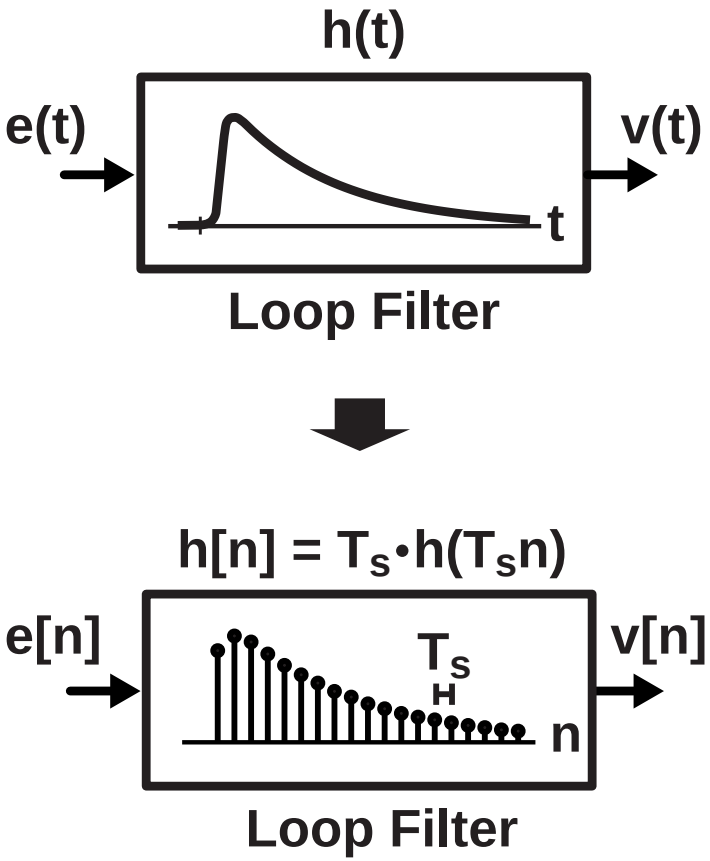




- Edge locations of PFD output are quantized
  - Resolution set by time step: T_s
- Reduction of T_s leads to long simulation times

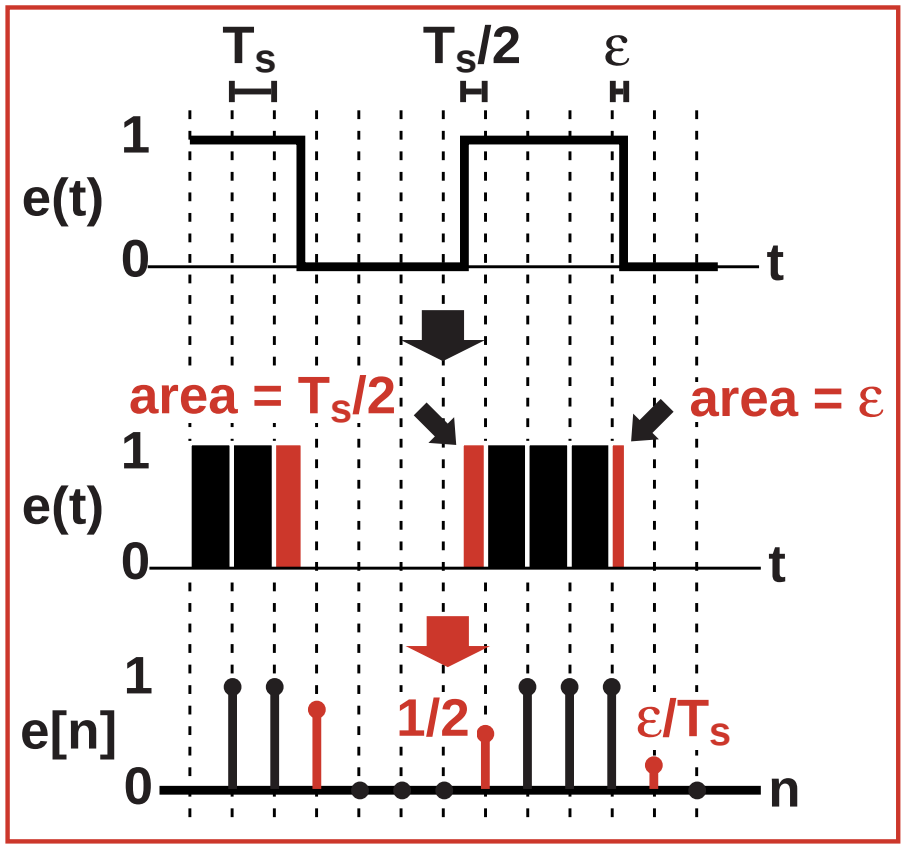
Proposed Approach: View as Series of Pulses

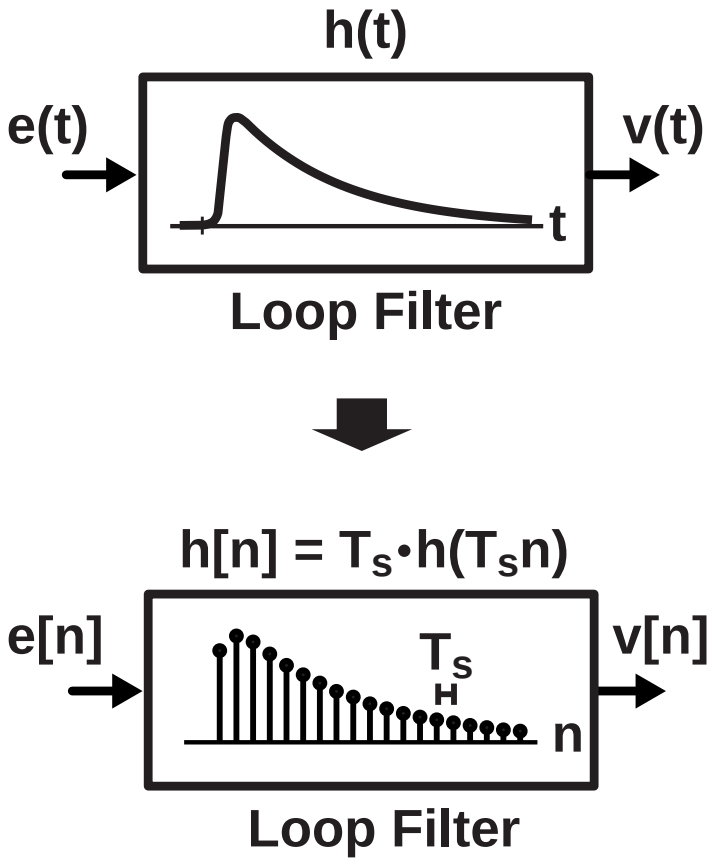




- Area of each pulse set by edge locations
- Key observations:
  - Pulses look like impulses to loop filter
  - Impulses are parameterized by their area and time offset

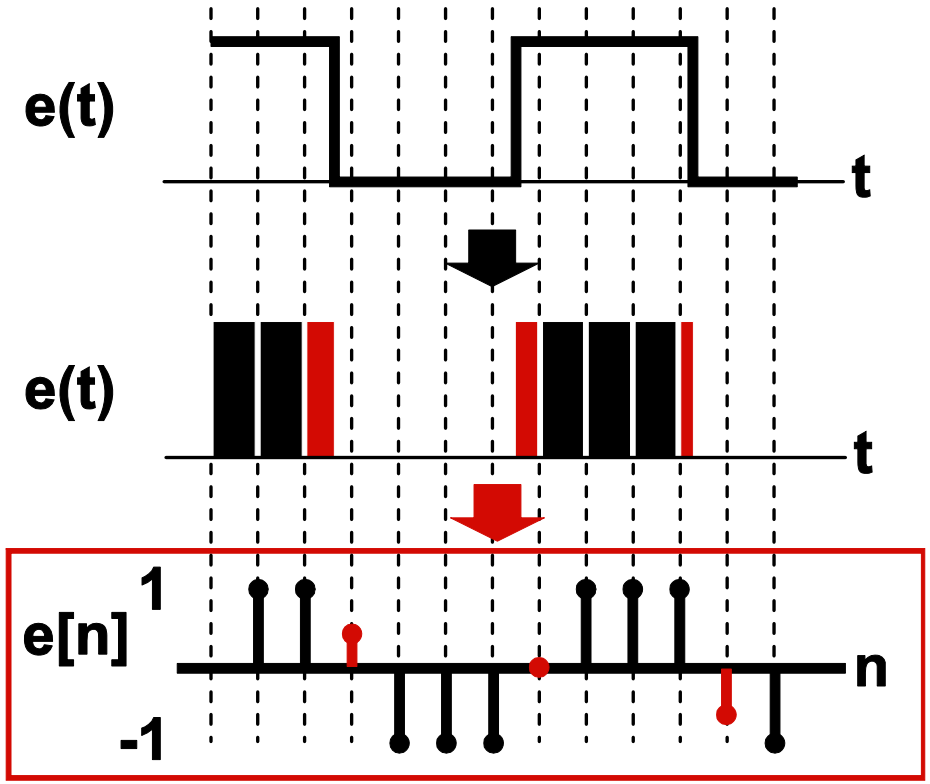
Proposed Area Conservation Method





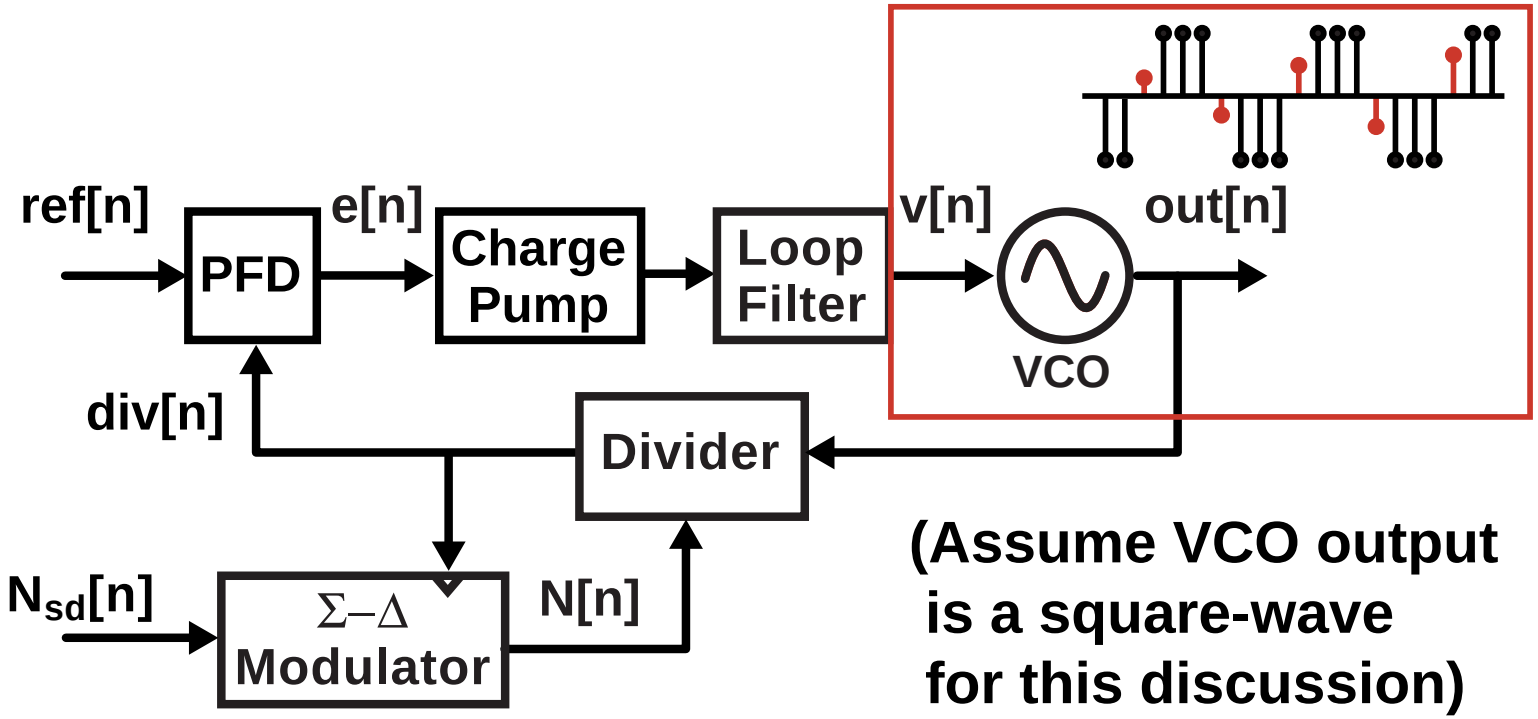
- Set e[n] samples according to pulse areas
  - Leads to very accurate results
  - Fast computation

