Behavioral Simulation of Decision Feedback Equalizer Architectures Using CppSim

Matt Park
http://www-mtl.mit.edu/research/perrottgroup/

October 31, 2005
Revised: June, 2008

Table of Contents

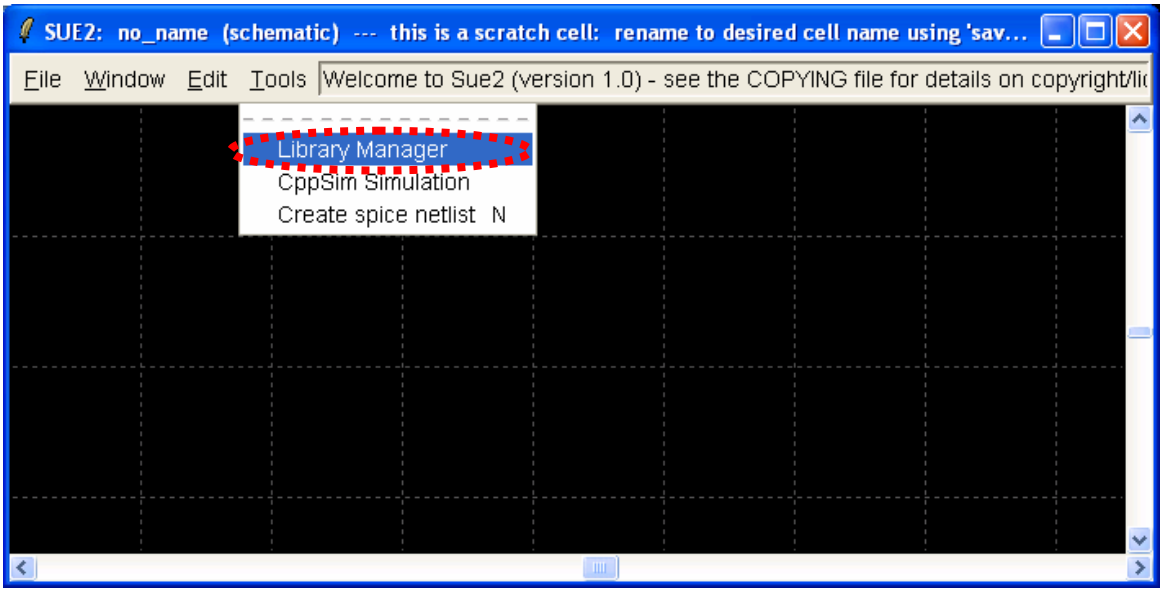
Setup

Download and install the CppSim Version 3 package (i.e., download and run the self-extracting file named **setup_cppsim3.exe**) located at:

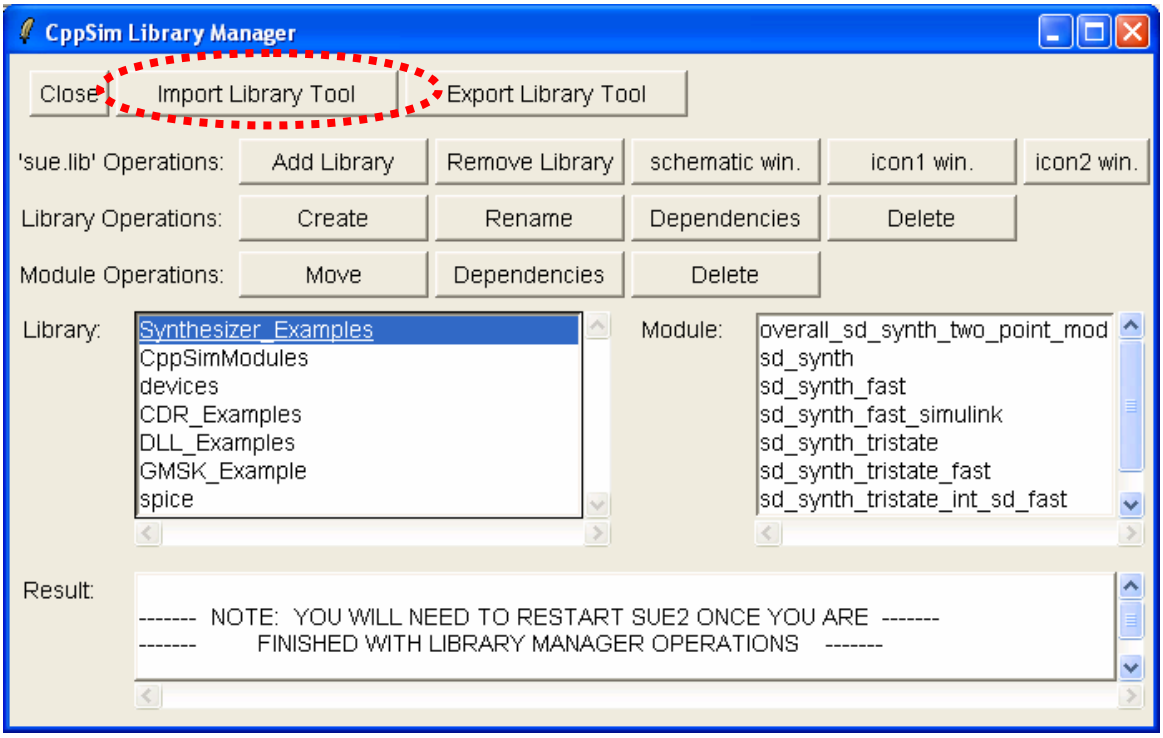
http://www.cppsim.com

Upon completion of the installation, you will see icons on the Windows desktop corresponding to the PLL Design Assistant, CppSimView, and Sue2. Please read the "**CppSim (Version 3) Primer**" document, which is also at the same web address, to become acquainted with CppSim and its various components. You should also read the manual "**PLL Design Using the PLL Design Assistant Program**", which is located at http://www.cppsim.com, to obtain more information about the PLL Design Assistant as it is briefly used in this document.

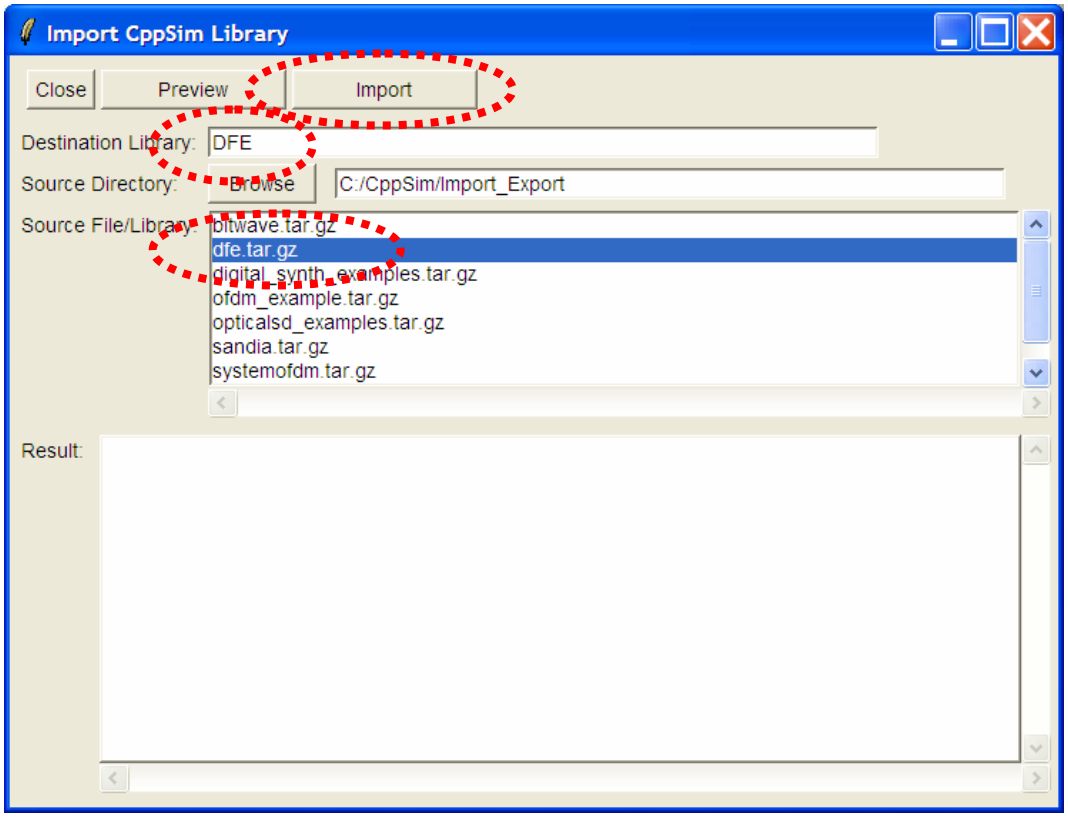
To run this tutorial, you will also need to download the file **dfe.tar.gz** available at http://www.cppsim.com, and place it in the **Import_Export** directory of CppSim (assumed to be **c:/CppSim/Import_Export**). Once you do so, start up **Sue2** by clicking on its icon, and then click on **Tools->Library Manage**r as shown in the figure below.



In the **CppSim Library Manage**r window that appears, click on the **Import Library Tool** button as shown in the figure below.



In the **Import CppSim Library** window that appears, change the **Destination Library** to **DFE**, click on the **Source File/Library** labeled as **dfe.tar.gz**, and then press the **Import** button as shown in the figure below. Note that if **dfe.tar.gz** does not appear as an option in the **Source File/Library** selection listbox, then you need to place this file (downloaded from http://www.cppsim.com) in the **c:/CppSim/Import_Export** directory.



Once you have completed the above steps, restart **Sue2** as directed in the above figure.

Introduction

Most significant improvements in performance in increasingly complex communications systems will arise from architectural innovations. These innovations are only possible when you can quickly and accurately model and simulate the system under consideration. CppSim, initially designed for simulating phase-locked loops, is a free behavioral simulation package that leverages the C++ language to allow very fast simulation of a wide array of system types. The goal of this tutorial is to expose the reader to a non-PLL based system where modeling with CppSim enables the exploration of key design issues, and may inspire new architectures for improved performance.

A. Challenges in High-Speed Serial Link Design

As IC technology continues to scale, multi-Gb/s data rates have become the norm in many high-speed chips. This improvement in on-chip speed has led to a growing interest in developing faster I/O for chip-to-chip communication. Unfortunately, the bandwidth limitations of PCB and backplane traces and wires have not improved as dramatically over the years, largely due to cost considerations. Consequently, channels that were originally designed to support data rates in the 100 Mb/s realm are now being used to transfer data in the 1-10 Gb/s range.

