An Efficient Approach to System Level, Mixed-Signal Simulation Using CppSim and VppSim

Michael H. Perrott
December 2012

Copyright © 2012 by Michael H. Perrott
All rights reserved.
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
Consider a Top Down, Mixed-Signal Design Flow

System Design Architecture

Circuit Design
- Analog
- Digital

Circuit Verification
- Analog
- Digital

Digital Test Vectors
Timing Checks

Code Creation
Place & Route

High Level Investigation & Analysis

Schematic Creation

Extracted Layout Creation
PVT Corners
Monte Carlo

System Verification
- Circ. & Arch.
- Test Vectors
Good Execution Is Certainly A Key to Success

- Execution often becomes key focus
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
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 $f_o \cdot T_s \cdot 360^\circ$

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
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

- 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

(Johns and Martin, Analog Integrated Circuit Design)
**Alternative: Event Driven Simulation**

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

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

\[ e(t) \]

\[ v(t) \]

\[ h(t) \]

\[ h[n] = T_s \cdot h(T_s n) \]

\[ e[n] \]

\[ v[n] \]
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
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.

(Assume VCO output is a square-wave for this discussion)
Calculation of Transition Time Values

- Model VCO based on its phase
Determine output transition time according to phase
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.
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

\[ \text{osc_out}[k] \rightarrow \text{Conversion Module} \rightarrow \text{osc_buf}[k] \]
Summary of Block-by-Block Computation Method

- Requires unilateral flow through blocks
- Impacts phase margin of feedback loops
  - Need $1/T_s \gg$ 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

What is the best programming language for this approach?
## Verilog Versus C++ for Block-by-Block Simulation

### Verilog
- Excellent language for digital modeling and verification
- Time consuming to implement analog modeling
- Best choice in cases where blocks have sparse transition activity

### C++
- Excellent language for analog modeling
  - Object oriented
  - Signal processing
- Time consuming to implement digital modeling
  - SystemC?
- Best choice in cases where blocks require continual update every time step
An Approach That Seems to Work Well

High Level Investigation & Analysis  \(\rightarrow\) C++

Schematic Creation

Extracted Layout Creation
PVT Corners
Monte Carlo

System Level Test Vectors  \(\rightarrow\) Verilog

System Design Architecture

Circuit Design Analog

Circuit Design Digital

Circuit Verification Analog

Circuit Verification Digital

System Verification Circ. & Arch.

Code Creation Place & Route

Digital Test Vectors Timing Checks
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

Diagram:

- System Design Architecture
  - Circuit Design Analog
  - Circuit Design Digital
  - Circuit Verification Analog
  - Circuit Verification Digital
  - System Verification Circ. & Arch.

Arrows indicate the flow:
- High Level Investigation & Analysis → C++
- System Level Test Vectors → Verilog
**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**
  - 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
Free Download at www.cppsim.com

CppSim System Simulator

Discover a faster and easier way to perform system level simulation of complex mixed-signal circuits.

CppSim automatically generates, compiles, and runs C++ code corresponding to the schematic design that you create.

**Graphical Interface:**
Systems are specified and simulated within a schematic editor, Sue2, and results are viewed using a waveform viewer (CppSimView or GTKWave).

**Analog modules:**
A simple text template for each module is filled in by the user which can make use of a rich set of C++ classes to represent common functions such as filtering, noise, etc.

**Digital modules:**
CppSim utilizes Verilator to automatically create C++ code corresponding to your Verilog modules, and seamlessly integrates this code into your system simulation.

"Designing precision timing circuits that exhibit grossly non-linear behavior requires performing accurate simulations in the time domain. Such simulations take prohibitively long, even in commercial behavioral simulators, which have often limited our ability to evaluate new PLL, CDR, and ADC architectures in the past. CppSim and its Cadence compatibility have filled this void and have been instrumental in the design of..."
Screenshot of CppSim/VppSim (Windows Version)

CppSim: A C++ Behavioral Simulator

Readily Interfaces with Matlab and GTKWave

Written by Michael Perrott (http://www-mtl.mit.edu/~perrott)
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 Module Description**
- Name
- Inputs, Outputs
- Parameters
- Code

**Verilog Module Description**
- Name
- Inputs, Outputs
- Parameters
- Code

---

M.H. Perrott
**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

- Seamless Verilog Support
  - Verilog PLI Code
  - PLI Header Code
  - PLI to C++ Signal Conversion
  - Call C++ Top Module (for one time step)
  - C++ to PLI Signal Conversion

- Seamless Matlab Support
  - Matlab Mex Code
  - Mex Header Code
  - Mex to C++ Signal Conversion
  - Call C++ Top Module (for many time steps)
  - C++ to Mex Signal Conversion

Fast C++ Simulation

- CppSim Code
  - Loop
  - Call C++ Top Module
  - Record Probed Signal Values
  - If (Final Simulation Sample)
  - Break

C++ Code Is Easily Embedded In Other Simulators

- C++ Class for Module 1
- C++ Class for Module 2
- C++ Class for Module 3
- C++ Class for Submodule 1
- C++ Class for Submodule 2
- C++ Class for Submodule 3
- C++ Class for Submodule 4
- C++ Class for Module 4
- C++ Class for Module 5
- C++ Class for Module 6
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
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
Analog Modeling in CppSim
Transfer function calculation is tedious, but simulation is fast.