Double_Interp Protocol



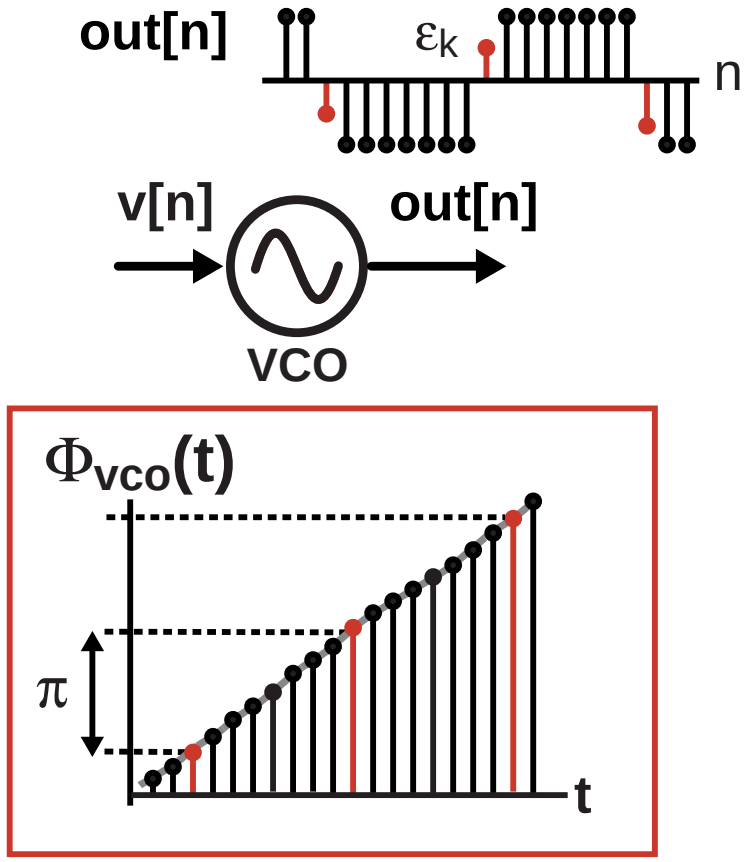
- **Protocol sets signal samples to -1 or 1 except for transitions**
  - **Transition values between -1 and 1 are directly related to the edge time location**
  - **Can be implemented in C++, Verilog, and Matlab/Simulink**

VCO is a Key Block for Double_Interp Encoding



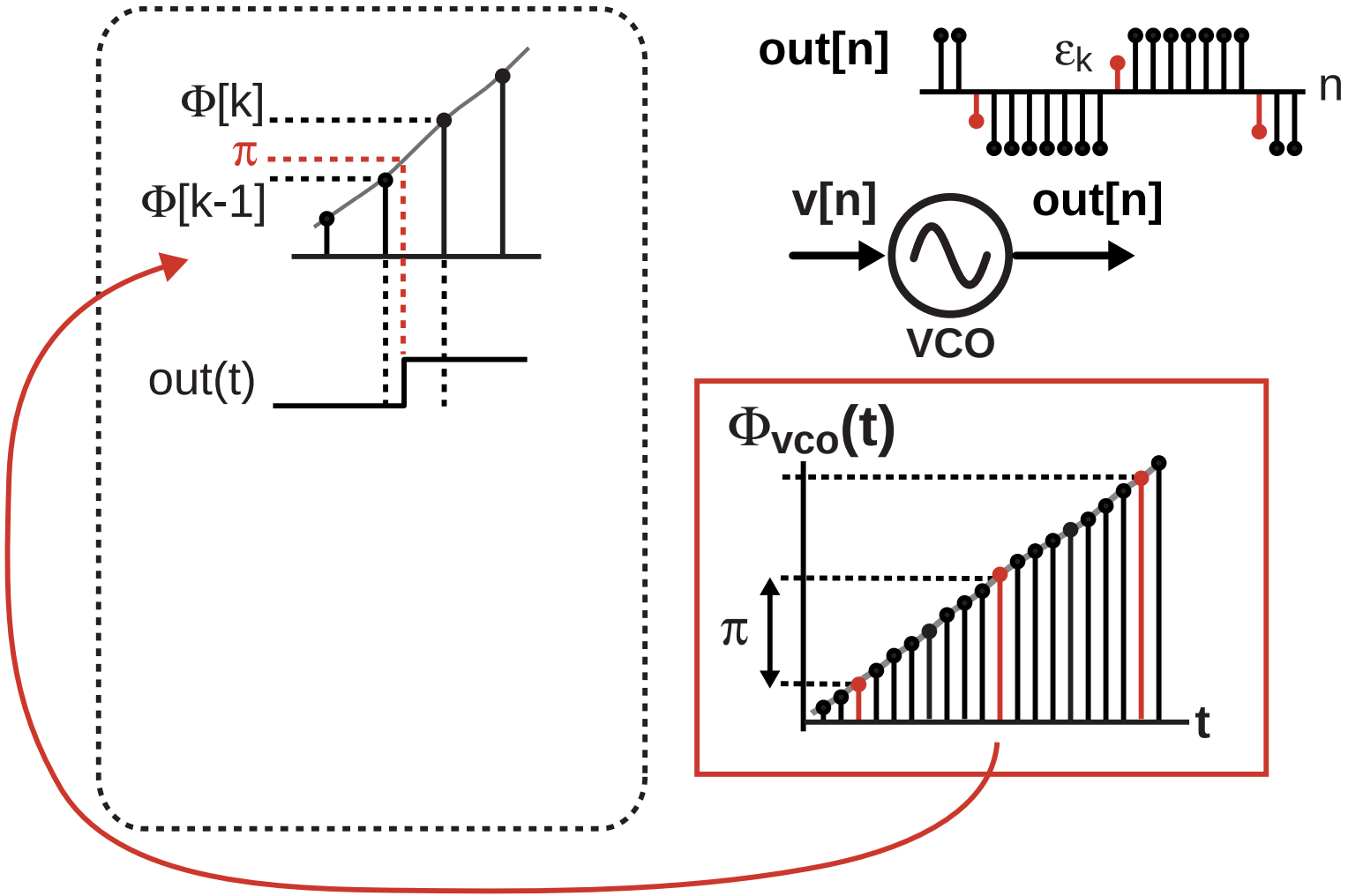
- The VCO block is the key translator from a bandlimited analog input to an edge-based waveform
  - We can create routines in the VCO that calculate the edge times of the output and encode their values using the double_interp protocol

Calculation of Transition Time Values



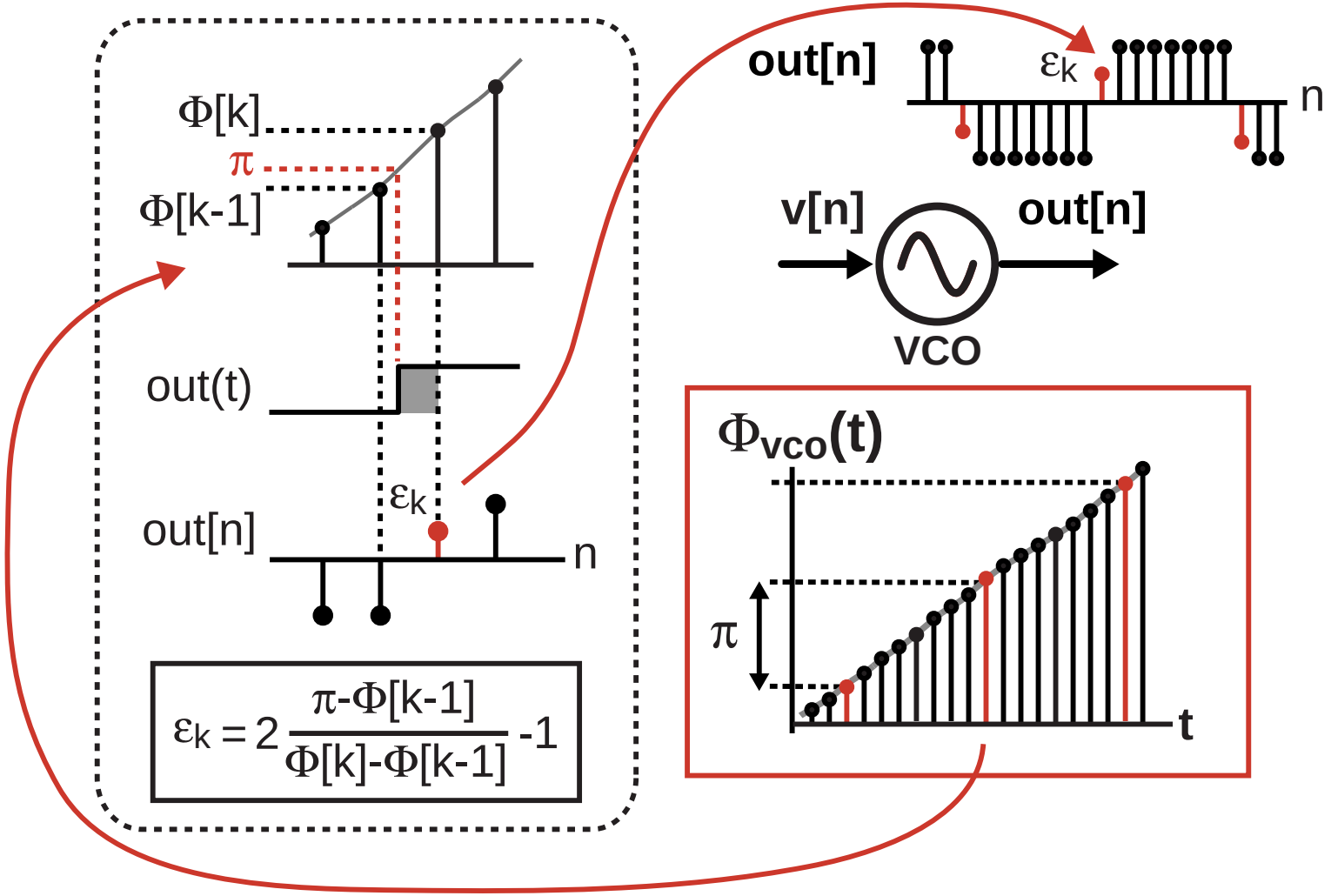
- **Model VCO based on its phase**

Calculation of Transition Time Values (cont.)



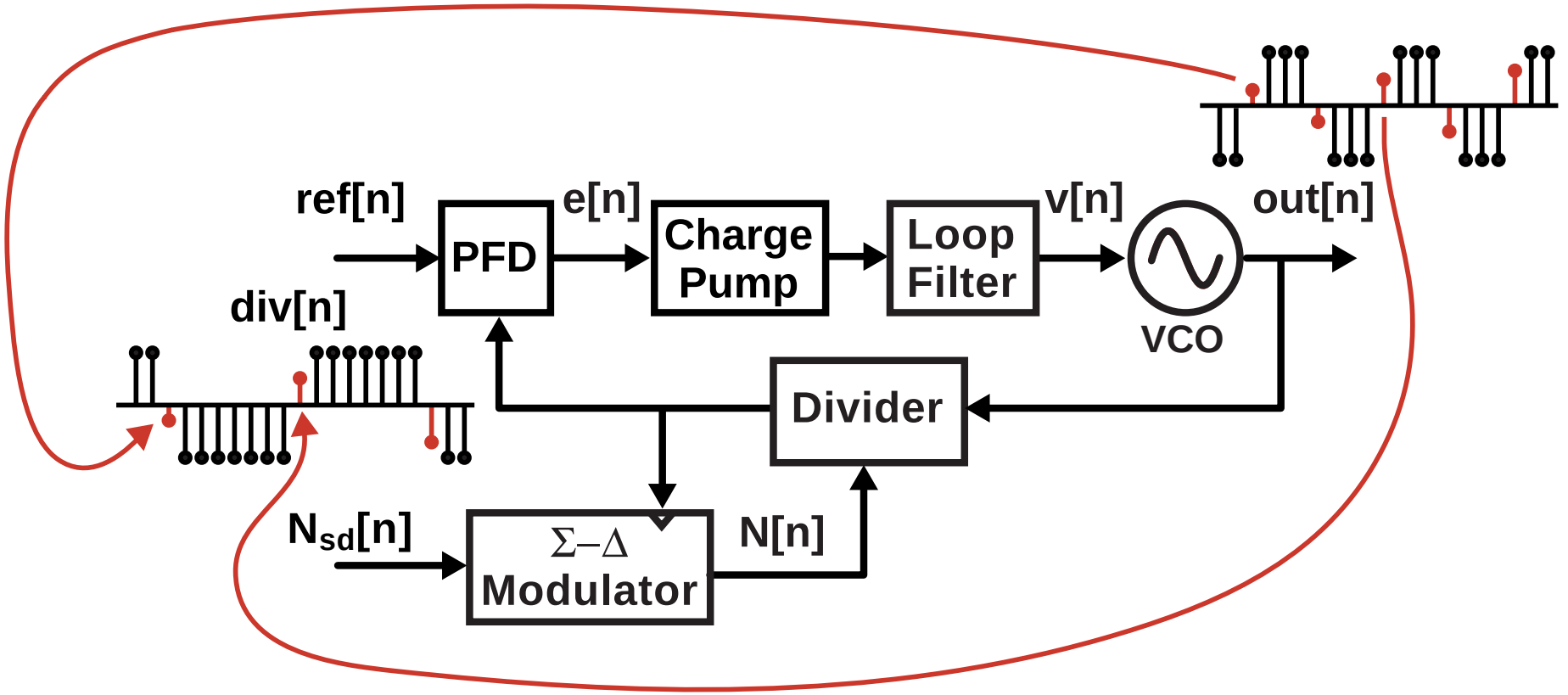
- **Determine output transition time according to phase**

Calculation of Transition Time Values (cont.)