The frequency responses of two typical channels with different terminations are shown in Figure 1. Notice the general low-pass characteristic of the channels caused by capacitances and termination resistances, as well as skin effects and dielectric losses of the PCB. Nulls in the response are due to resonances caused by impedance mismatches and reflections. As shown by the impulse response in Figure 2, the PCB loss is manifest as inter-symbol interference (ISI) in the time domain. Depending on when the pulse is sampled, the receiver can make incorrect decisions, resulting in bit-errors. Hence, for multi-Gb/s data rates to be viable in such channels, it is clear that some form of channel equalization is required[1].

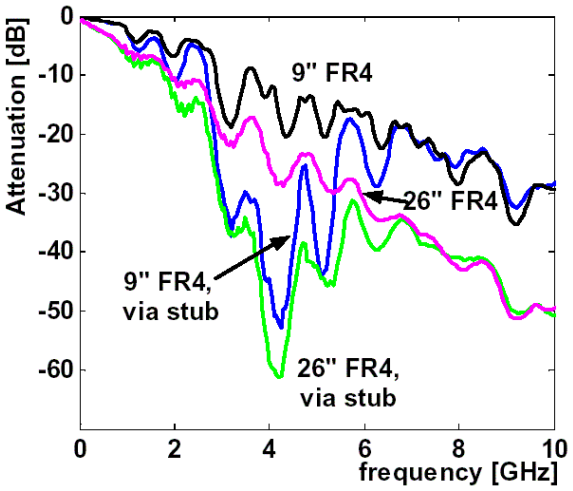


Figure 1: frequency response of backplane channel[1]

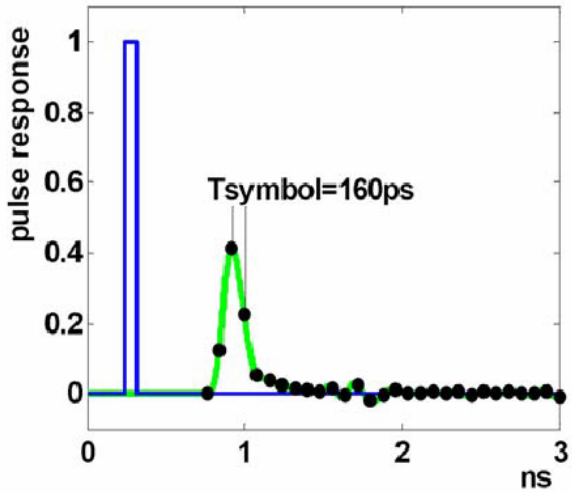


Figure 2: pulse response of backplane channel[1]

B. Signal Restoration Using a Decision Feedback Equalizer (DFE)

Channel equalization can be accomplished through a number of techniques, such as high pass filtering the data at the transmitter and/or receiver (a.k.a., feed-forward equalization or

[1] V. Stojanovic and M. Horowitz, "Modeling and Analysis of High Speed Links", *IEEE Custom Integrated Circuits Conference*, September 2003

FFE), and using tunable, impedance matching networks, just to name a few. The merits and disadvantages of these approaches have been analyzed thoroughly in the literature[2], but are beyond the scope of this basic tutorial.

This tutorial focuses on a particular form of equalization known as *decision feedback equalization*, or DFE. The operation of a DFE can be understood by observing Figure 3. Assuming the channel is linear time-invariant (LTI), ISI can be described as a deterministic superposition of time-shifted smeared pulses. The DFE then uses information about previously received bits to cancel out their ISI contributions from the current decision, as shown in Figure 4. A slightly subtle point is that the DFE can only remove *post-cursor* ISI, that is, the ISI introduced from *past* bits. The architecture cannot remove *pre-cursor* ISI, or ISI introduced from *future* bits (see Figure 3). To eliminate pre-cursor ISI, FFE must be leveraged to generate faster edges.

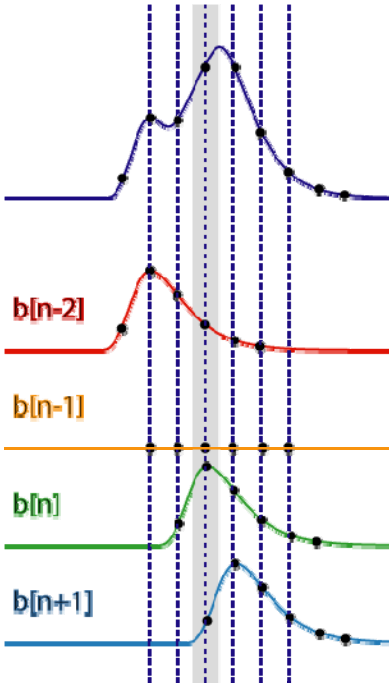


Figure 3: illustration of ISI as a superposition of time-shifted smeared pulses

[2] M. Sorna et al, "A 6.4 Gb/s CMOS SerDes Core with Feedforward and Decision-Feedback Equalization", *ISSCC Digest*, February 2005.

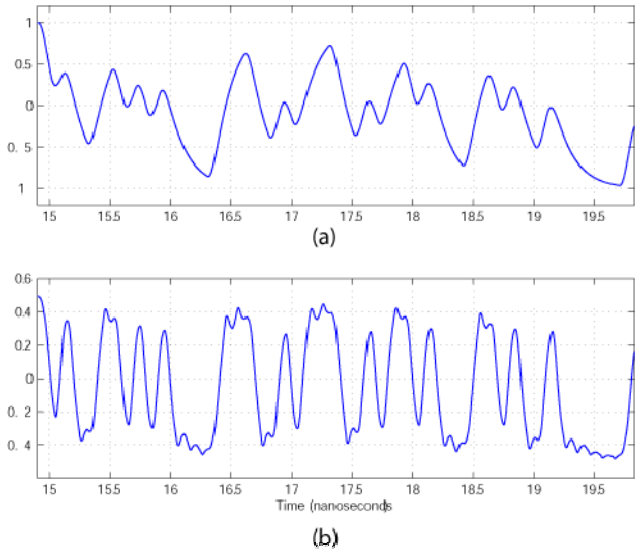
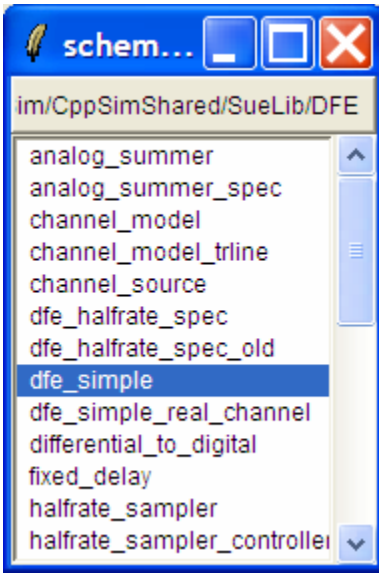


Figure 4: (a) DFE input heavily distorted by ISI, (b) equalized DFE analog output prior to being sampled and latched digitally

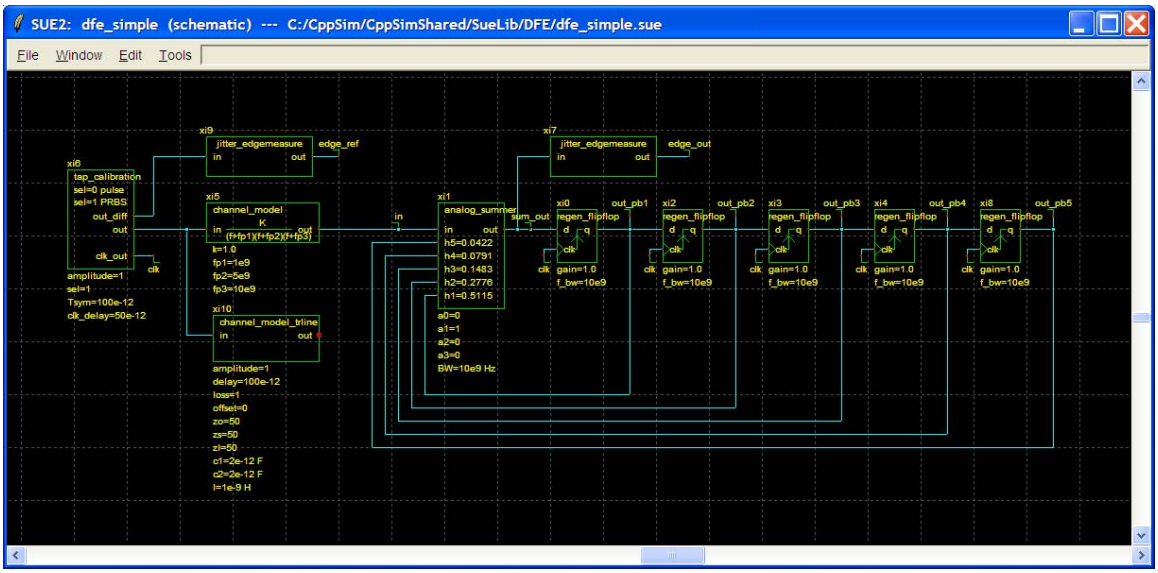
Preliminaries

A. Opening Sue2 Schematics

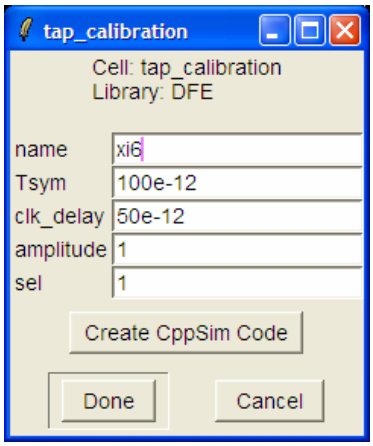
Click on the Sue2 icon to start Sue2, and then select the **DFE** library from the **schematic listbox**. The **schematic listbox** should now look as follows:



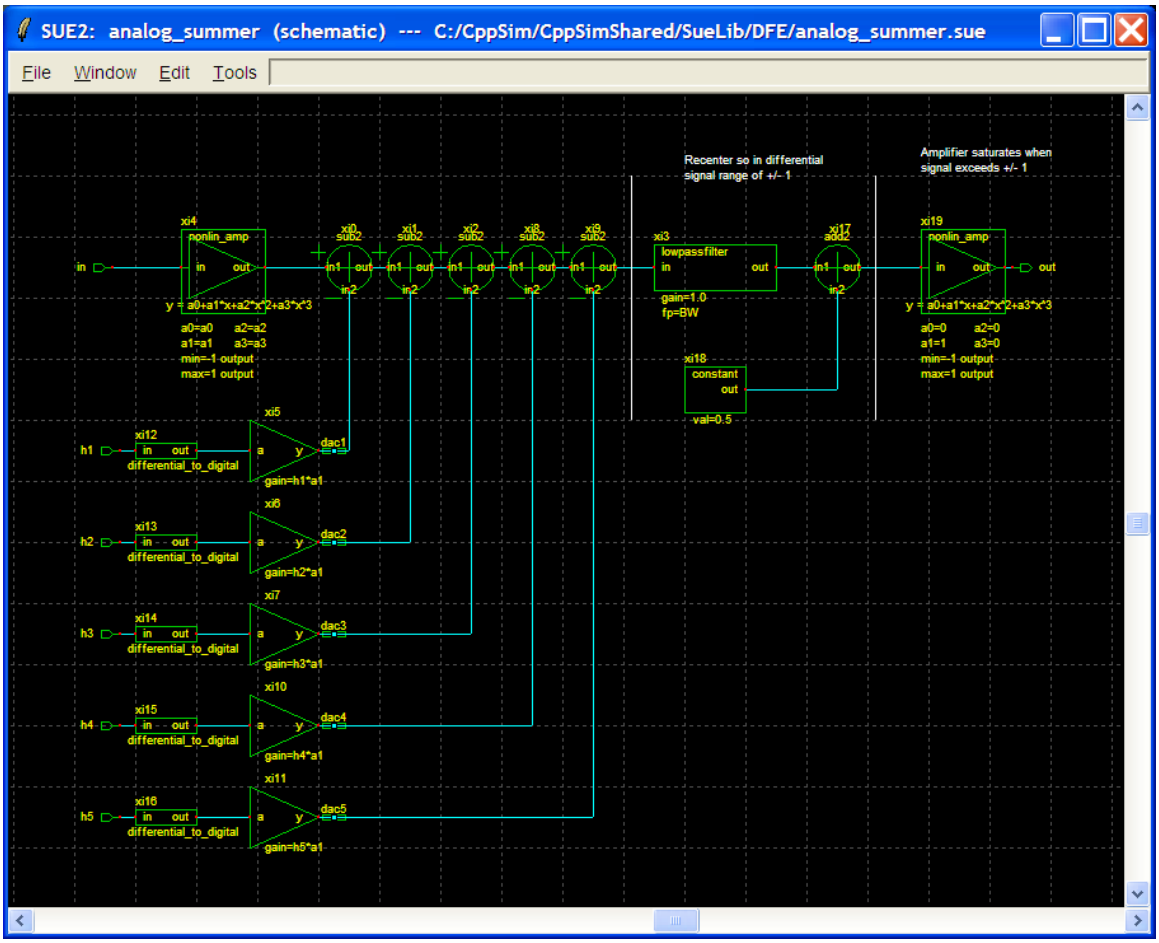
Select the **DFE_simple** cell from the above **schematic listbox**. The Sue2 schematic window should now appear as shown below. Key signals for this schematic include
sum_out: output of analog summing stage
out_pb*n*: DFE output from flip-flop *n*
in: input to the DFE
edge_ref: reference edge for jitter measurement
edge_out: output edge for jitter measurement



Drive the DFE with a PRBS signal by double-clicking on the module labeled **tap_calibration**, and ensure that the signal selection variable **sel** is set to 1 for PRBS, as shown below:



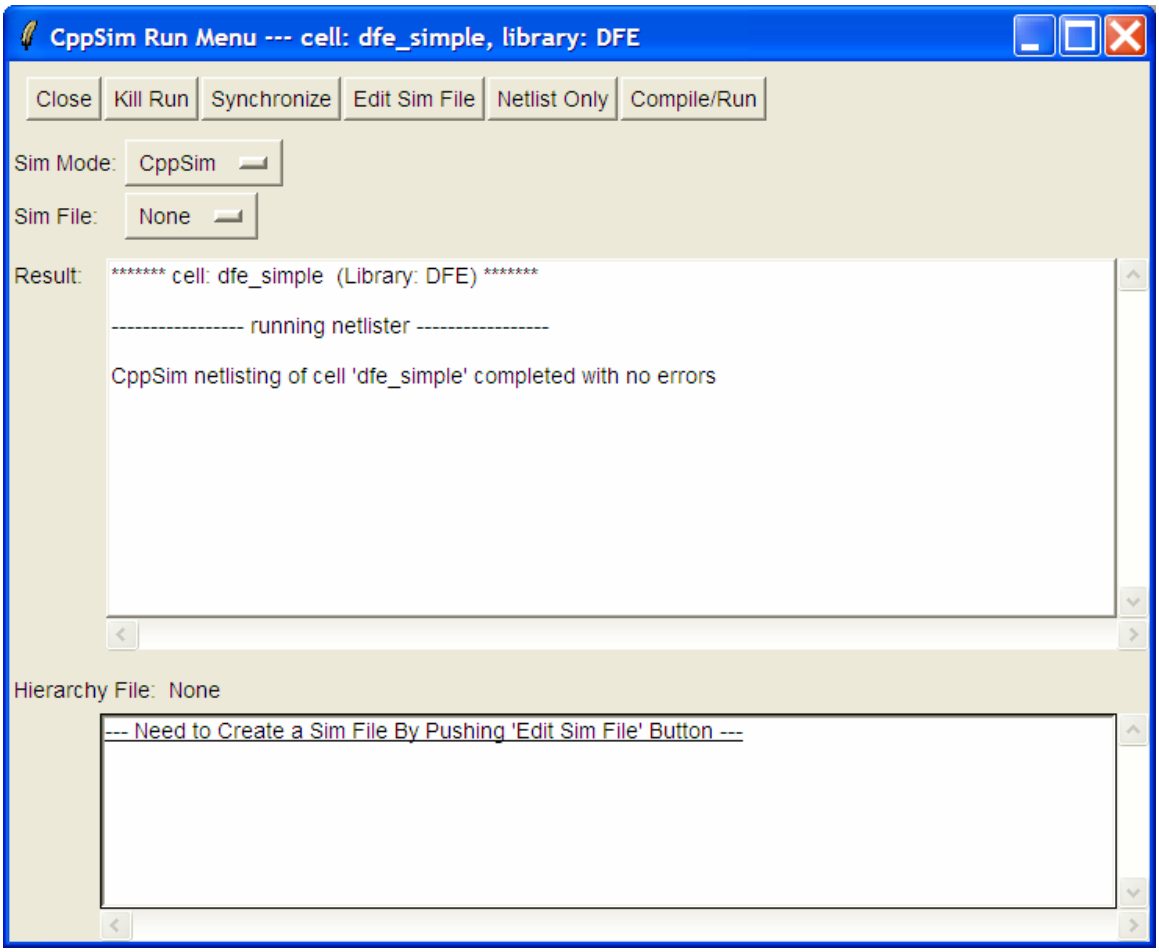
Select the analog_summer icon within the above schematic, and then press e to descend down into the associated schematic. You should now see the schematic shown below.



Press **Ctrl-e** to return to the **DFE_simple** cellview.

B. Running CppSim Simulations

In the Sue2 schematic window, click on the **Tools** text box in the menubar, and then select **CppSim Simulation**. A Run Menu window similar to the one shown below should open automatically. Note that the Run Menu is already synchronized to the schematic that you will be simulating (**dfe_simple**). If for whatever reason this is not the case, click on the Synchronize button in the menu bar, the Run Menu will be synchronized to the schematic in your Sue2 window.



To establish the simulation parameters, click on the **Edit Sim File** button in the menu. An Emacs window should appear displaying the contents of the simulation parameters file (**test.par**). The contents of your **test.par** file should look something like what is shown below:

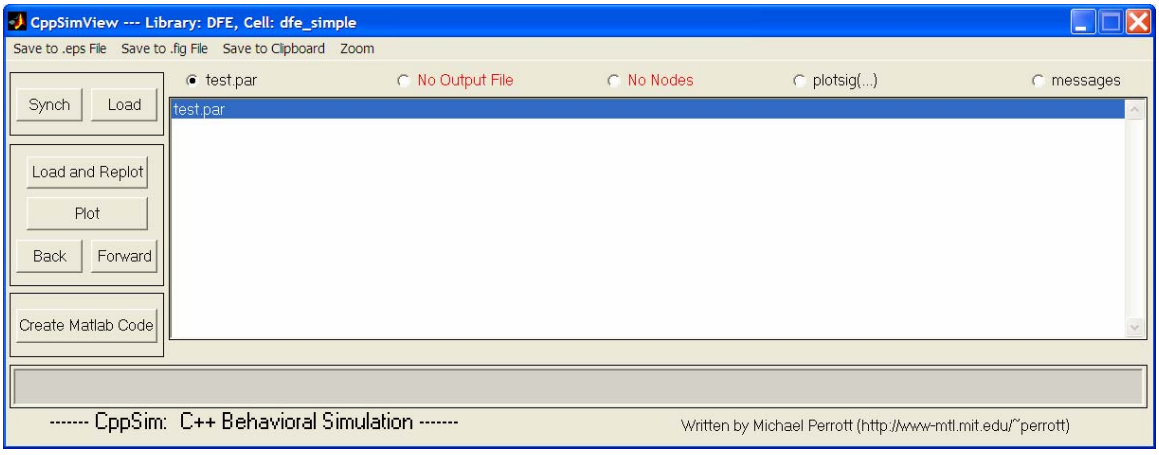
```
////////////////////////////////////////////////////////////
// CppSim Sim File: test.par
// Cell: dfe_simple
// Library: DFE
////////////////////////////////////////////////////////////

// Number of simulation time steps
// Example: num_sim_steps: 10e3
num_sim_steps: 10e4
```

```
// Time step of simulator (in seconds)
// Example: Ts: 1/10e9
Ts: 1/1000e9

// Output File name
// Example:  name below produces test.tr0, test.tr1, ...
// Note: you can decimate, start saving at a given time offset, etc.
//    -> See pages 34-35 of CppSim manual (i.e., output: section)
output: test

// Nodes to be included in Output File
// Example: probe: n0 n1 xi12.n3 xi14.xi12.n0
probe: in sum_out clk out_pb1 edge_ref edge_out

////////////////////////////////////////////////////////////
// Note:  Items below can be kept blank if desired
////////////////////////////////////////////////////////////

// Values for global parameters used in schematic
// Example: global_param: in_gl=92.1 delta_gl=0.0 step_time_gl=100e3*Ts
global_param:

// Rerun simulation with different global parameter values
// Example: alter: in_gl = 90:2:98
// See pages 37-38 of CppSim manual (i.e., alter: section)
alter:
```

When you are finished, you can close the Emacs window by pressing **Ctrl-x Ctrl-c**. To launch the simulation, click on the menu bar button labeled **Compile/Run**.

Plotting Time-Domain Results

Double-click on the CppSimView icon to start the CppSim viewer. The viewer should appear as shown below – notice that the banner indicates that it is currently synchronized to the **DFE_simple** cellview. If this is not the case, Sue2 and CppSimView can be synchronized by clicking the **Synch** button on the left-hand side of the CppSimView window.