Example: use Filter class
- Specify with ‘s’ polynomials of numerator and denominator
- Run by using `inp()` function of object to update output

Filter `filt cocina(num(s),den(s),...)`

\[ vout = filt \_ sig. \text{inp}(vin) \]
Adding Noise to the Model

- Easy to create Noise objects
  - Specify with distribution (i.e., “gauss” for Gaussian)
  - Run by using inp() function of object to update output
- A bit painful to derive all of the transfer functions…
More Complicated Circuits

- **Switched capacitor circuits** are common in filters, ADCs
  - 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…
A linear network with switches can be represented as a state-space model with switch dependent matrices.
- An equivalent unilateral flow block is created.
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 $1/T_s > 20 \times \text{bandwidth of switch settling time}$
Supported Electrical Elements in CppSim

- Resistor
- Capacitor
- Inductor
- Electrical Transformer
- Mutual Inductors
- Voltage-Controlled Voltage Source (VCVS)
- Current-Controlled Current Source (CCCS)
- Electrical Diode
- Electrical Switch
- DC Voltage
- DC Current
- DC Voltage with Noise
- DC Voltage with Noise Square
- DC Current with Noise
- DC Current with Noise Square
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 modules are not sensitive to loading
  - Allows CppSim to automatically separate into sub-networks

Code modules preferred to achieve fast simulation speed

Filter \texttt{filt1(“K”, “1+1/wo*s”, ...)}
\texttt{vout = filt1(vinp-vinm)}
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
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?
CppSim allows one level of hierarchy for coupled networks
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
Summary of Analog Modeling in CppSim

CppSim Code Modules
- Require unilateral flow but allow arbitrary analog functions including nonlinearity, filtering, hysteresis, etc.

Electrical Element Modules
- Enable straightforward modeling of linear networks with switches and diodes
  - User simply creates schematic level representation
  - State-space model of network automatically created
- Fast speed retained by keeping network sizes small
  - De-coupled networks are automatically separated
  - Instances are decoupled unless they are electrical elements
- High accuracy retained for time-based circuits
  - Constant time step allows straightforward FFT analysis
  - Double_interp protocol enforced for electrical switches
Digital Modeling in CppSim
**Code Modules: CppSim or Synthesizable Verilog**

- **CppSim modules utilize bool signals**
  - Correspond to integer vectors whose elements are 0 or 1
- **Verilog modules must be synthesizable in CppSim**
  - Note: full support of Verilog in VppSim
Getting and Setting Boolean Signal Values (CppSim)

- **Bool signals:** integer vectors with element values of 0 or 1
  - Support functions such as `get_elem()`, `set_elem()`, etc.
  - For convenience: `get_decimal_value()`, `set_decimal_value()`
    - Restricted to 32-bit values

```cpp
a_dec = a.get_decimal_value();  // full bit range (a[2:0])
b_dec = b.get_decimal_value(3,1); // limited bit range (b[3:1])
b_bit1 = b.get_elem(1);          // get b[1]

y.set_decimal_value(15);        // full bit range (y[5:0] = 15)
r.set_decimal_value(21,7,2);    // limited bit range (r[7:2] = 21)
r.set_elem(8,1);                // set r[8] = 1
```
Implementing Clock Edge Based Processing

- **timing_sensitivity**: clk must be of type bool
- **EdgeDetect**: clk must be of type double_interp
**EdgeDetect() versus timing_sensitivity: for VppSim**

**EdgeDetect (simplified)**

```vhdl
// 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:**

```vhdl
// 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)
Basic conventions supported
- Iterated instance: xi1<2:0>
- Bus: a<2:0>
- Bundle: a<1>, b<1:0>

Key rules for bused signals:
- Code modules: buses only valid for type bool
  - Exception for electrical_element: modules:
    - Declare as bool, but actual type becomes double
- Schematic signals: buses can be any type
VppSim Example: Using Buses 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;
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!
Summary of Digital Modeling

- Verilog or CppSim code modules are supported
  - CppSim simulator: Verilog must be synthesizable code
  - VppSim simulator: Verilog is fully supported
- Key constructs for CppSim modules:
  - `bool` signal type allows bus notation
  - `timing_sensitivity`: advantageous for VppSim simulator
- Buses, bundles, and iterated instances supported
- Care should be taken to avoid introducing timing quantization noise
  - Conversion of `double_interp` signals to type `bool` leads to loss of high resolution timing information of edges
Final Points

- 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
    - Flexible output storage for Matlab or GTKwave
  - 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
  - Top schools and industry professionals use it