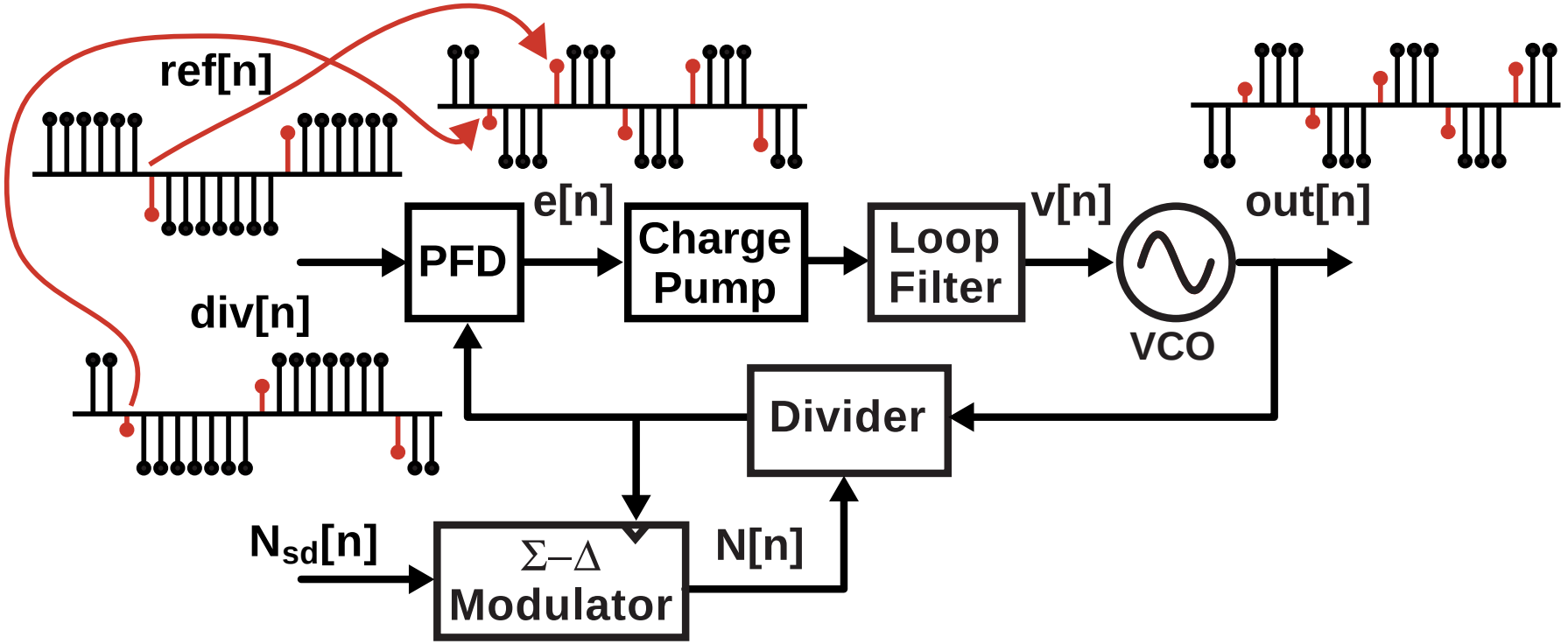
- **Use first order interpolation to determine transition value**

Processing of Edges using Double_Interp Protocol



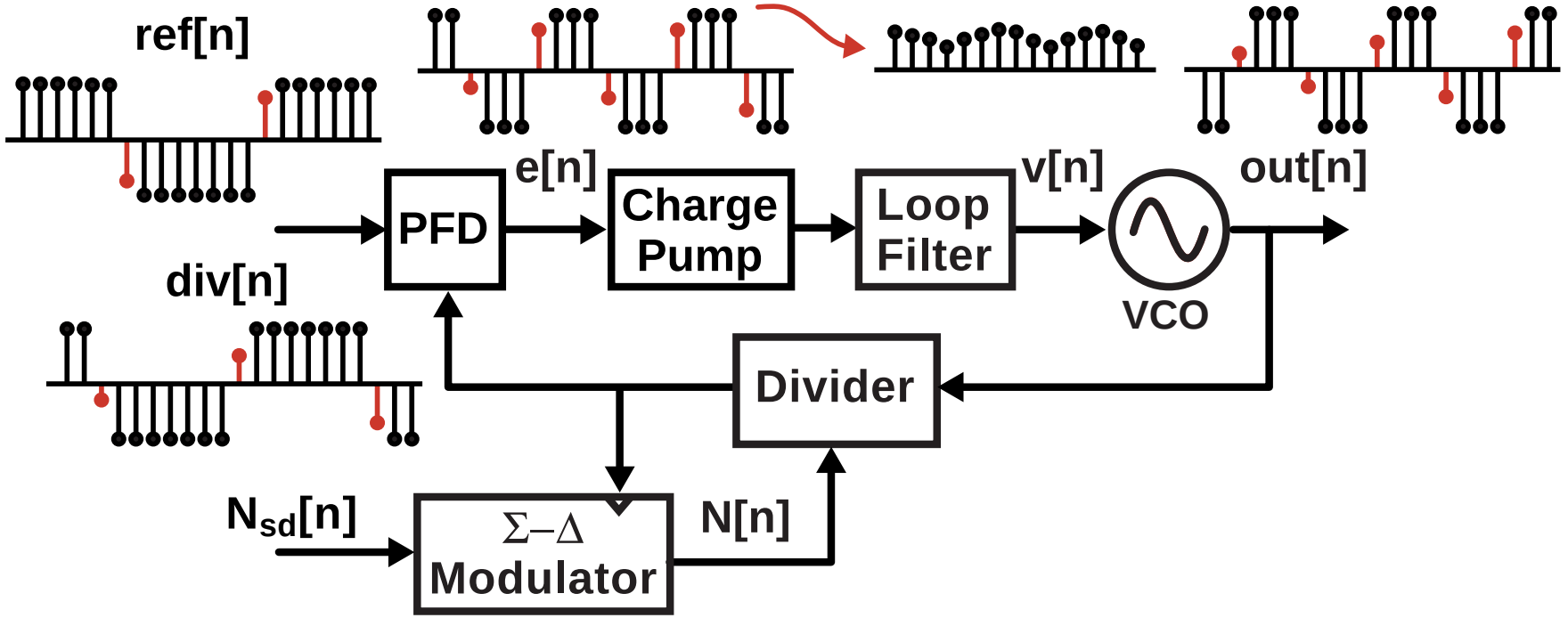
- Frequency divider block simply passes a sub-sampling of edges based on the VCO output and divide value

Processing of Edges using Double_Interp Protocol



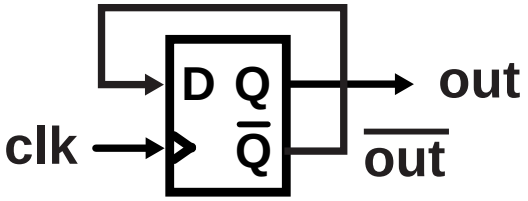
- Phase Detector compares edges times between reference and divided output and then outputs pulses that preserve the time differences

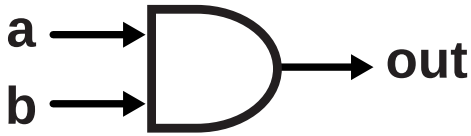
Processing of Edges using Double_Interp Protocol

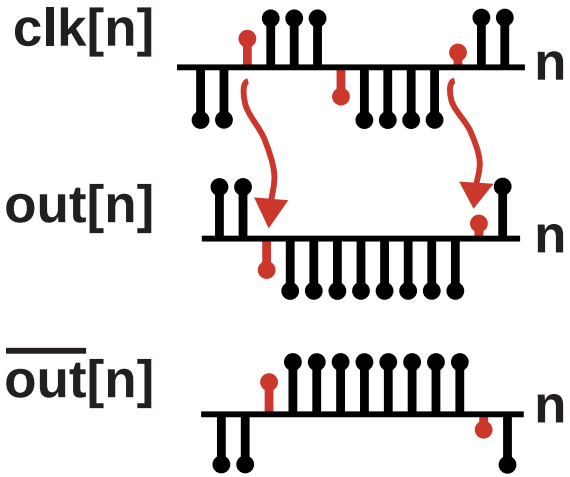


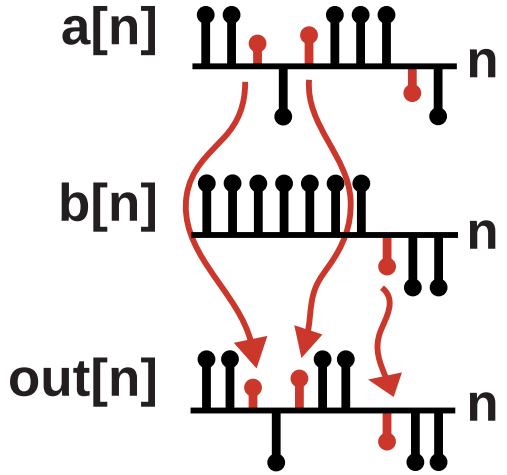
- Charge Pump and Loop filter operation is straightforward to model
  - Simply filter pulses from phase detector as discussed earlier