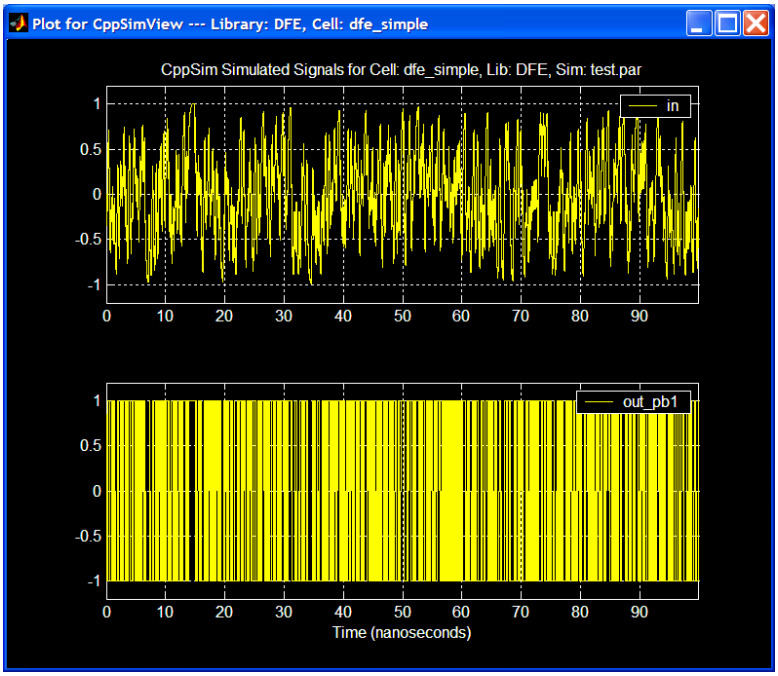


To view the simulation results, first click on the radio button titled **No Output File**. Immediately after this button is clicked, the radio button will instead display the output

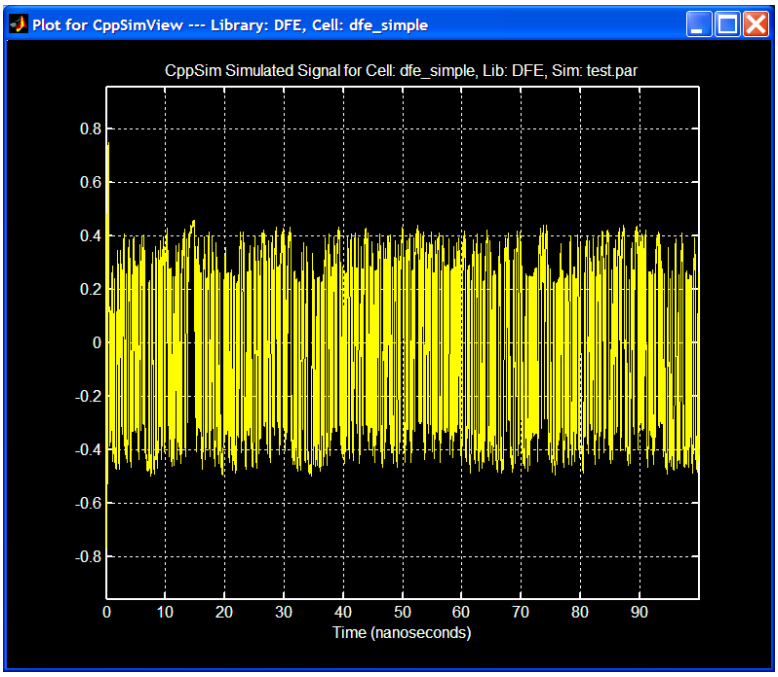
file's name, **test.tr0**. Next, click on the Load button on the left-hand side of the CppSimView window. Once this button is pressed, the **Nodes** radio button will be filled in, and the probed nodes will be listed, as shown below.

A. Output Signal Plots

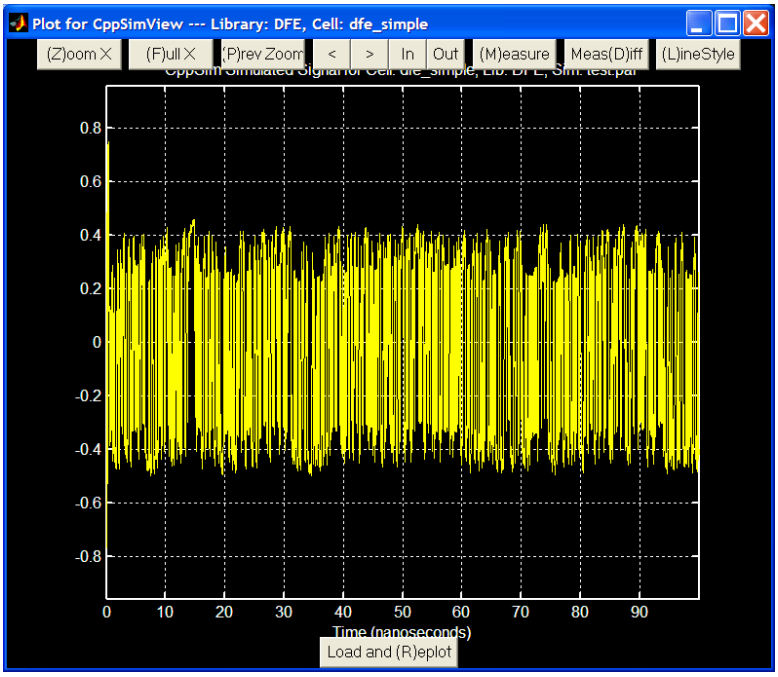
The input data is a PRBS data stream at 10 Gb/s.
In the CppSimView window, double-click on signals **in** and **out_pb1**. You should see plots of the DFE input and first flip-flop output waveforms as shown below:



Now click on the Reset Node List button in the CppSimView window, and then double-click on signal **sum_out**. You should see a plot of the analog summing amplifier output as shown below:

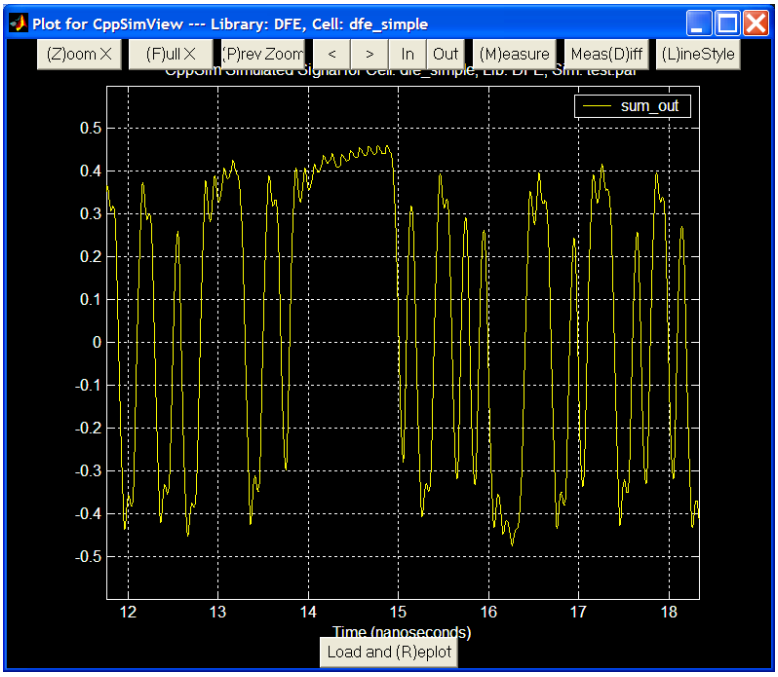


To change the x-axis of the figure (the y-axis automatically scales), hit the **Zoom** radio button on the CppSimView menu-bar. This will cause a series of buttons to appear on the top and bottom of the plot window, as shown below.



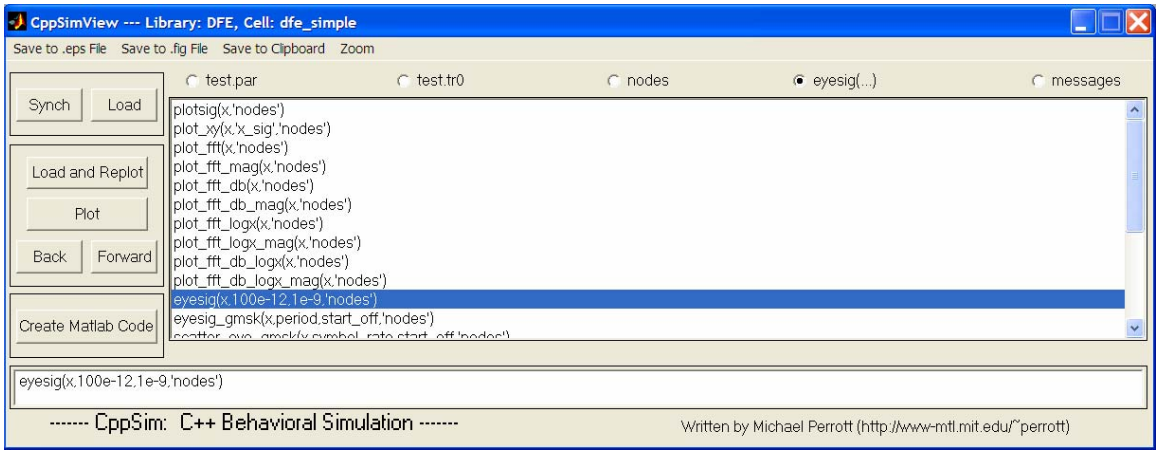
Next click the **(Z)oom X** push-button located at the top of the plot window. Select the desired x-axis range by clicking at the beginning and ending location in any of the plotted signals. The figure will look similar to the figure below. Additionally, you can zoom in and out and pan left and right using the **In** and **Out** and the < and > push-buttons, respectively,

located at the top of the plot figure.

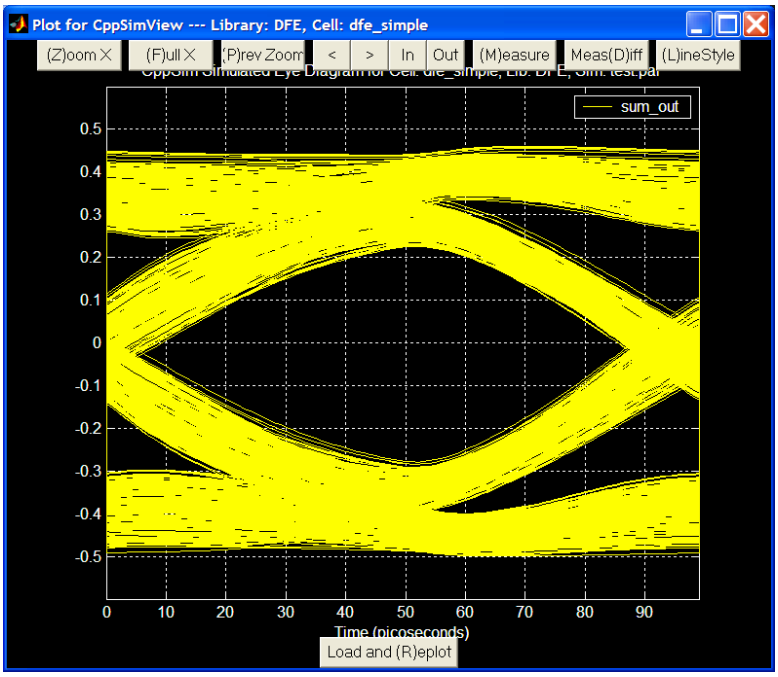


B. Output Signal Eye Diagrams

Click on the **plotsig(…)** radio button in CppSimView and then select the **eyesig(…)** function. Set **period** to be 100e-12 (i.e., one symbol long for the output signal waveforms) and **start_off** to be 1e-9. Since there is no startup-time to the DFE, the **start_off** time is arbitrary; however, if there were a startup transient (e.g., settling of tap weights determined by an adaptive algorithm), then the **start_off** time should be set to a time when the algorithm has completely settled. Hit Return to enter in the parameters. CppSimView should now appear as shown below.