Using the Double_Interp Protocol with Digital Gates





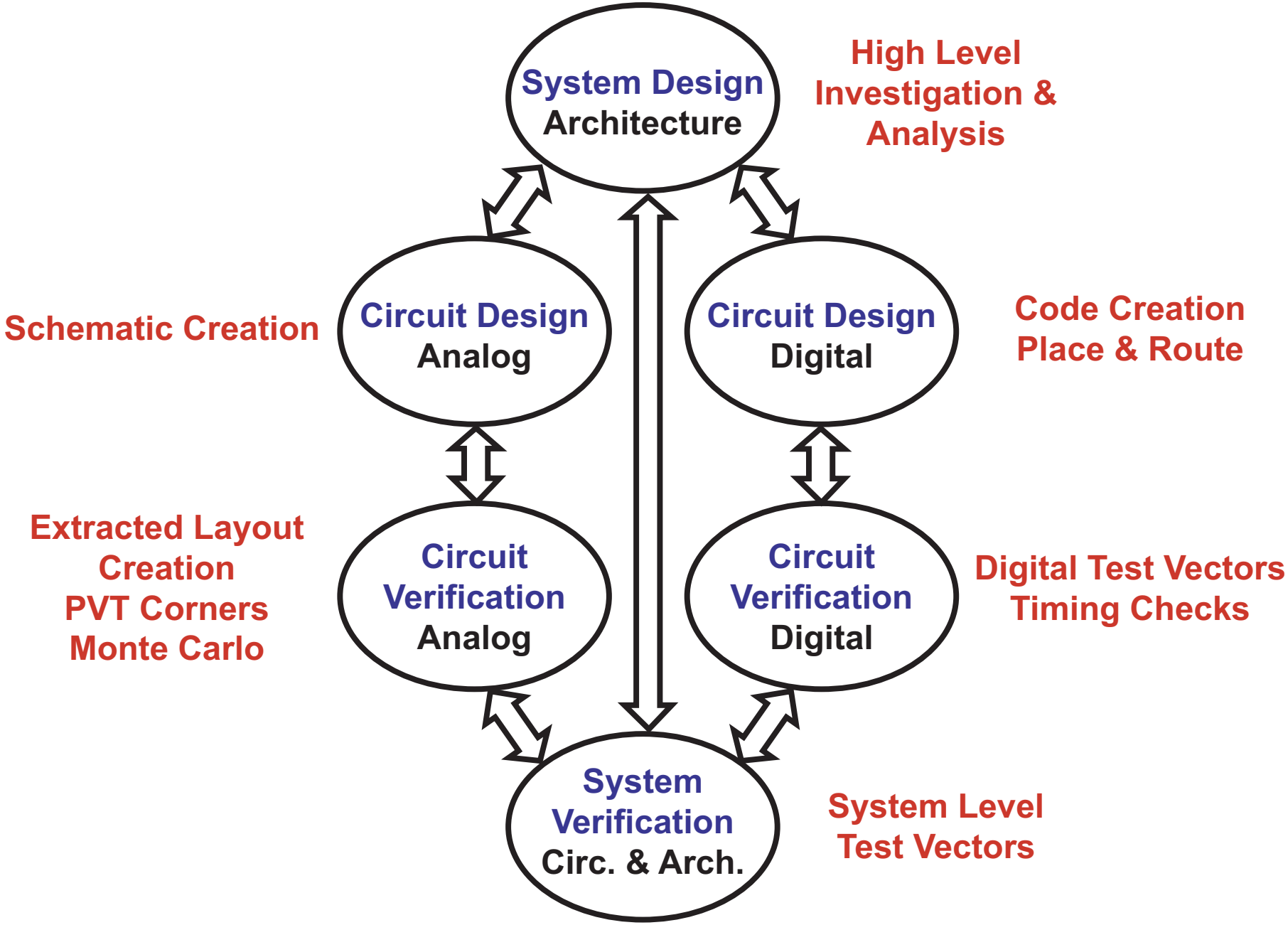




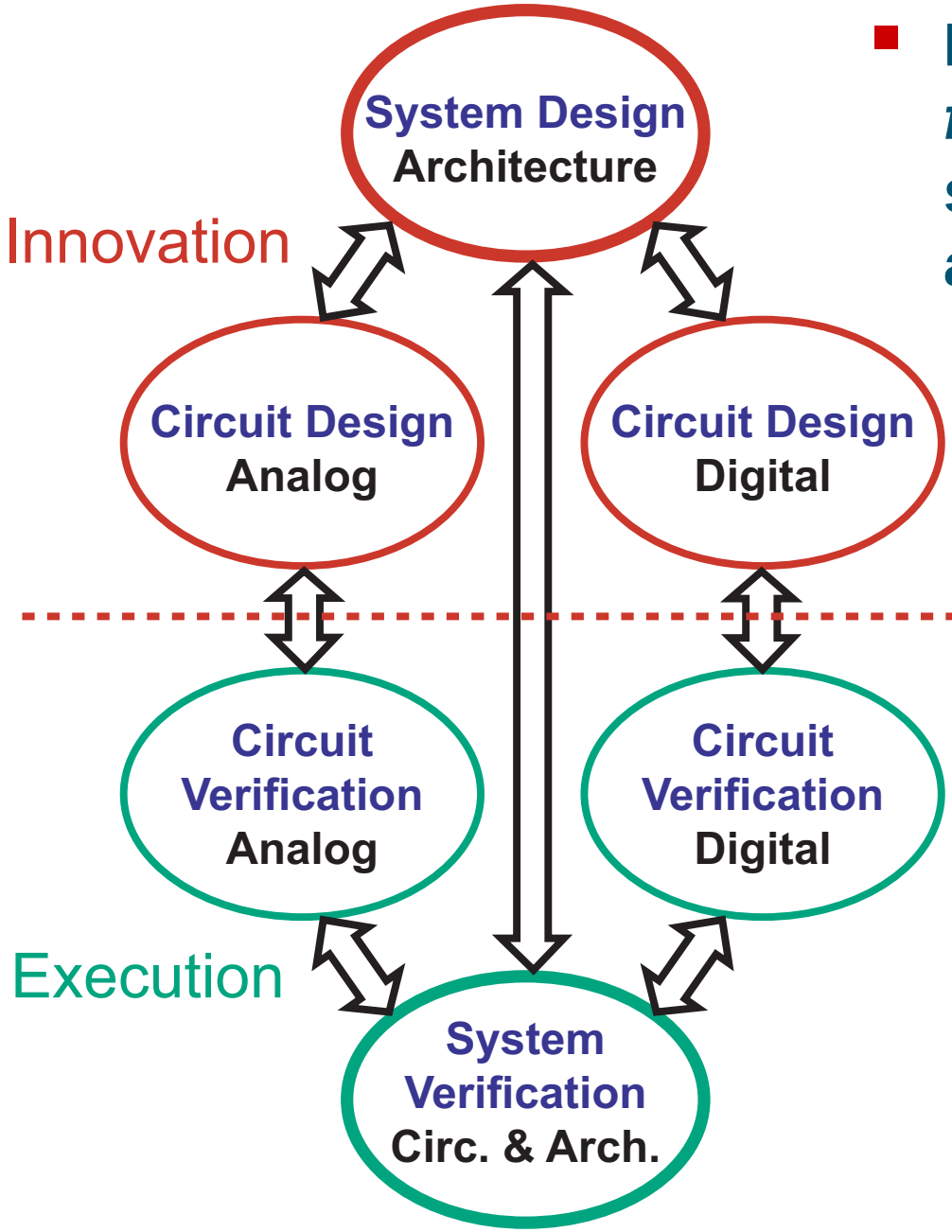
- **Relevant timing information contained in the input that causes the output to transition**
  - **Determine which input causes the transition, then pass its transition value to the output**

A Closer Look at System Level Simulation

Consider a Top Down, Mixed-Signal Design Flow

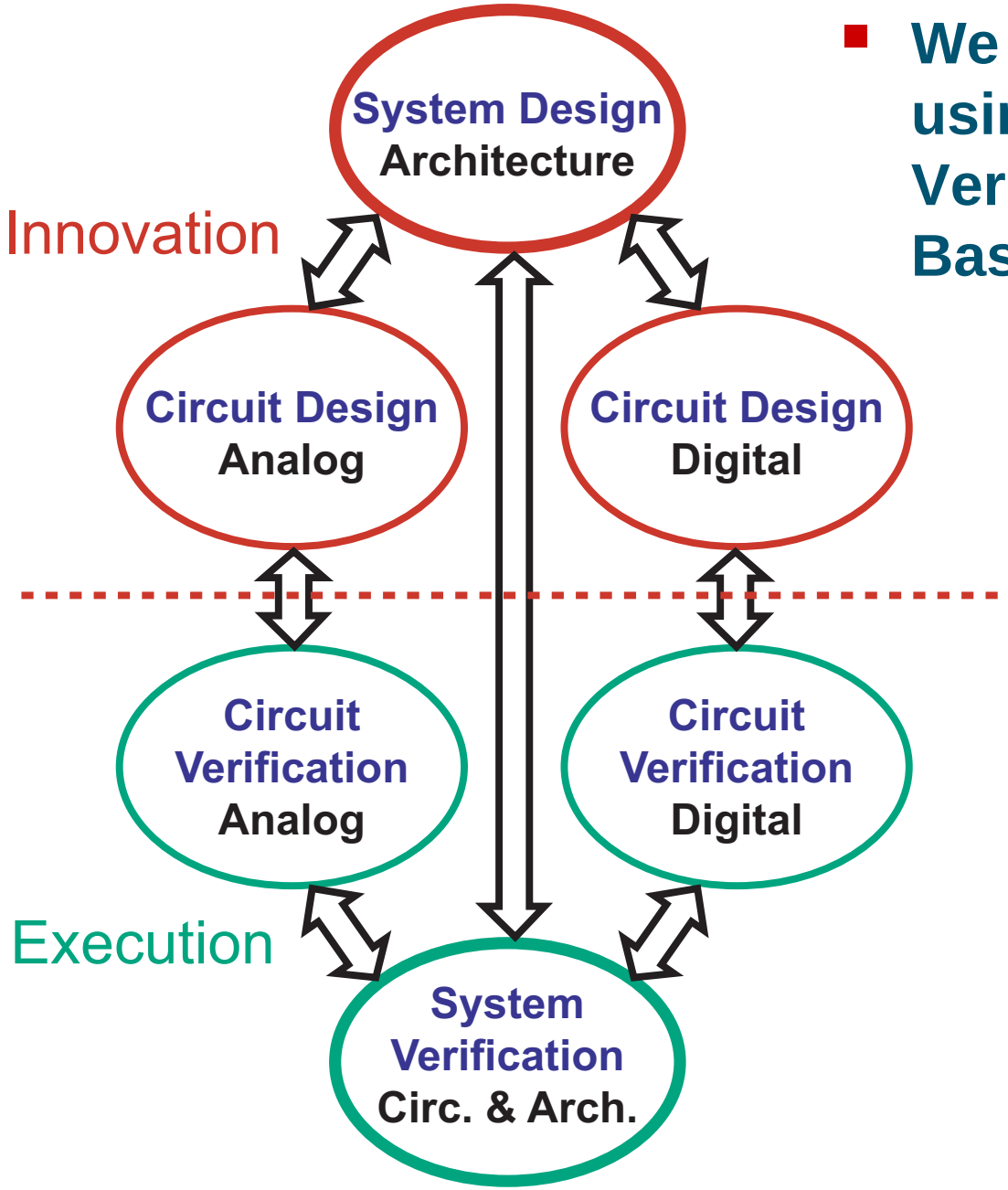


New Circuit Architectures Require Innovation



- Key to innovation is *fast* and *detailed* simulation of new architectures
  - Allows evaluation of *many* new ideas
  - Pinpoints key problem areas

C++ and Verilog Offer Fast System Level Simulation



- We will focus on using C++ and Verilog for Time-Based Circuis

Verilog Versus C++ as Behavioral Simulators

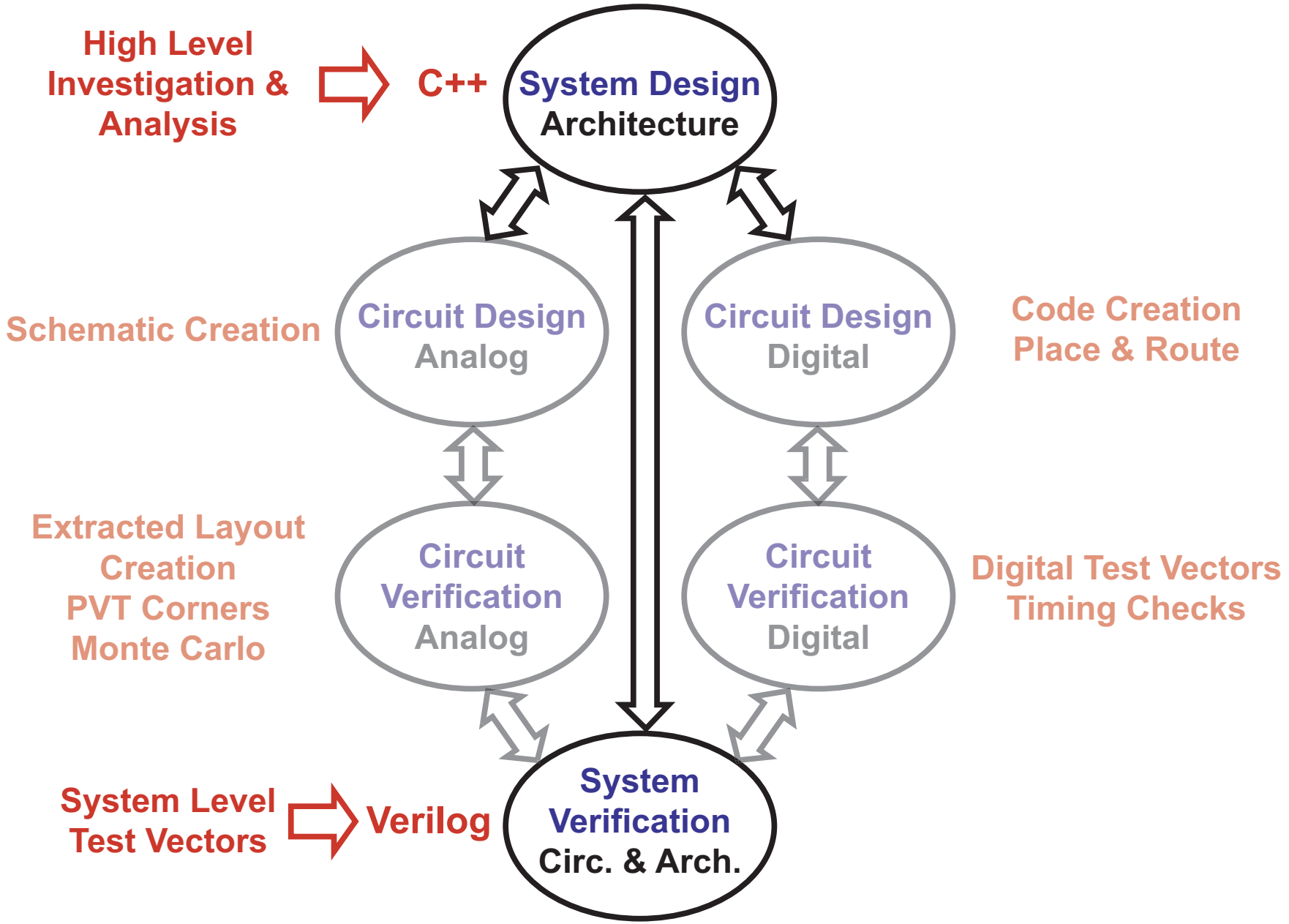
Verilog

- Popular in the US for digital verification
- Fairly limited as a language for analog modeling
  - Relatively time consuming to implement behavioral models
- Faster choice for systems that have sparse transition activity