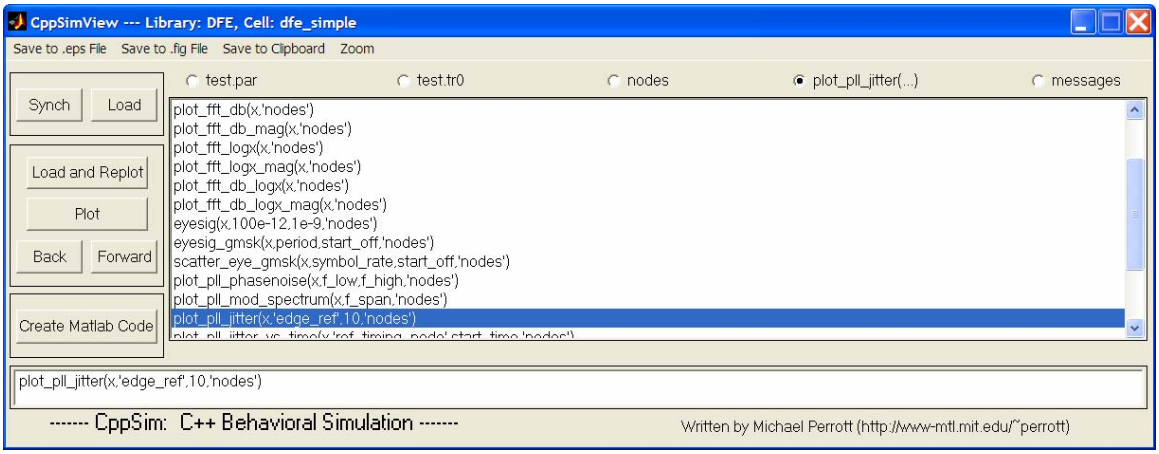


Click on the **nodes** radio button, and then double-click on signal **sum_out**. The eye diagram of the ISI-corrected summing stage output should appear as shown below:

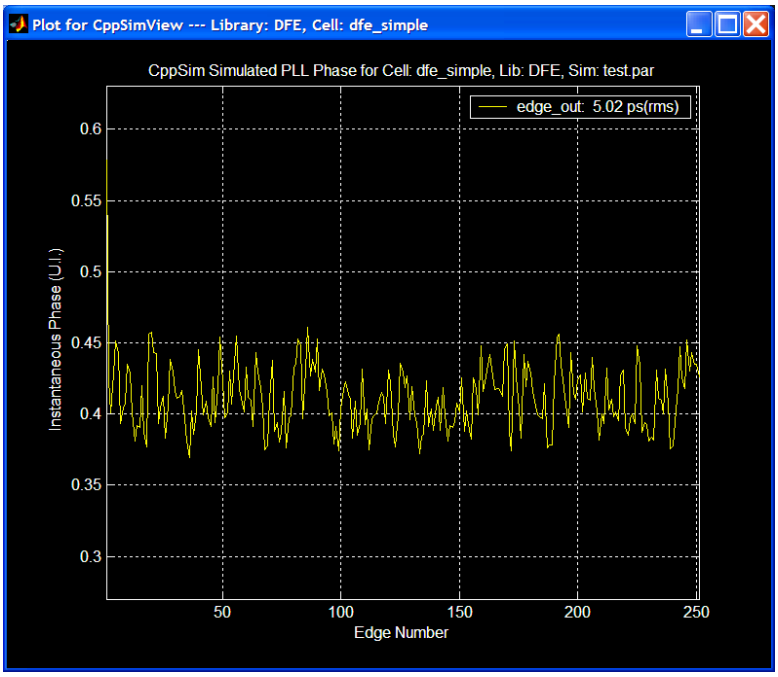


C. RMS Jitter

Now perform the following operations in CppSimView: Click on the **eyesig(…)** radio button and then choose the plotting function to be **plot_pll_jitter(…)**. Set the **ref_timing_node** parameter to **edge_ref** (this is the interpolated reference clock output signal) and the **start_edge** to 10, Hit **Return** to enter the values into the CppSimView function list. CppSimView should appear as shown below:



Click on the **No Nodes** radio button, and then double-click on **edge_out**. A plot of the instantaneous phase of the DFE output should appear, as shown below. The RMS jitter for each signal, in units of mUI (i.e. UI is unit interval, or one data period) is indicated in the legend.



The resulting RMS jitter for the DFE summing node, **sum_out**, is 5.02 ps rms.

Examining Non-Idealities

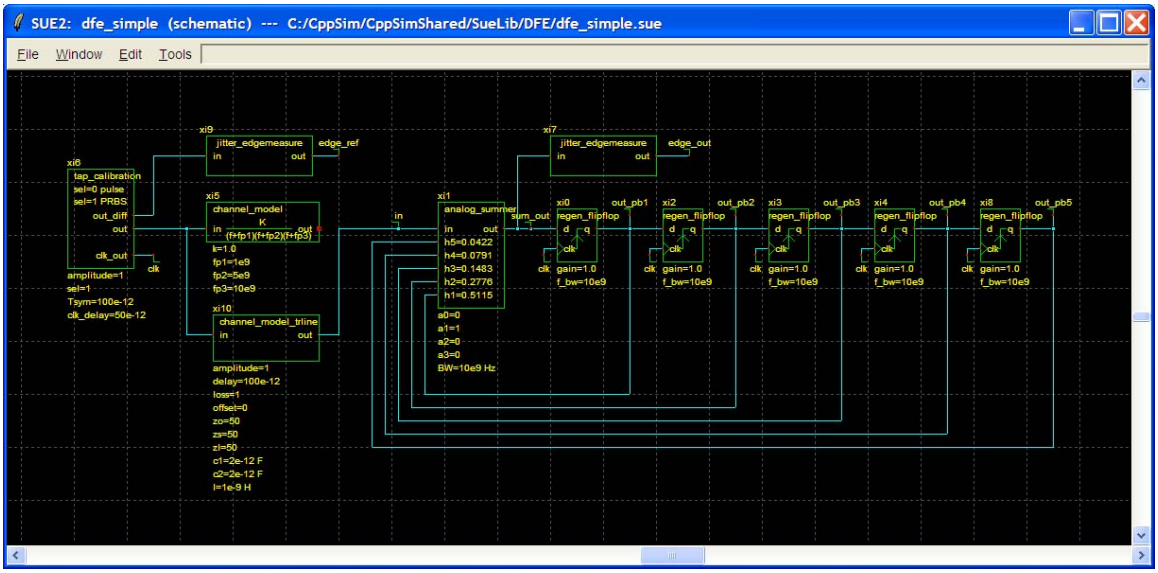
A. Intersymbol Interference (ISI)

As described in the Introduction, if the channel is an LTI system, then ISI can be described as a superposition of time-shifted pulses. CppSim can model the channel in three ways:
channel_model represents the channel as a simple 3-pole low-pass filter
channel_model_trline describes the channel as a transmission line, with the potential for loss, impedance mismatches, and reflections
link_channel uses an impulse response generated from actual channel measurements (see Appendix for more)

B. Reflections

By default, **channel_model** is connected to the DFE input in the schematic **dfe_simple**. To use **channel_model_trline** in the DFE simulations, replace the connection from the output of **channel_model** to node **in**, to the output of **channel_model_trline** to node **in**. The schematic should then resemble what is shown below.

By double-clicking on the **channel_model_trline** cell, parameters concerning transmission line loss and delay, as well as package interface (trace/pad capacitance and bondwire inductance), and source and load impedances can be entered.



C. Non-Linearity and Offset

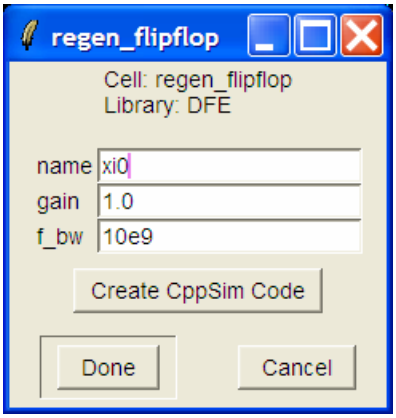
Non-Linearity and offset in the summing amplifier are described using a third-order polynomial description of the amplifier in the form:

$$y = a0+a1*x+a2*x^2+a3*x^3$$

The ideal gain of the amplifier is expressed by the **a1** term, offset is characterized by the **a0** term, and nonlinearity by the **a2** and **a3** terms. These coefficients can be obtained from a regression analysis of the DC transfer characteristics of the summing amplifier in HSPICE or SPECTRE.

D. Gain-Bandwidth Limitations and Clock-to-Q Delays

All cells in the DFE model have a gain and bandwidth parameter that can be adjusted according to the actual design. For example, double-clicking the **regen_flipflop** cell in the **dfe_simple** schematic brings up a window with the parameters **gain** and **f_bw**, as shown below:



Simply change the **gain** and **f_bw** to match the actual circuit design. These parameters can

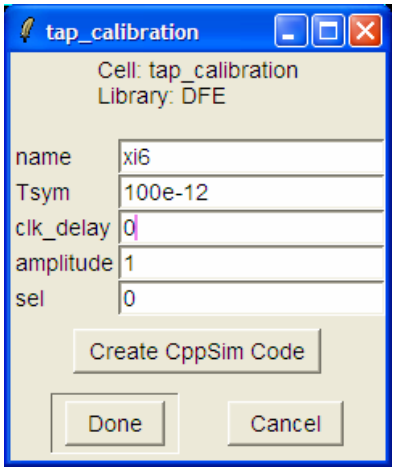
be adjusted to match a particular design corner, allowing for quick determination of the system's robustness over corners.

While the gain-bandwidth parameters partially describe the clock-to-Q delay of a latch or flip-flop, they do not describe the *signal-dependent* delay. This information is captured in the model for the regenerative latch (**regen_latch**) in CppSim. Here, the latch is approximated as an amplifier (**gain**) that linearly settles to a min or max value when clocked. The settling time is then determined by the bandwidth (**f_bw**) of the structure. While the model is not an exact representation of a true latch, it does mimic the signal-dependent nature of clock-to-Q delays, and enables analysis of the impact of latch metastability in the DFE.

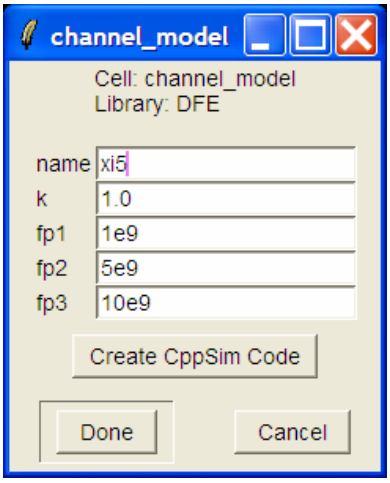
DFE Calibration

The DFE tap coefficients must be tuned to cancel out the ISI contributions of the previous bits. In an actual receiver, this would be accomplished with the aid of an eye-monitoring circuit and adaptive tap-weight adjustment algorithm. For the purposes of studying architectures, however, the taps can be set without either of these blocks by simply monitoring the channel pulse response.

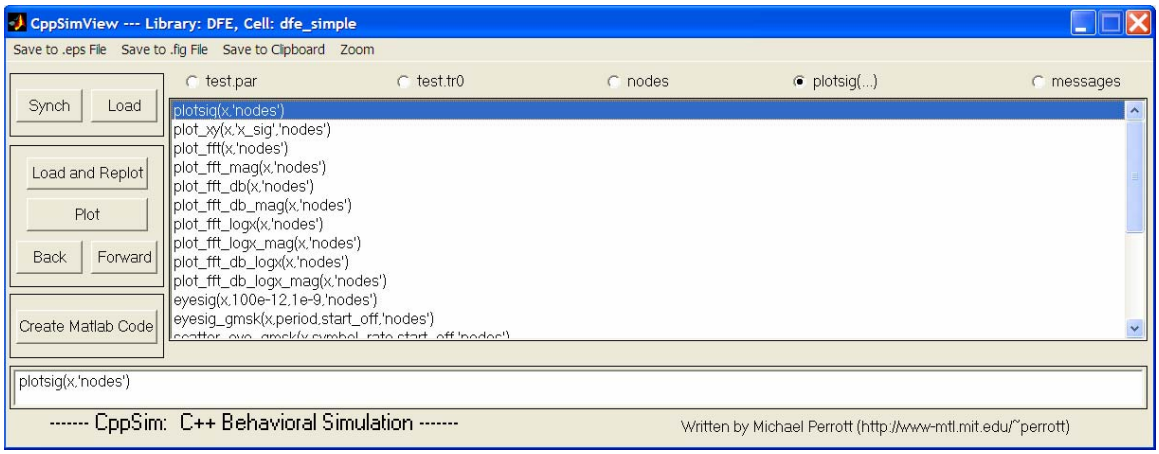
Double-click on the cell **tap_calibration** in the schematic **dfe_simple**. A window should open with prompts for input. Enter values for **Tsym** and **amplitude**, and set **sel** to 0. For example, for a data rate of 10 Gb/s and a data amplitude of 1V, set **Tsym** to 100e-12 and **amplitude** to 1. Setting **sel** to 0 generates a pulse for the duration of one symbol. **clk_delay** describes when the DFE samples the data relative to the ideal data rising edge, and can be set to 0 for the time being (ideally, this sampling point would be determined by a CDR). When you are finished, the tap_calibration window should resemble what is shown below:



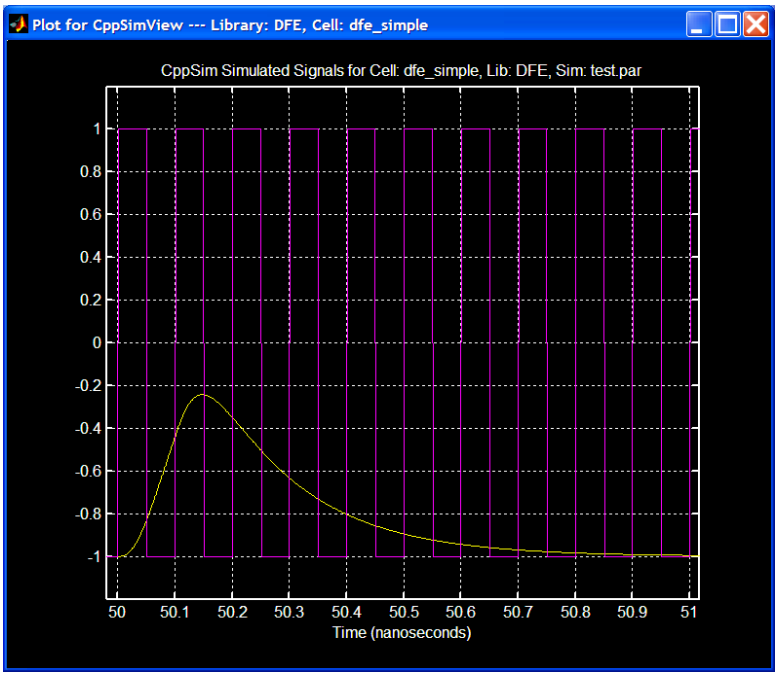
Reconnect the **channel_model** cell to the DFE input, and double-click on the cell to modify its properties. A window with prompts for inputs should open, as shown below:



Set **k**, **fp1**, **fp2**, and **fp3** to 1.0, 1e9, 5e9, and 10e9, respectively. Save changes in Sue2, and click on the **Compile/Run** button In CppSimView. When CppSim finishes simulating, click on the **plot_pll_jitter(…)** radio button, and select the **plotsig(…)** plotting option. The CppSimView window should look like this:



Click on the Nodes radio button, and replace the **'nodes'** text in the **plotsig(…)** statement with '**in,clk'**, and press the **Plot** button. A plot window should open displaying both the **in** and **clk** signals on the same axis. Since the pulse is hidden by all the clock transitions, click on the **zoom** radio button at the top of the CppSimView window, and use the **(Z)oom X** option in the plot window to zoom into the time segment from roughly 50ns to 51ns. The plot window should look something like what is shown below:



This purpose of this plot is to determine what values of the pulse amplitude the DFE will sample at the rising edge of **clk**. Actual DFE eye-monitoring circuits will try to adjust the clock phase relative to the data such that the peak of the eye is sampled. An equivalent operation adjustment can be done here by delaying rising edge of **clk** so that it is approximately coincident with the peak of the pulse response. For this particular channel, the plot indicates that a delay of 50 ps is sufficient. In the **dfe_simple** schematic window in Sue2, double click on the **tap_calibration** cell, and enter 50e-12 for **clk_delay**.

Now that the clock is aligned with the pulse response peak, it is now possible to calibrate the tap weights. In the CppSimView window, click on the **Edit Sim File** button and change the **output** statement to the following:

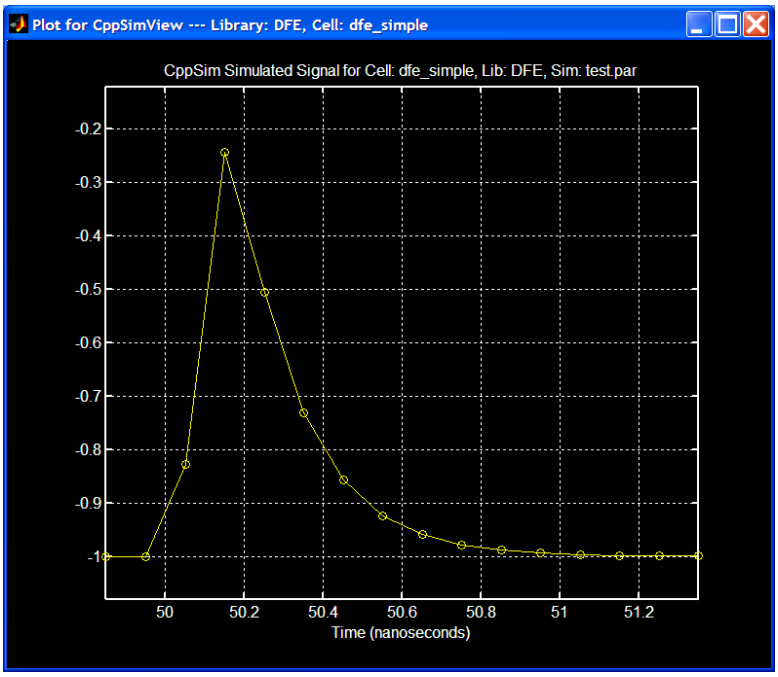
```
output: test trigger=clk
```

Remove **clk** from the list of probed nodes by modifying the probe statement to:

```
probe: in sum_out out_pb1 in edge_out edge_ref
```

The trigger statement will sample the signals listed in the probe statement only at the rising edge of clk, simplifying the tap weight measurements. Save changes, and close the Emacs window, and click on **Compile/Run**.

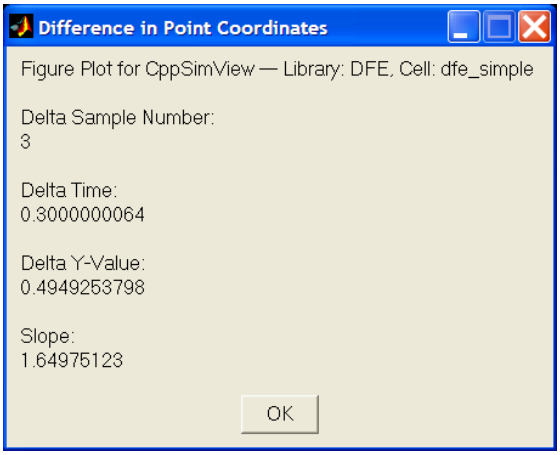
When the CppSim simulation finishes, ensure that the **plotsig(…)** plotting function is selected. Click on the **No Nodes** radio button, and double-click on **in**. A plot window will open showing the sampled pulse response. Use the **(Z)oom X** tool to zoom into the region from 50 ns to 51 ns. To see the individual sample points, click on the **(L)ineStyle** button at the top of the plot window. The plot window should now look like this:



Sweeping the sampled pulse response from left to right, the following observations can be made (note that -1 corresponds to a "zero" value in the plot of the differential signal):

The first non-zero sample is pre-cursor ISI
The second non-zero sample is the peak of the pulse response
The third non-zero sample is the first post-cursor ISI (i.e., h1)
The fourth non-zero sample is the second post-cursor ISI (i.e., h2)
Etc.