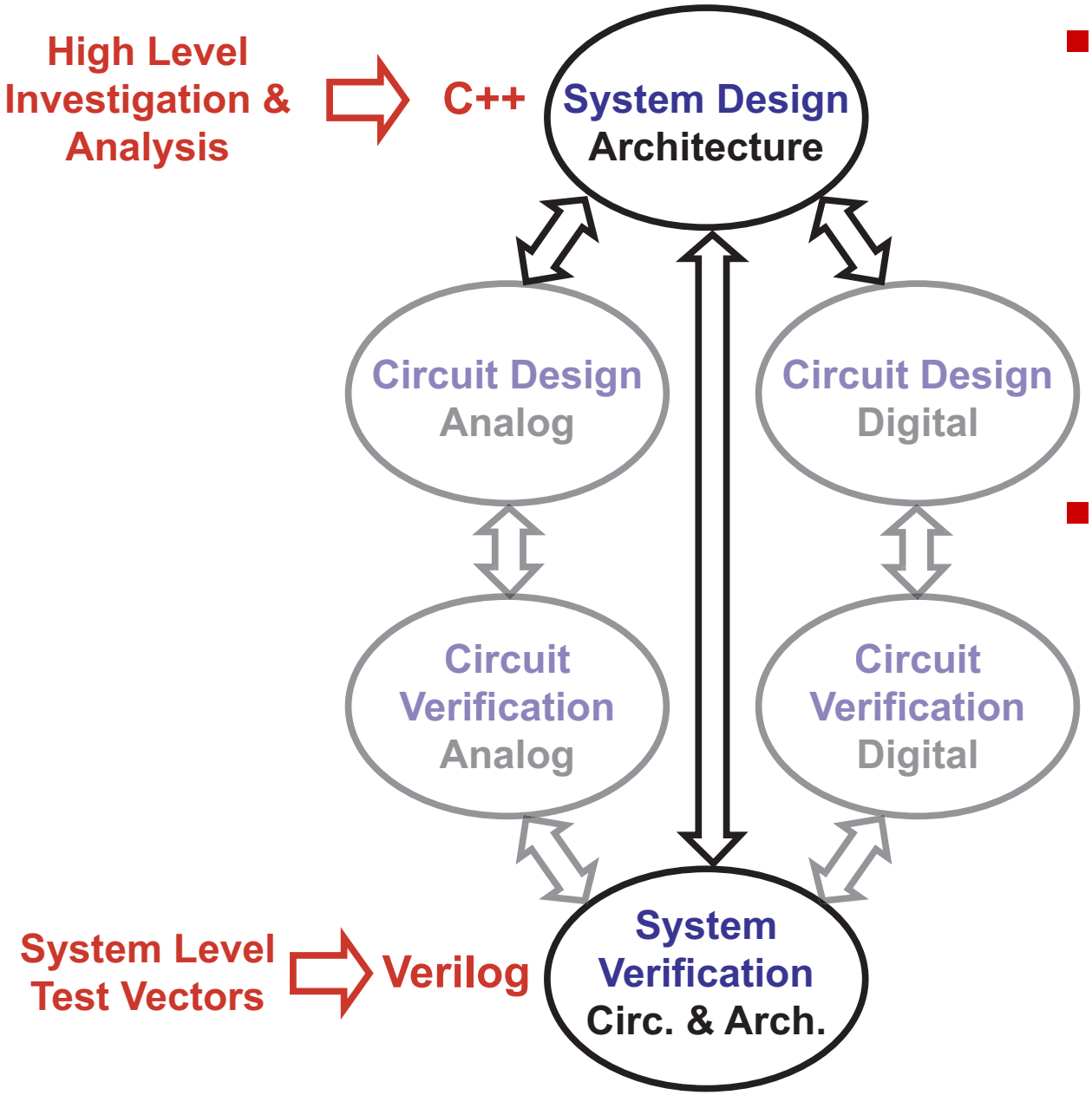
C++

- Increasing in popularity for digital simulation
  - SystemC, SystemVerilog
- Extremely powerful language for analog modeling
  - Class and function support allows simple path to sophisticated modeling
- Faster choice for systems that require continual update in their blocks

An Approach That Seems to Work Well



How Do We Make This Approach Efficient?



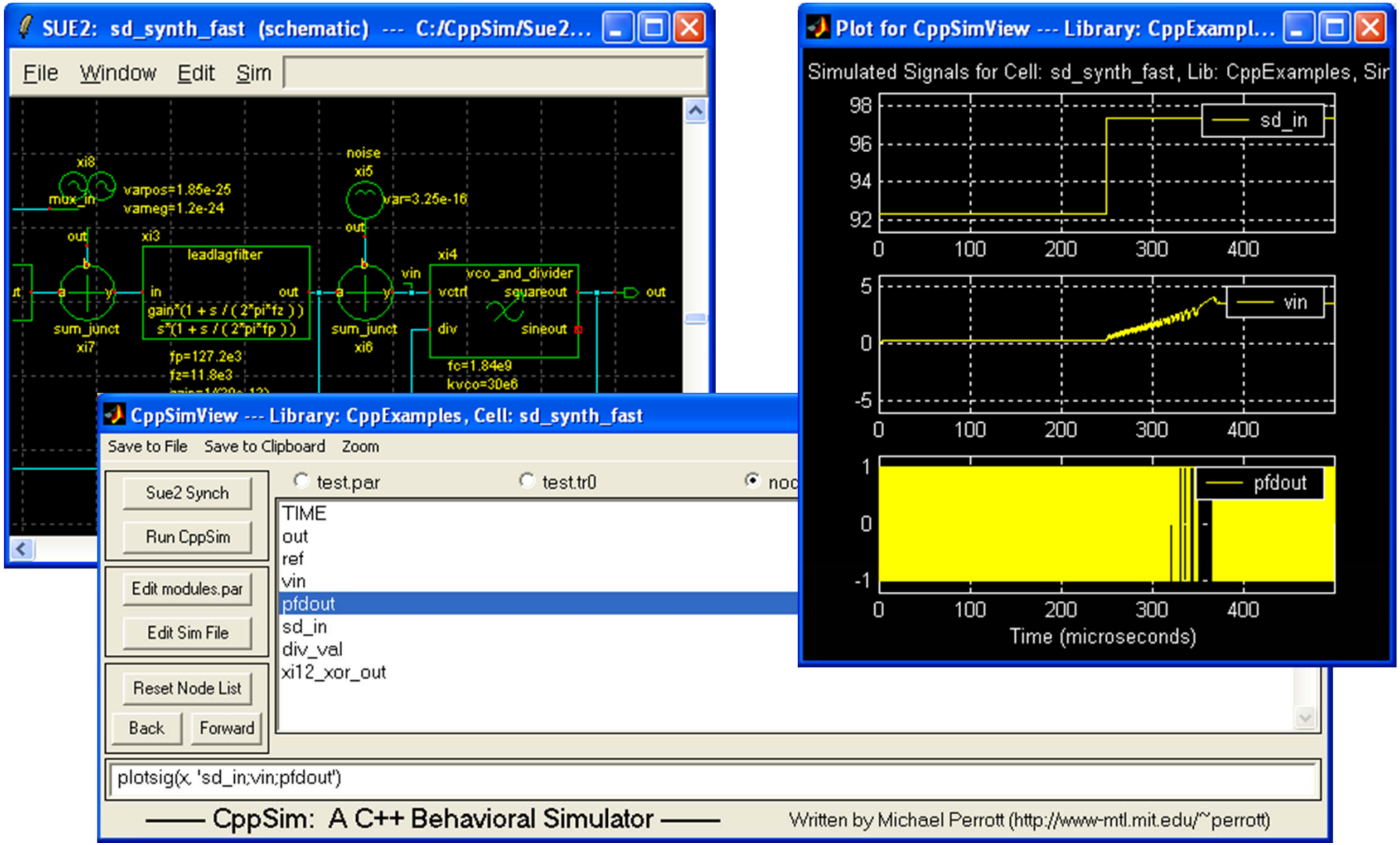
- Would like to incorporate Verilog models into C++
  - Provides accurate models for digital processing and control
- Would like to incorporate C++ models into Verilog
  - Allows re-use of critical block models
  - Provides C++ for complex test vector generation

CppSim and VppSim Offer C++/Verilog Co-Simulation

- **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**
  - **NCVerilog is the simulation engine**
    - C++ blocks accessed through the Verilog PLI
  - **Best option when system simulation focuses on digital verification with C++ stimulus**

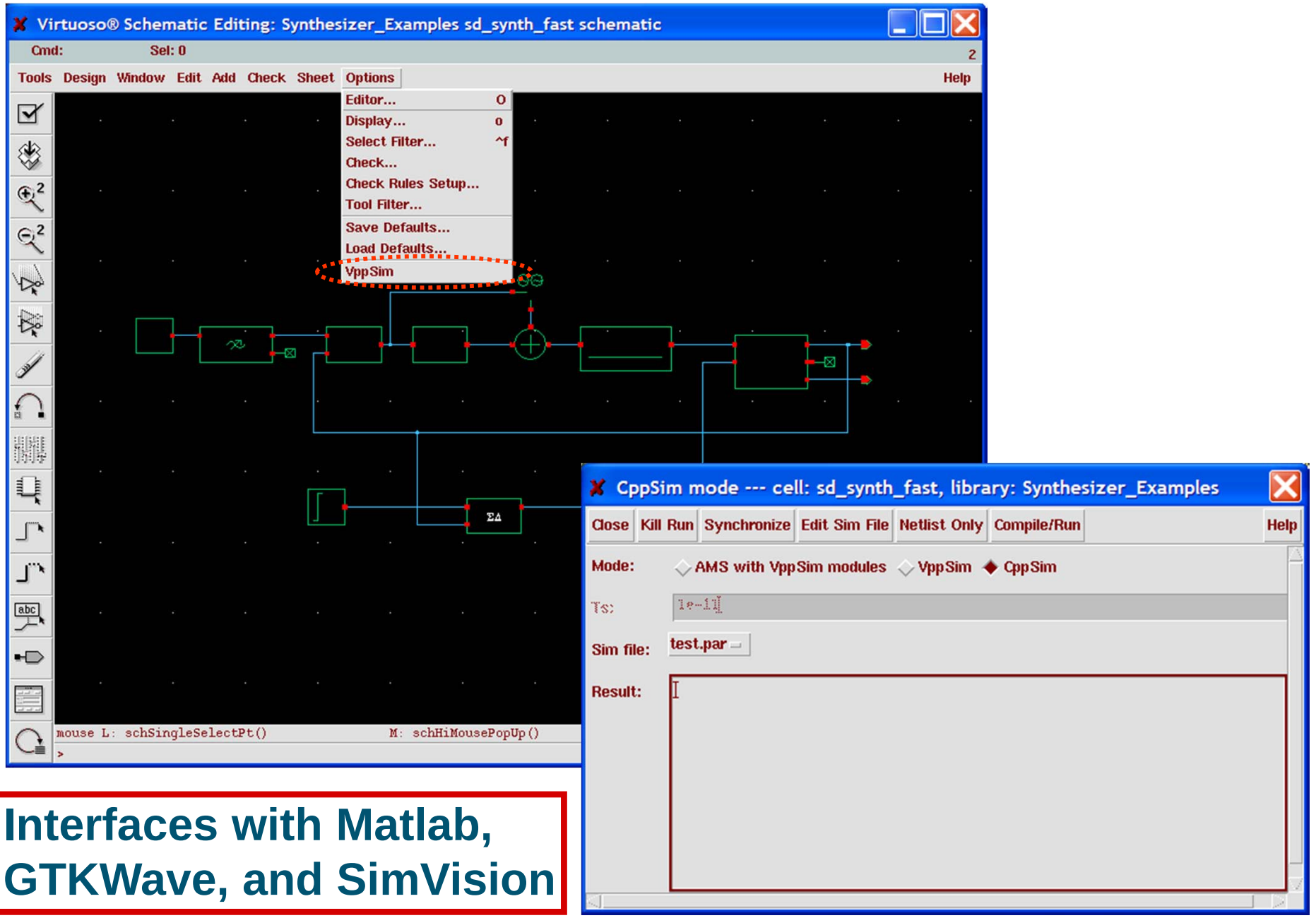
Each of these packages can be downloaded at
http://www.cppsim.com
and are free for both commercial and academic use
(VppSim requires an NCVerilog license)

Screenshot of CppSim (Windows Version)