To measure the amplitude of the sampled values, use the (M)easDiff button at the top plot window. Click on a sample of zero value (i.e., -1) with the left mouse button and right click in the particular sample of interest. A red line will connect the two samples, and left clicking again will bring up a pop-up window displaying the difference information, as shown below:



The tap weight is then equal to the Delta Y-Value. For example, the value of h1 was determined to be approximately 0.495. Repeat this procedure until all tap weights have been determined. Finally, double click on the cell **analog_summer** in the **dfe_simple** schematic, and enter in the recorded tap values h*n*.

DFE Architectures

A. Prototypical DFE

While the DFE illustrated in the schematic **dfe_simple** works well in a functional sense, it is not necessarily the most elegant or efficient structure, especially in high-speed applications. In particular, observe that all amplifiers and flip-flops must operate at the maximum data rate. This places stringent requirements on the latch setup and hold times, and even stricter requirements for the settling time of the feedback signals at the summing amplifier output node (See Figure 5):

$$t_{CLK2Q} + t_{pd,h1\text{-}hN} + t_{sum} + t_{setup} < 1 \text{ U.I.}$$

Where t_{CLK2Q} is the clock-to-Q delay of the flip-flop, $t_{h1\text{-}hN}$ is the propagation delay through the tap (h1-hN), t_{sum} is the summing amplifier propagation delay, and t_{setup} is the flip-flop setup and hold time. When parasitic and wiring capacitances are included in the design, these rigid timing requirements may be extremely difficult to satisfy across corners, or require a power consumption and device area that is prohibitive.

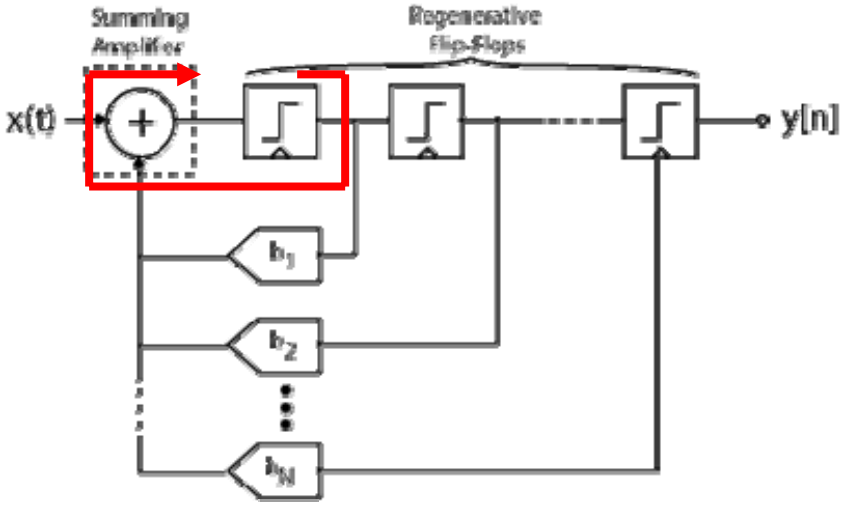


Figure 5: critical path (red line) in basic DFE architecture has only 1 U.I. to settle

B. Half-Rate DFE with One Tap of Speculation

A more elegant and efficient DFE architecture that relaxes the timing requirements by a factor of two, while still achieving equalization at the maximum data rate, is shown below:

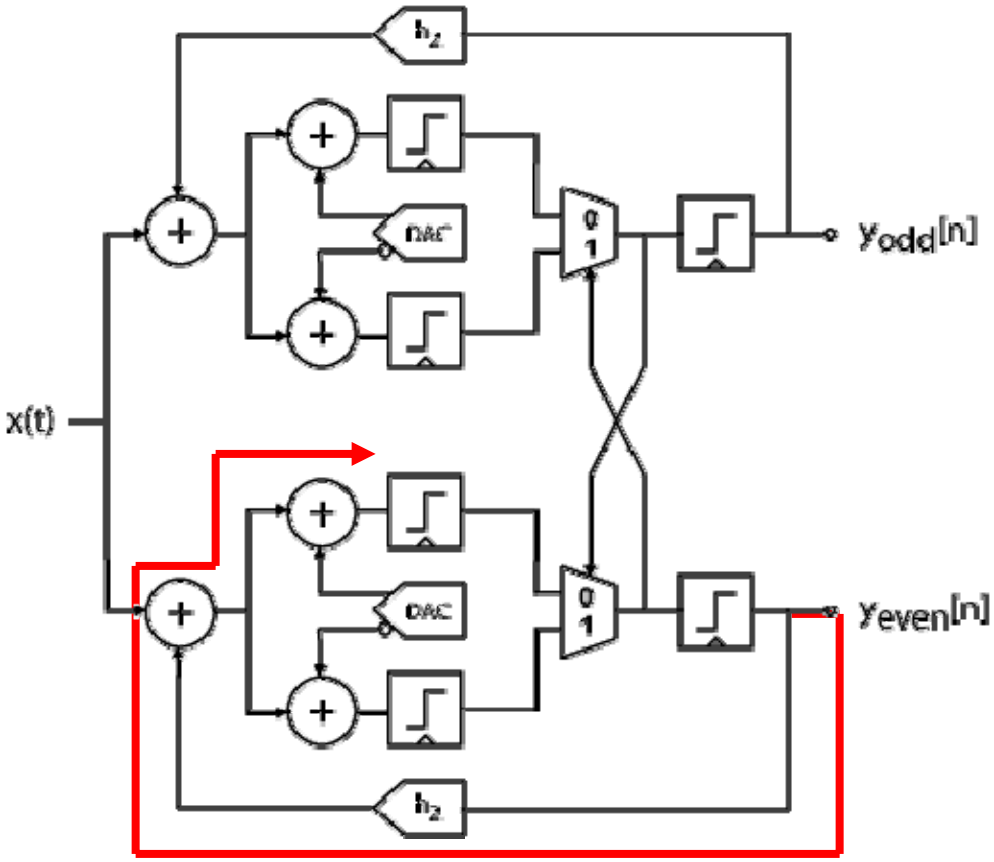


Figure 6: Alternate DFE architecture employing half-rate clocking and one tap of speculation to relax timing. Critical path (red line) now has 2 U.I. to settle.

This particular DFE employs two techniques to achieve this feat: *speculation* and *half-rate clocking*. In speculation (also called "loop-unrolling"), decisions are made for both cases in which the previous bit was a "1" and a "0"; this is accomplished by having two analog summers, two regenerative flip-flops, and a mux (see Figure 7). The h1 tap now functions as a DC offset and is not dynamically switched. A second flip-flop at the output then drives the mux select, and effectively picks the "correct" decision, thus ignoring the "wrong" decision.

The advantage in speculation is that the potentially significant h1 feedback delay is eliminated (i.e., the loop is unrolled). Presumably, the clock-to-Q delay of the second stage latch is minimal since it is latching a digital input. However, it is important to note that the critical timing path in Figure 7 is still the same: the feedback signal from the mux output to the summing amplifier still has only 1 U.I. to settle.

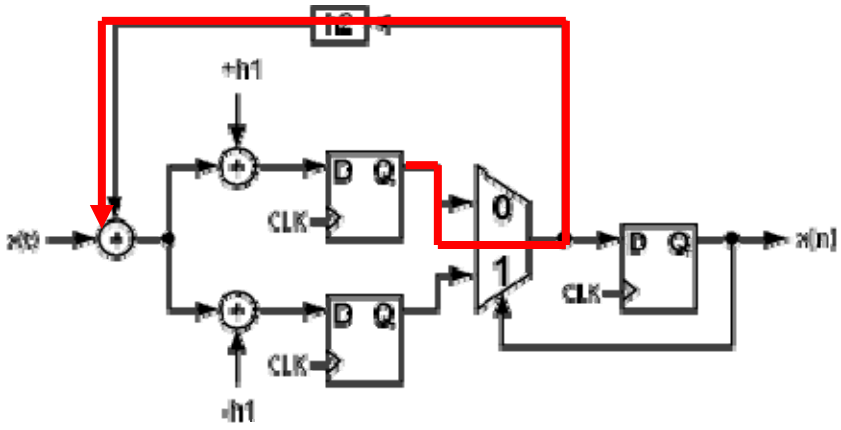


Figure 7: DFE with one tap of speculation. Critical path (red line) has 1 U.I. to settle.

In order to allow 2 U.I. for the critical timing path to settle, a second technique of half-rate clocking is employed (see Figure 6). Here, a half-rate clock drives two duplicate paths at opposite clock phases. Decisions "ping-pong" back and forth between the two paths, and generate even and odd bit sequences. Due to the ping-pong action of the two halves, the critica time path now has 2 U.I to settle:

$$t_{CLK2Q} + t_{pd,h2} + t_{MUX} + t_{sum} + t_{setup} < 2 \text{ U.I.}$$

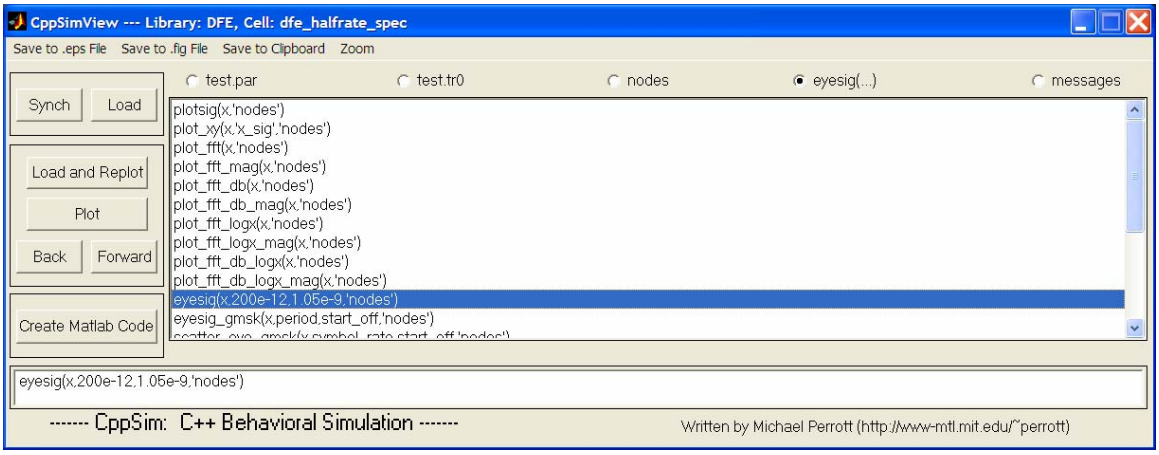
The architecture has the added benefit that all circuitry operates at half the data rate. The penalty is that there are now twice as many devices. However, the power and area costs do not necessarily scale in a one-to-one fashion with the prototypical DFE of Figure 5, making the speculative, half-rate architecture an attractive alternative.