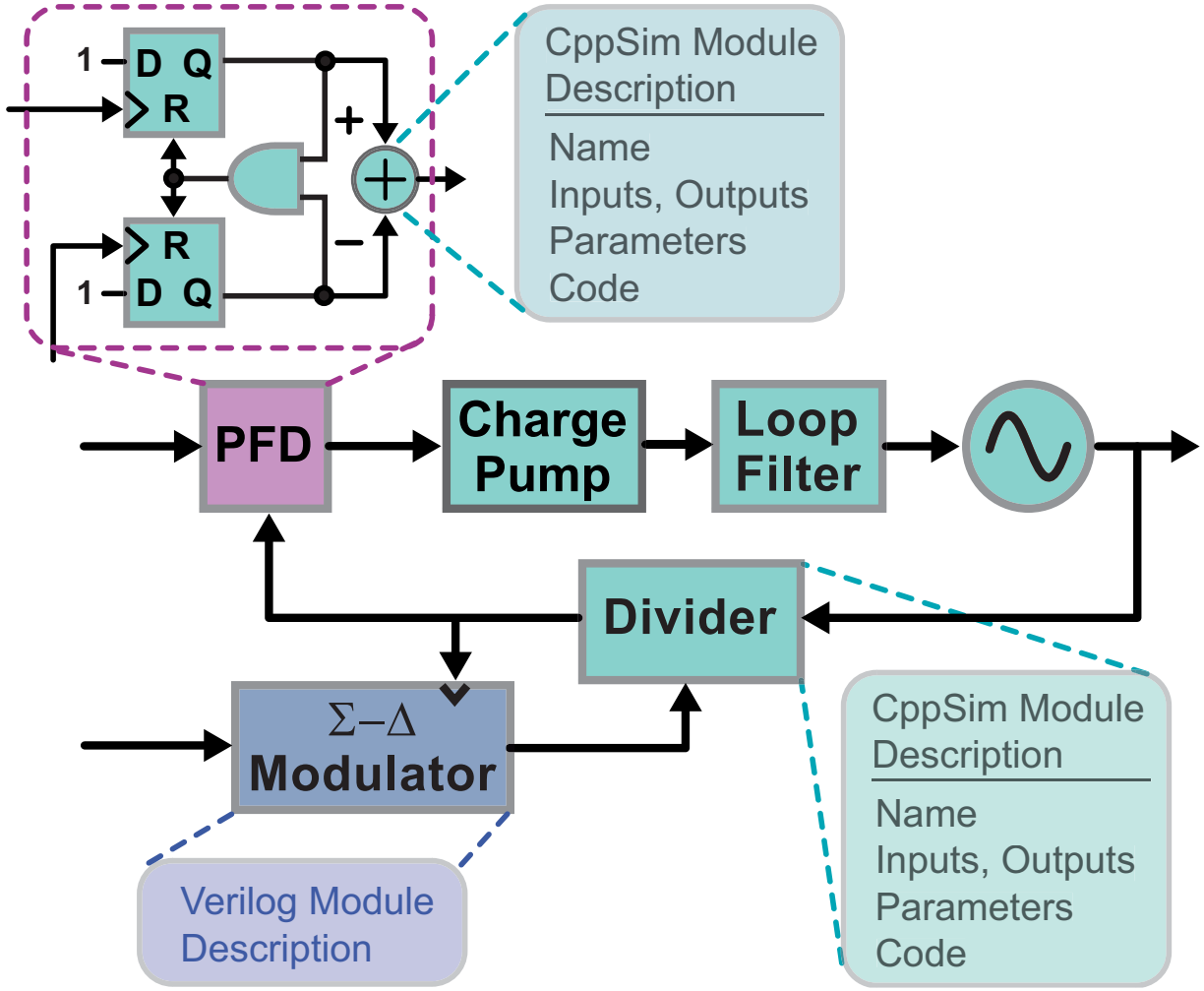
Cadence version is also free (part of VppSim package)

Screenshot of CppSim/VppSim (Cadence Version)



Interfaces with Matlab, GTKWave, and SimVision

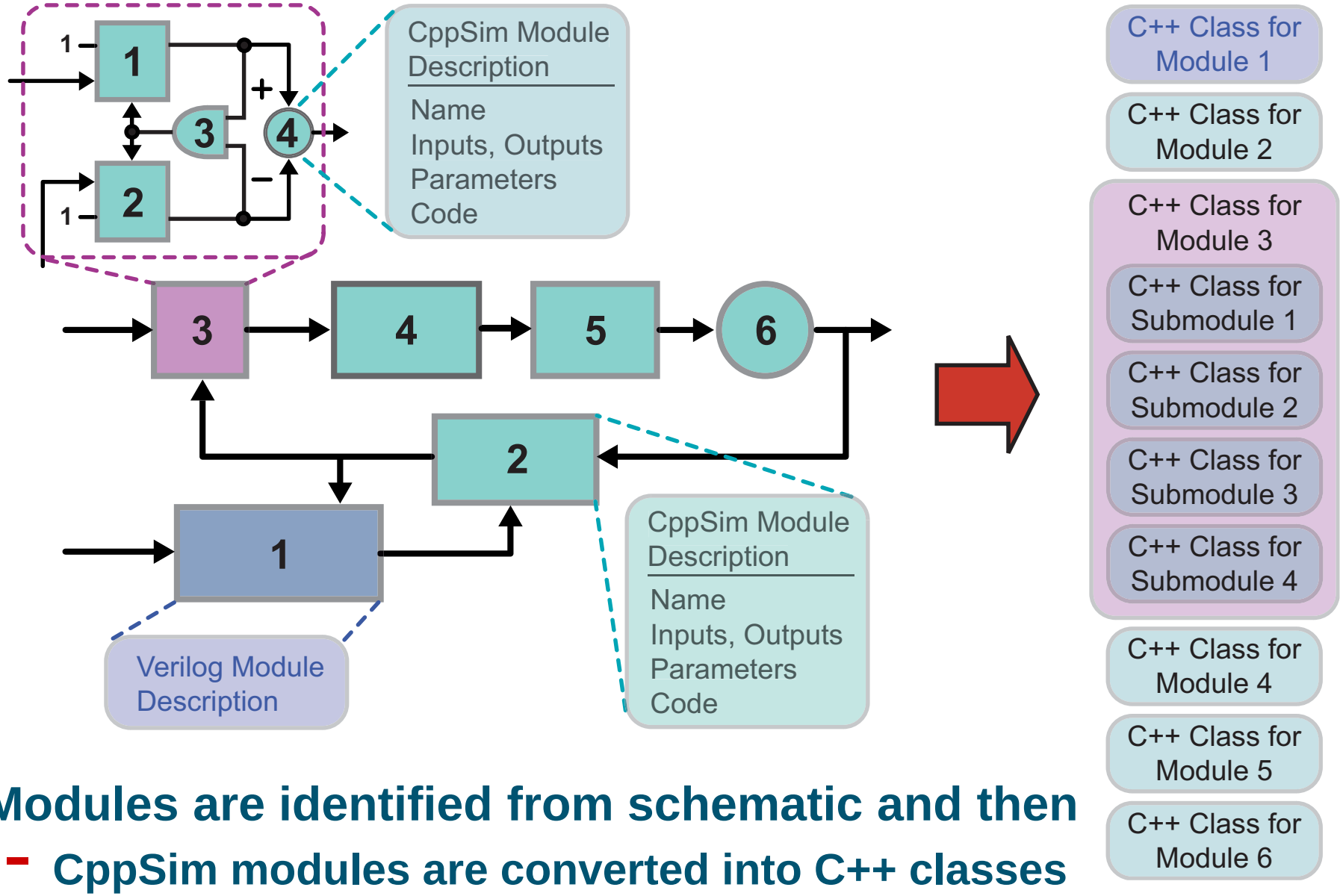
A Closer Look at CppSim/VppSim Methodology



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

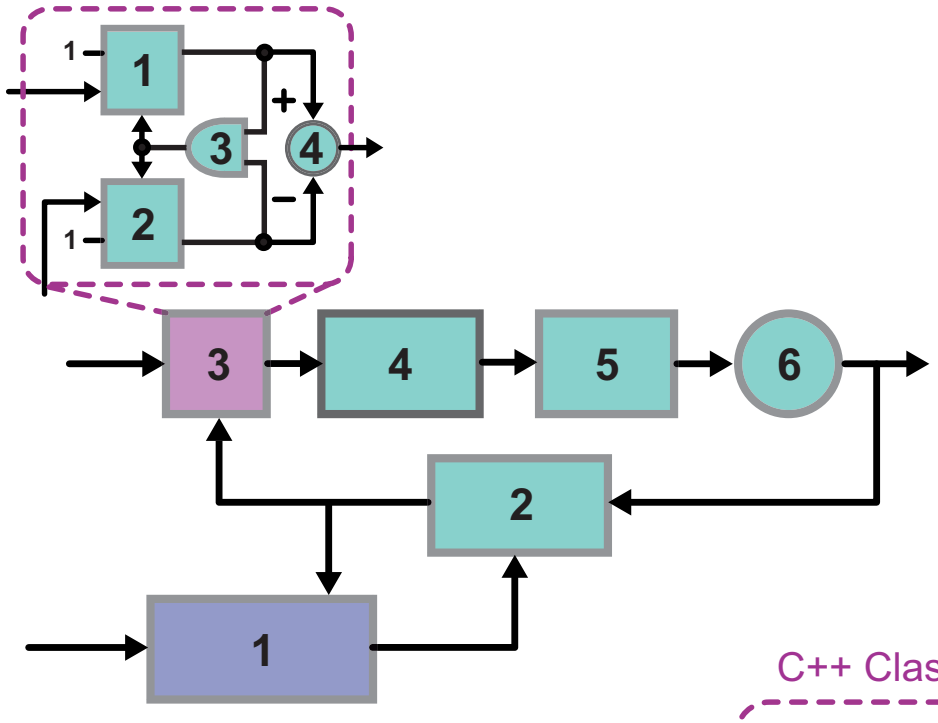
- **Designers graphically develop system based on a library of C++/Verilog symbols and code**
  - **Easy to create new symbols with accompanying code**

CppSim Automates C++ Class Generation

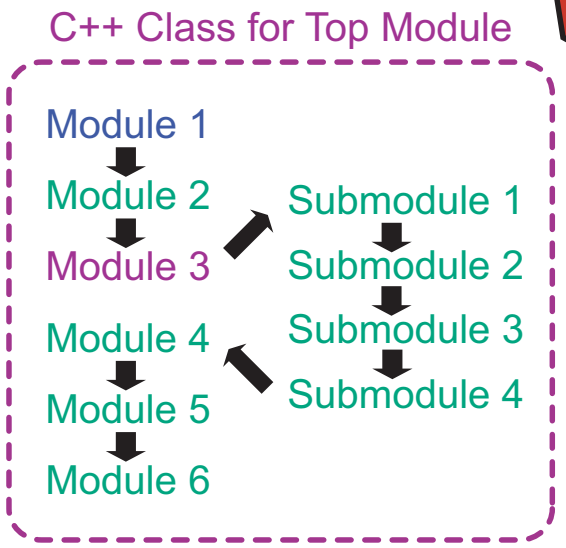


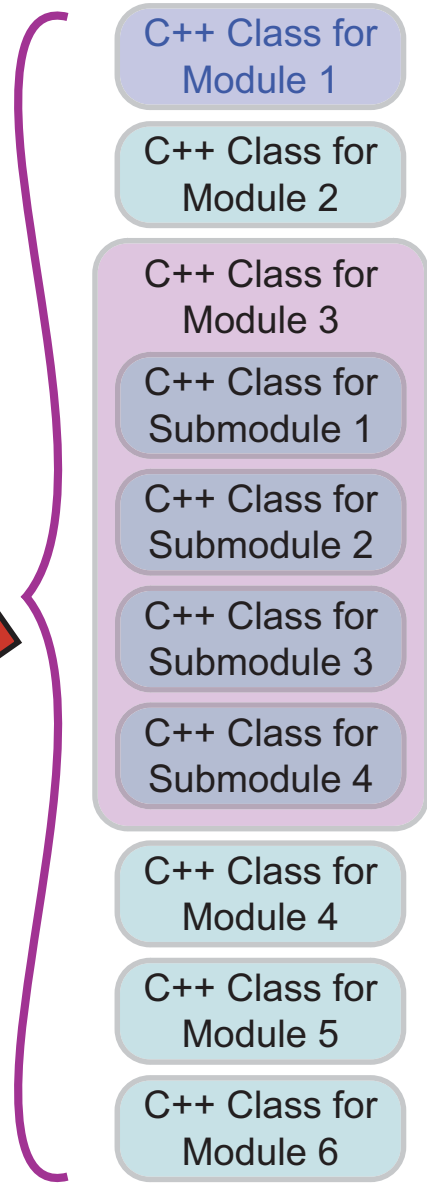
- **Modules are identified from schematic and then**
  - **CppSim modules are converted into C++ classes**
  - **Verilog modules are translated into C++ classes using Verilator**

CppSim Assembles C++ Classes into Overall Sim Code

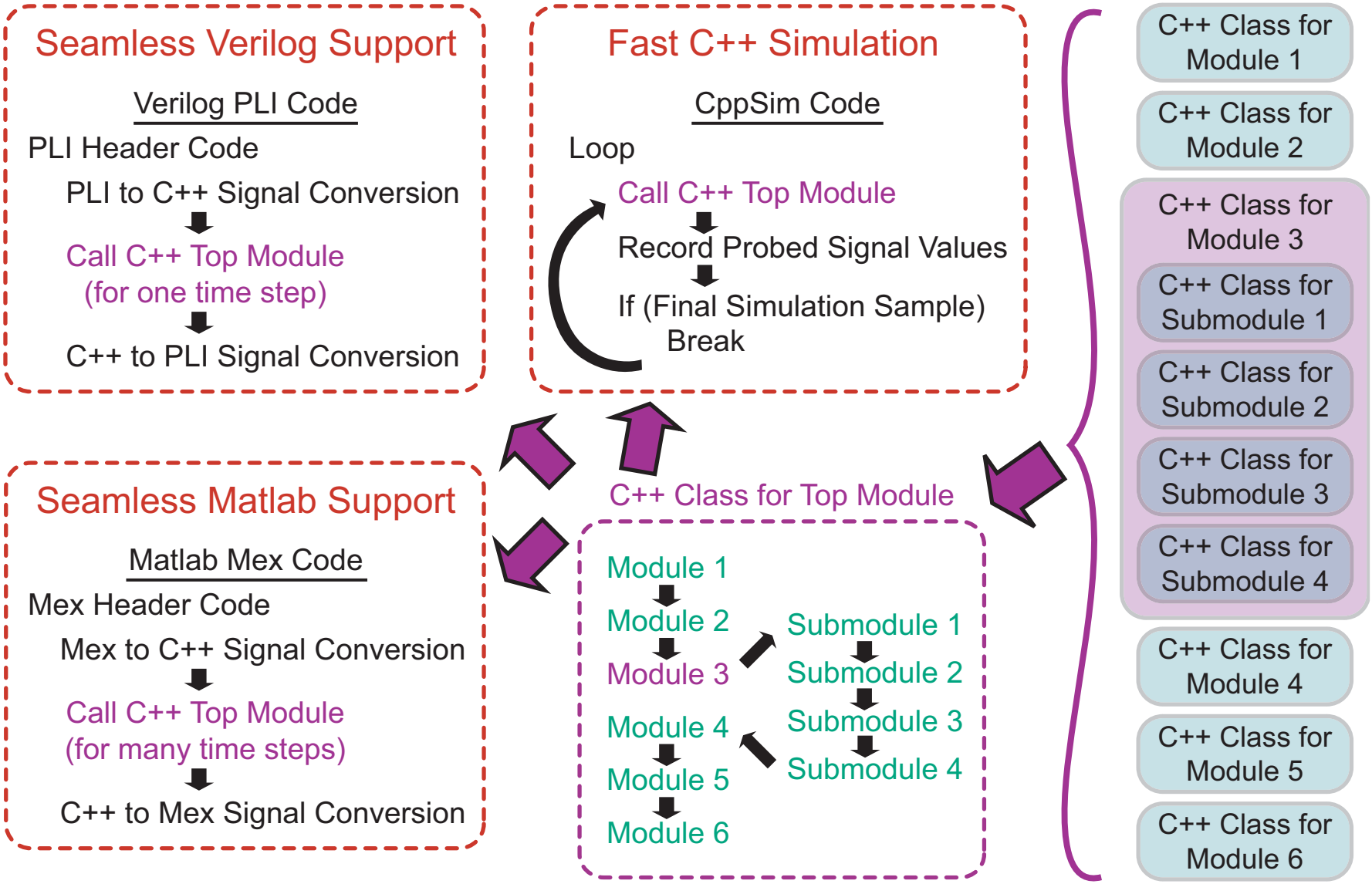


- **Block-by-block execution of each module at each time step**
- **Hierarchical description is retained**





C++ Code Is Easily Embedded In Other Simulators

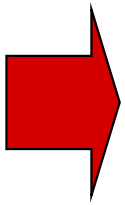


- **CppSim provides automatic Matlab mex file generation**
- **VppSim embeds C++ into NCVerilog simulator**

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
```

VppSim Example: Using Busses in CppSim Module

CppSim module

```
module: queue2
parameters: int bit_width
inputs:     double_interp clk,
            double rst_n,
            bool in[2047:0],
            int enqueue,
            bool dequeue[31:0]
outputs:    bool out[2047:0],
            bool not_empty[31:0],
            int not_full
```

Resulting Verilog module

```
//////////// Auto-generated from CppSim module ////////////
module queue2(clk, rst_n, in, enqueue,
                    dequeue, out, not_empty,
                    not_full);

  parameter bit_width = 0;
  input clk;
  input rst_n;
  input [2047:0] in;
  input [31:0] enqueue;
  input [31:0] dequeue;
  output [2047:0] out;
  output [31:0] not_empty;
  output [31:0] not_full;