C. Simulating and Analyzing Half-Rate Architecture in CppSim

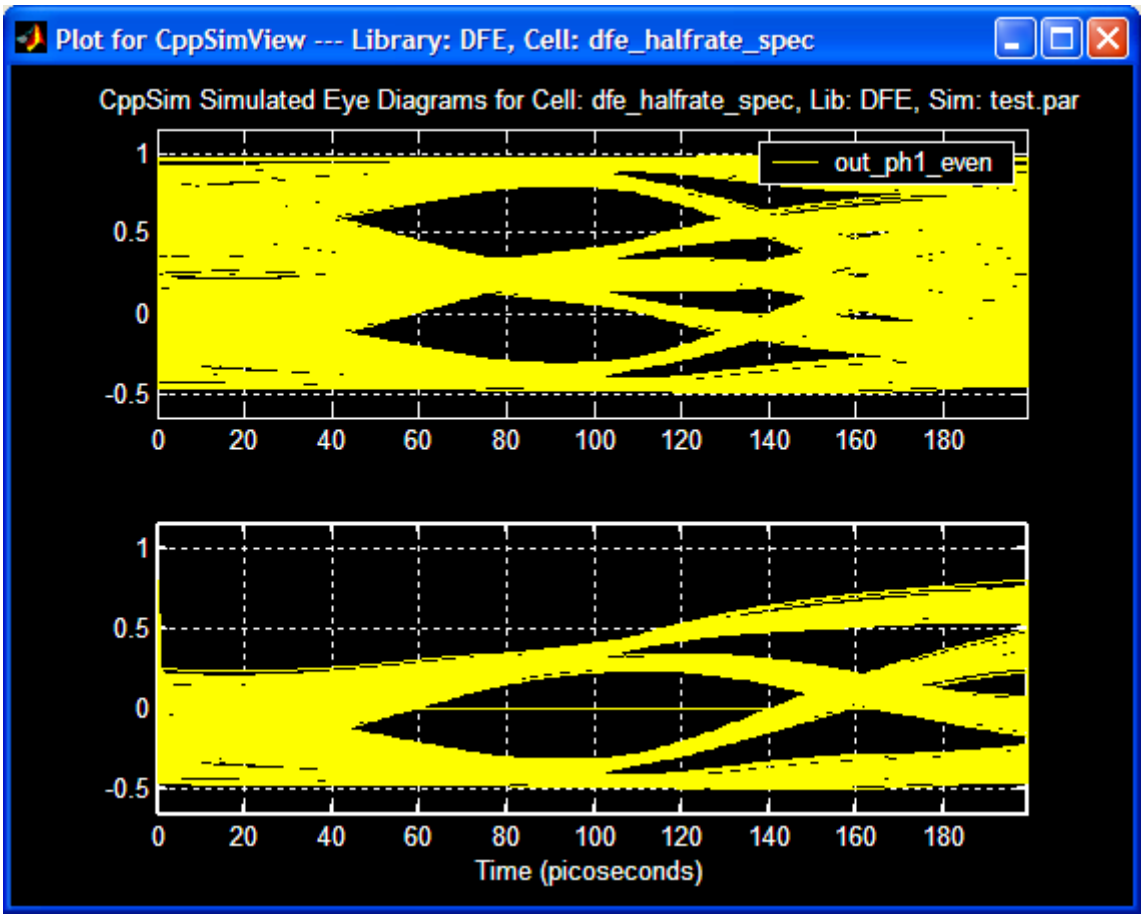
The Sue2 schematic for the behavioral model of the half-rate DFE with one tap of speculation can be found under the **DFE** library, in the schematic **dfe_halfrate_spec**. The schematic should appear as shown below:

The **dfe_halfrate_spec** schematic is very similar to the **dfe_simple** schematic. Indeed, the channel models are identical, as are the regenerative flip-flops, signal sources (PRBS and pulse), and the calibration procedure. The significant difference between the two models concerns generating the eye diagrams. Observe that the half-rate architecture with one-tap of speculation now has four analog summer outputs to handle the four different combinations (odd/even, 1/0 previous bit). At a given moment in time, the output of the summers may be valid, and at other times invalid depending on what the previous bit truly was. Therefore, the eye diagram plotting function must only plot the summer output when the output is valid. Since the PRBS input is known a priori, the summing stage output can be sampled when it is known to be valid. This is accomplished by the **halfrate_sampler** blocks at the bottom of the schematic.

To see the effect of the **halfrate_sampler** block, go to the CppSim Run Menu window and click the **Synch** button, and then click the **Compile/Run** button. When simulation finishes, click on the **eyesig(…)** radio button and replace **period** and **start_off** in the eyesig(…) plot expression with **200e-12** and **1.05e-9**, respectively, and hit enter. The CppSimView window should now look like this:



Next, click on the **No Nodes** radio button, and double click on **out_ph1_even** and **out_ph1_even_sel**. A plot window should open displaying what is shown below:



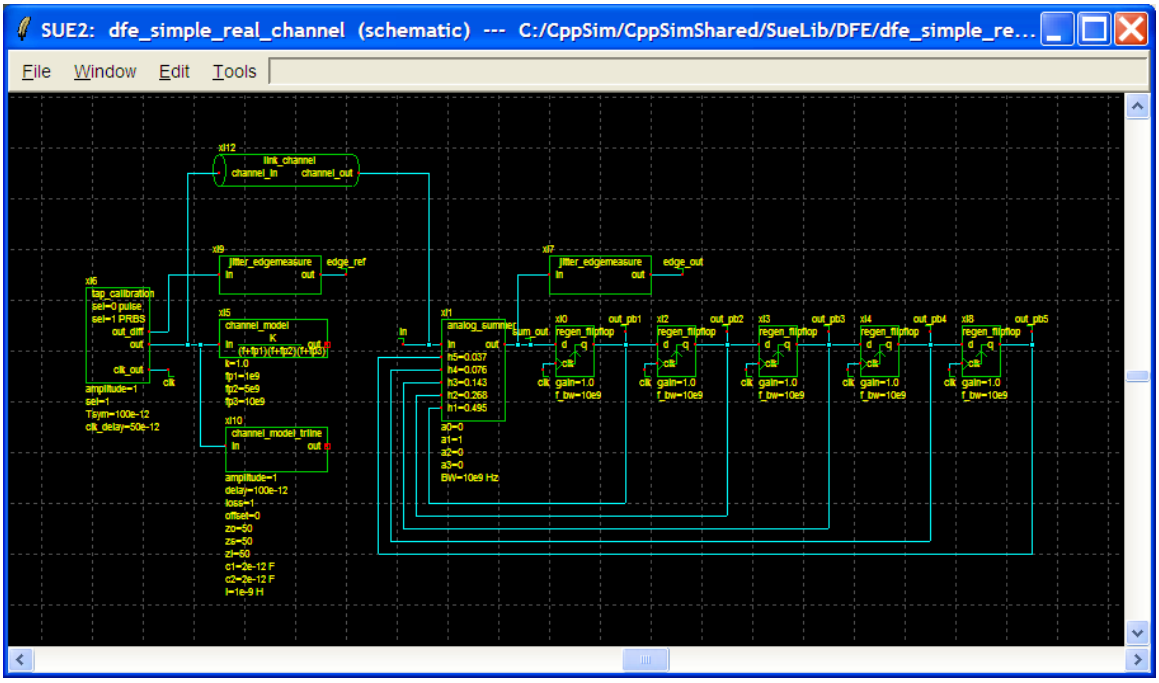
Comparing the plots, the **halfrate_sampler** block is able to recover the eye by setting the summer output to zero when it is invalid. This is visible in the **out_ph1_even_sel** plot as a solid yellow line at zero.

Conclusion

This tutorial covers the basic issues related to behavioral simulation of a simple DFE receiver example using CppSim. In particular, the reader has been introduced to the tasks of running CppSim simulations, plotting eye diagrams, and performing jitter measurements, as well as viewing the impact of non-idealities, such as inter-symbol interference, amplifier offset and non-linearity, and latch metastability and signal dependent clock-to-Q delay. Finally, the advantages of a half-rate DFE with speculation over the prototypical DFE architecture have been analyzed.

Appendix: Generating Channel Impulse Response from Measured Data[3]

An impulse response generated from measured S-parameter data stored in an S4P file can also be included in the DFE behavioral simulations. As an example, open the schematic **dfe_simple_real_channel** in the DFE library. The resulting schematic window should now look like this:



An example S4P file, **channel_data.s4p**, is included in this distribution for illustrative purposes, and is located in **c:\CppSim\SimRuns\DFE\dfe_simple_real_channel**. In order to simulate the DFE, the impulse response of the channel (S21) must be extracted from S4P data using the MATLAB script **link_channel_gen.m**[3]. The script appears below:

```
%%% link_channel_gen.m - generates channel impulse response based on
%%% measured data stored in an *.s4p file
clear all; close all; clc;
```

[3] Special thanks to Prof. Vladimir Stojanovic for the MATLAB scripts and channel models!

```
%%% Create channel response for the simulator %%%
channelName='C:\CppSim\SimRuns\DFE\dfe_simple_
real_channel\channel_data.s4p';
mode='s21';
[f,H]=extract_mode_from_s4p(channelName,mode);
figure(1)
subplot(211),plot(f*1e-9,20*log10(abs(H)),'b');
xlabel('frequency [GHz]');
ylabel('Transfer function [dB]');
grid on;
Tsym=100e-12; %%% Symbol Rate: e.g., Tsym = 1/fsym = 1/10 Gb/s
Ts=Tsym/100; %%% CppSim internal time step, also used to sample
%%% channel impulse response
imp=xfr_fn_to_imp(f,H,Ts,Tsym);
nsym_short=300*100e-12/Tsym;  %%% persistence of the impulse response
                              %%% tail in the channel in terms of the
                              %%% number of symbols
imp_short=imp(1:floor(nsym_short*Tsym/Ts));
figure(1)
subplot(212), plot(imp,'b.-');
hold on;
plot(imp_short,'r.-');
ylabel('imp response');
legend('long','short');
%%% Create the channel impulse response taps file, with appropriate
%%% sampling according to Ts used in the sims
save link_channel.dat imp_short -ascii;
```

In order for the script to generate the proper impulse response, the data rate (**Tsym**) and the internal time step (**Ts**) need to match those in the CppSim behavioral model. For example, in **dfe_simple_real_channel**, the default data rate is 10 Gb/s and the internal time step in the **test.par** file is 1 ps; consequently, Tsym = 100e-12 and Ts = Tsym/100 = 1e-12. Once this information is entered, the **link_channel_gen.m** script can be executed from the MATLAB command line, and an output data file, **link_channel.dat**, containing the impulse response of the channel will be created.

To use an alternate S4P file, the **channelName** variable in the above MATLAB script should be redefined as the complete path of the file; that is:

```
channelName='C:\CppSim\SimRuns\DFE\dfe_simple_real_channel\<your_channel
_data>.s4p';
```