  wreal clk;
  real clk_rv;
  wreal rst_n;
  real rst_n_rv;
```

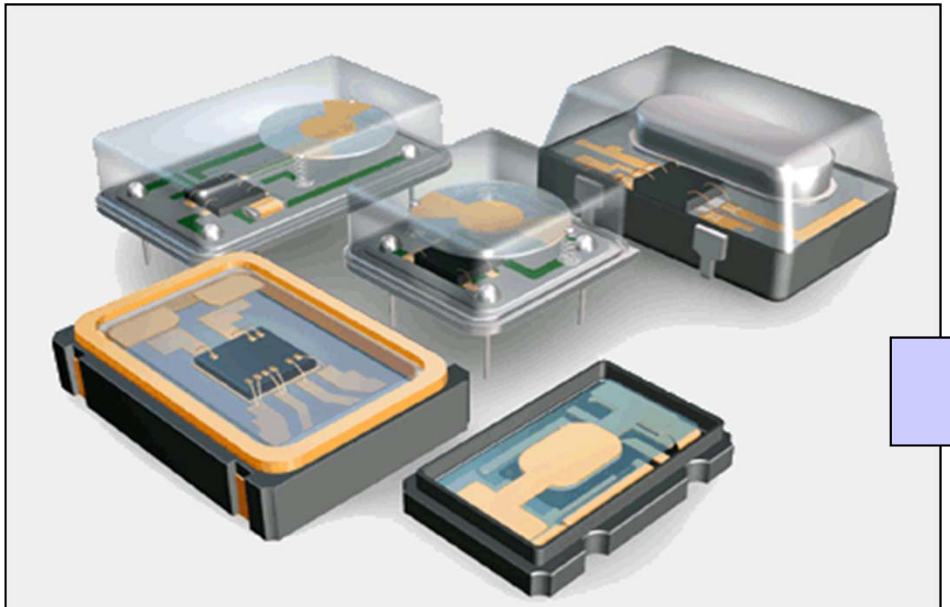
Many Tutorials Available for CppSim/VppSim

- **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

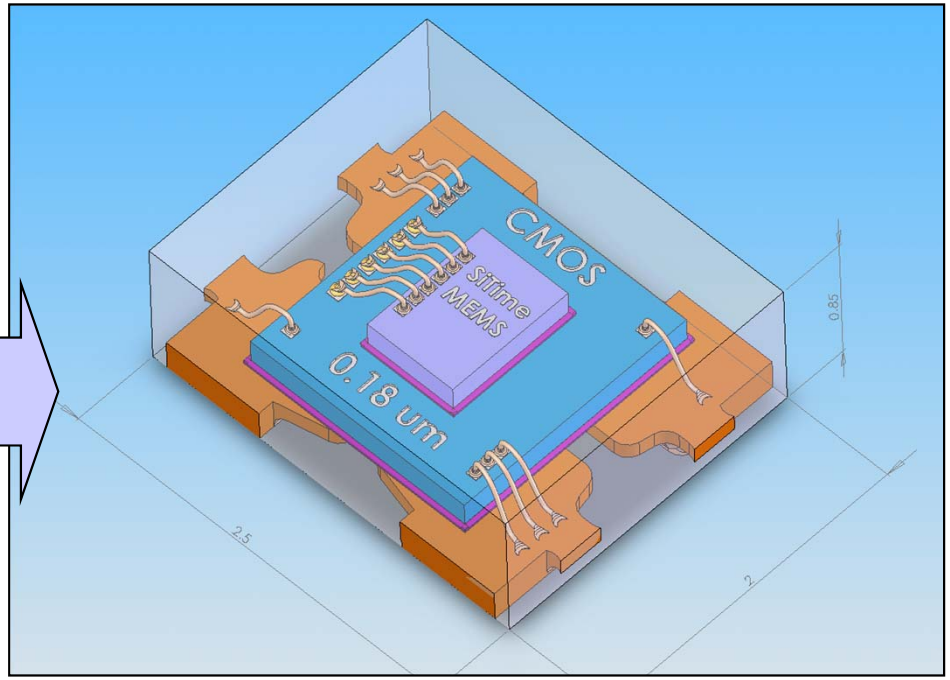
Case Study: MEMS-based Programmable Oscillators

Quartz Oscillators



source: www.ecliptek.com

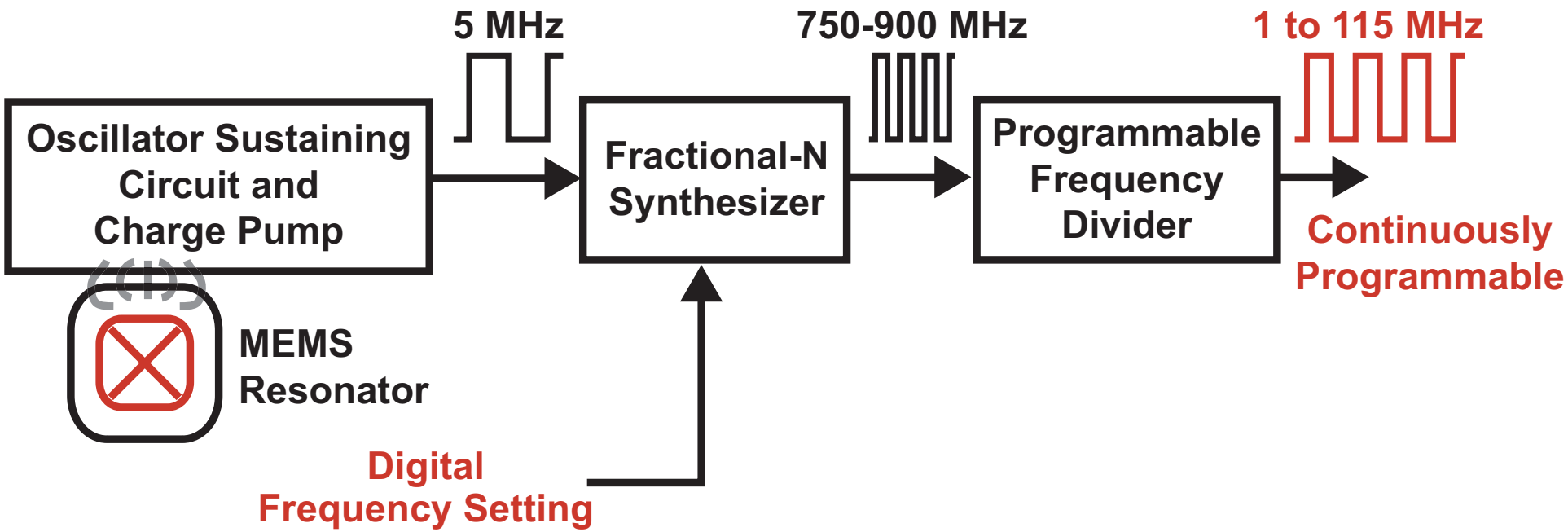
MEMS-based Oscillator



- **A part for each frequency and non-plastic packaging**
  - **Non-typical frequencies require long lead times**
- **Same part for all frequencies and plastic packaging**
  - **Pick any frequency you want without extra lead time**

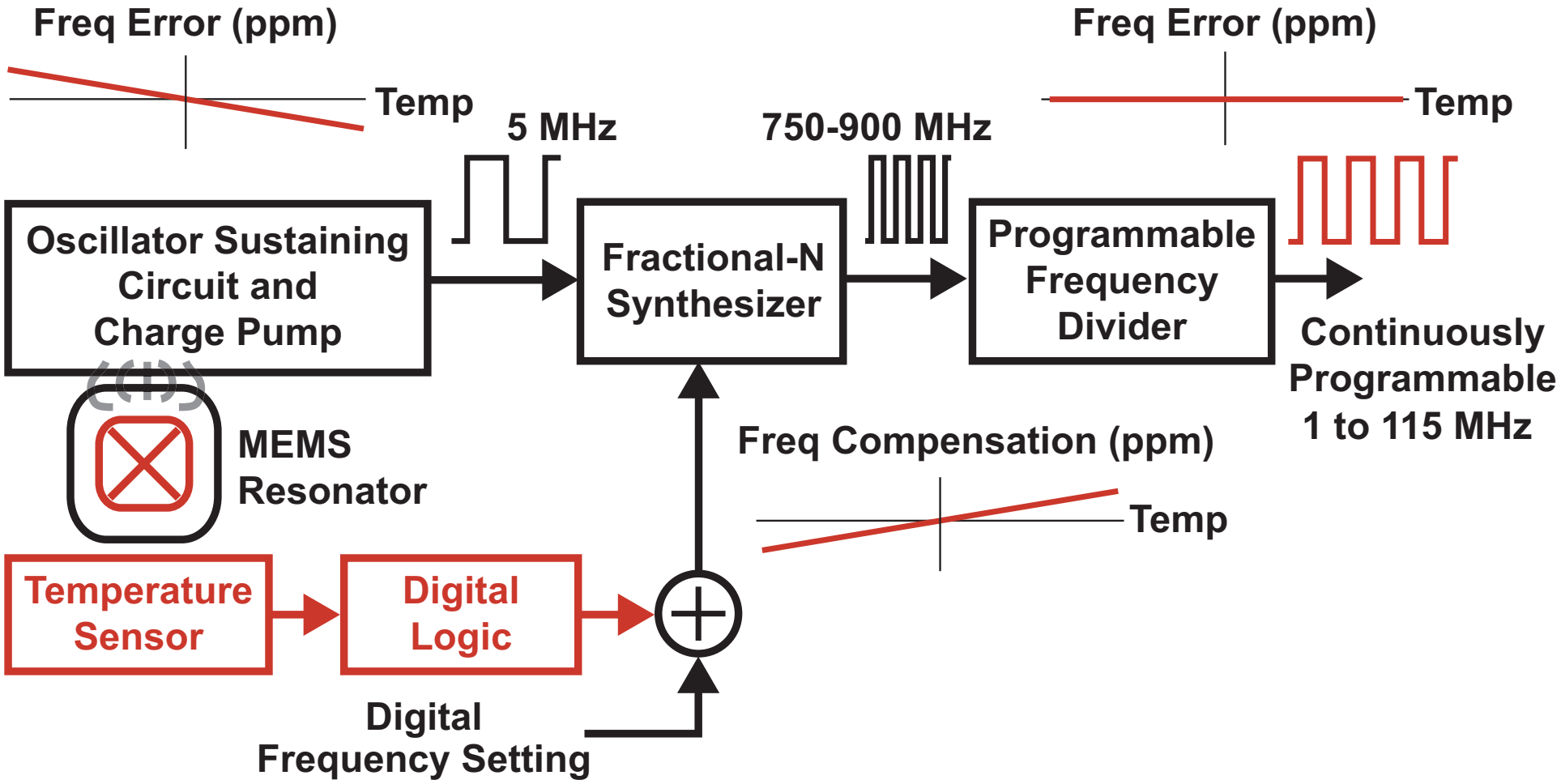
Key benefit: high volumes at low cost using IC fabrication

Architecture of MEMS-Based Programmable Oscillator



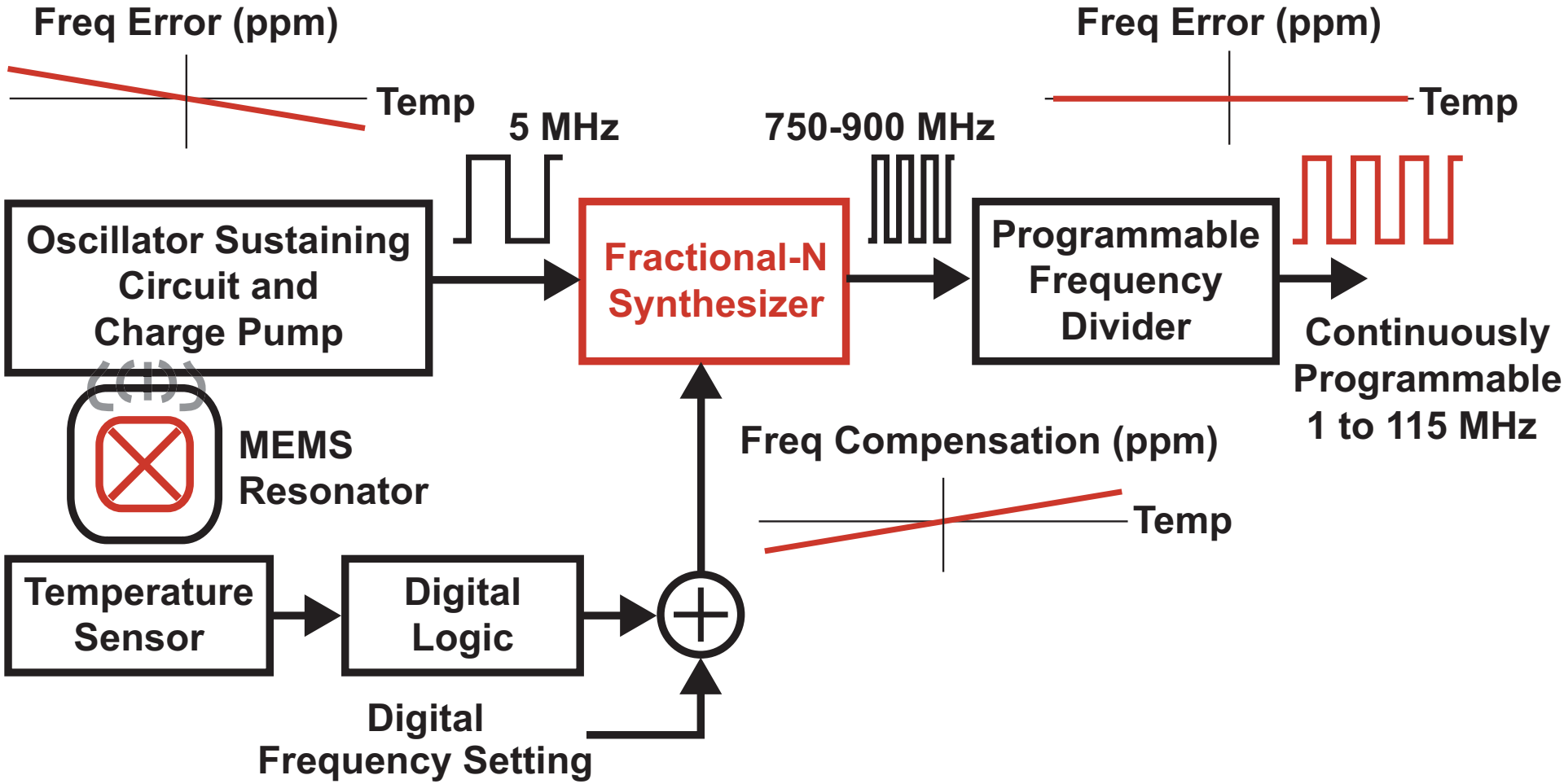
- MEMS device provides high Q resonance at 5 MHz
  - CMOS circuits provide DC bias and sustaining amplifier
- Fractional-N synthesizer multiplies 5 MHz MEMS reference to a programmable range of 750 to 900 MHz
- Programmable frequency divider enables 1 to 115 MHz output

Compensation of Temperature Variation



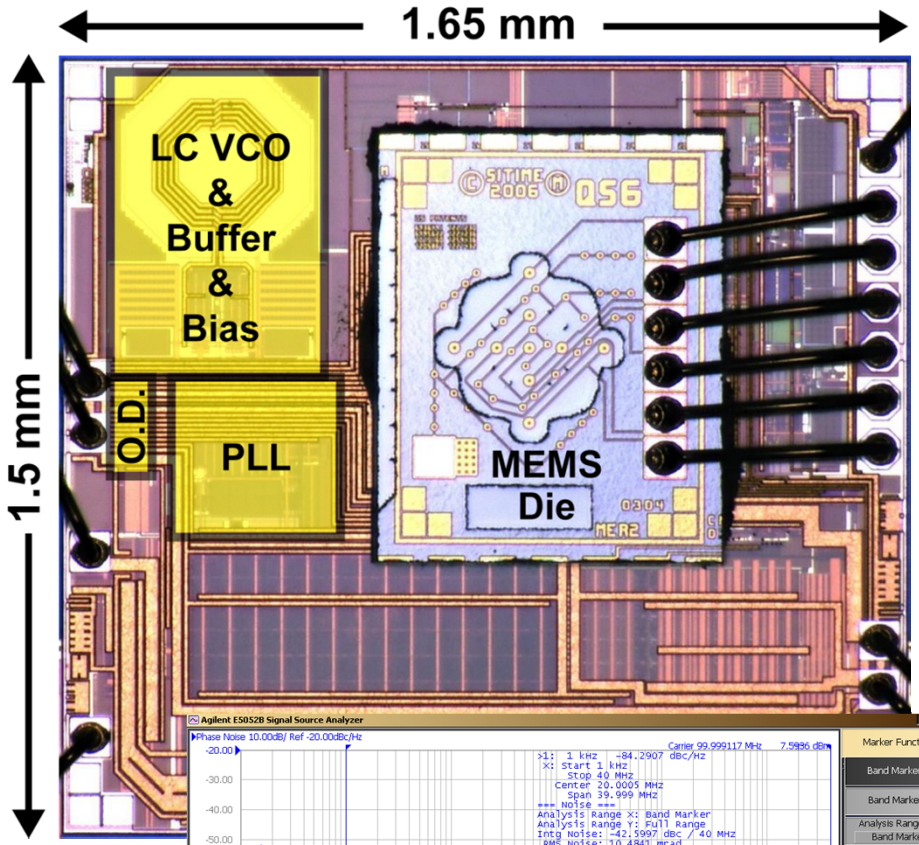
- High resolution control of fractional-N synthesizer allows simple method of compensating for MEMS frequency variation with temperature
  - Simply add temperature sensor and digital compensation logic

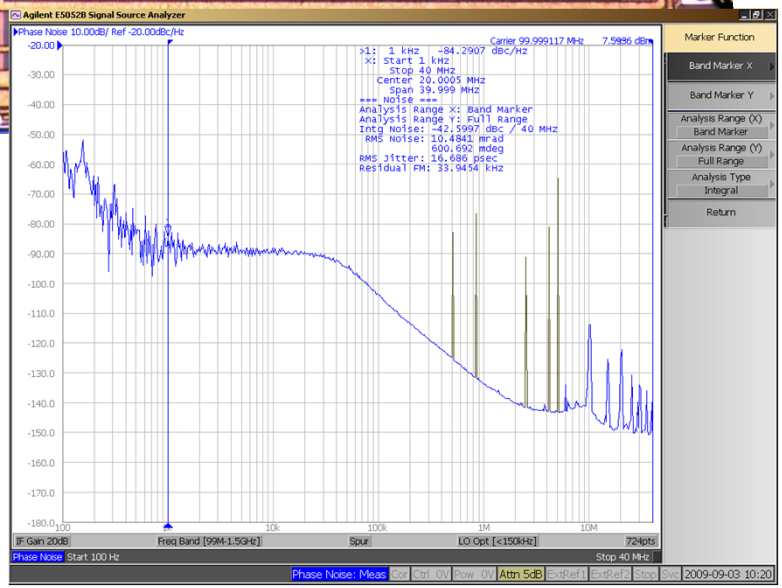
Achieving Fast and Accurate System Level Simulation

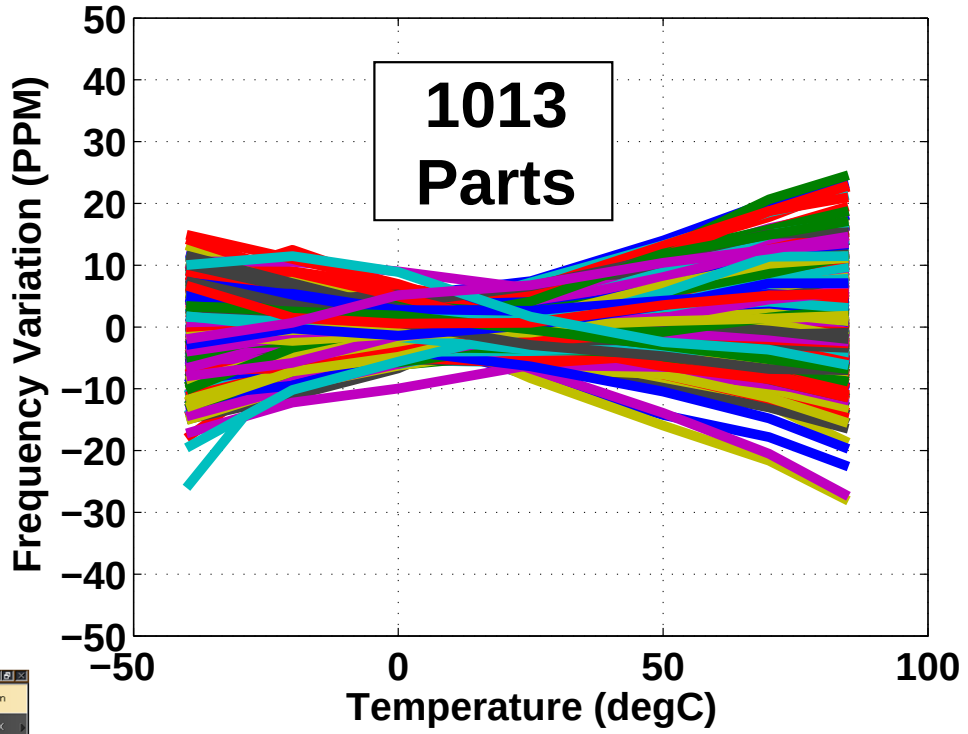


- System level simulation involves several types of circuits
  - Fractional-N synthesizer and MEMS oscilllator are time-based
  - Temperature sensor is traditional analog
  - Many digital blocks interact with the above

Measured Results Confirm CppSim/VppSim Flow







- Measured phase noise closely matches simulations
- Measured frequency stability is similar to many quartz parts

Benchmark of Simulators on Entire IC

Tabulated simulation times for our next generation MEMS oscillator:

- **Detailed architectural model using CppSim**
  - **Allows examination of noise and analog dynamics**
  - **Execution time: 2.8 milliseconds/hour**
- **Detailed verification model using VppSim**
  - **Allows validation of digital functionality in the context of analog control and hybrid digital/analog systems**
  - **Execution time: 7 milliseconds/minute**
- **Spice-level model**
  - **Allows checking of floating gate, over-voltage conditions, startup of bandgap and regulators, etc.**
  - **Execution time:**
    - **Spectre Turbo: 2 microseconds/day**
    - **BDA: 8 microseconds/day**

Conclusions

- **Time-based circuits are becoming more mainstream due their advantages in advanced CMOS processes**
  - **Digital phase-locked loops**
  - **VCO-based Analog-to-Digital Conversion**
  - **RF transmitters leveraging pulse width modulation**
- **Fast and accurate system level simulation of such circuits can be achieved with the “double_interp” protocol**
- **CppSim and VppSim provide a simple and free approach to achieving C++/Verilog Co-Simulation**
  - **CppSim is useful for primarily analog focus**
  - **VppSim is useful for primarily digital focus**