

# CppSim/VppSim Primer

## Version 5.3

Michael H. Perrott  
<http://www.cppsim.com>

May 24, 2014

Copyright © 2004-2014 by Michael H. Perrott  
All rights reserved.

---

### **Table of Contents**

Introduction .....	2
Installation and Setup .....	4
A. Windows.....	4
B. Mac OS X .....	4
C. Linux.....	4
Known Bugs .....	5
The Basics of Running CppSim Simulations .....	6
A. Running a CppSim Simulation in Sue2.....	6
B. Editing CppSim Module Descriptions.....	10
C. Editing Simulation Files (i.e., test.par files) .....	12
CppSim versus VppSim for Simulation of Verilog Modules .....	13
A. Running a CppSim Simulation with a Verilog Module .....	13
B. Utilizing GTKWave to view CppSim Simulations .....	19
C. Running a VppSim Simulation with a Verilog Module .....	27
D. Key Distinguishing Features between CppSim and VppSim.....	32
The Basics of Including Electrical Elements in CppSim/VppSim Simulations.....	34
A. Including Electrical Elements within System Descriptions .....	34
B. Key Constraints When Using Electrical Elements .....	36
Using The CppSim Library Manager.....	40
A. Basic Operations.....	40
sue.lib Operations.....	42
Library Operations and Module Operations.....	43
B. Exporting CppSim Libraries.....	43
C. Importing CppSim Libraries Generated from the Export Library Tool .....	44
D. Importing CppSim (Version 2) Libraries .....	47
Creating Matlab Mex Functions and Simulink S-Functions .....	48
A. Matlab Mex Function Generation .....	48
B. Simulink S-Function Generation .....	50
Using Python with CppSim.....	53
Using Matlab with CppSim.....	59
A. Running CppSim Simulations in Matlab.....	59
B. Creating Matlab Mex Functions .....	60
C. Creating Simulink S-Functions .....	61

Killing Runaway CppSim Simulations .....	61
More Details on CppSimView .....	63
A. Preliminary Setup .....	63
B. Selecting an Output File .....	64
C. Basic Plotting and Zooming Methods .....	65
D. Advanced Plotting Methods .....	67
E. Saving Plots to EPS files, FIG files, or the Windows Clipboard.....	68
F. Choosing Different Plotting Functions .....	68
G. Using the plot_pll_phasenoise(...) Plotting Function .....	69
H. Using the plot_pll_jitter(...) Plotting Function .....	70
More Details on Sue2.....	73
A. Using Navigation and Edit Commands .....	74
B. Creating a New Schematic .....	74
C. Creating an Icon View (And Associated Parameters) For A Given Schematic .....	81
Creating and Running New CppSim Simulations .....	83
A. Creating a New Simulation File for a Newly Created Schematic .....	83
B. Using the eyesig(...) Plotting Function .....	85
C. Using the alter: Statement.....	86
Creating New CppSim Primitives .....	87
A. Creating a Schematic View for the Primitive.....	87
B. Creating an Icon View for the Primitive .....	88
C. Instantiating the Primitive Within a Different Schematic .....	88
D. Running CppSim with the Primitive .....	90
E. Creating Code for the Primitive.....	92
F. Running CppSim with the Primitive (Part II) .....	93
Conclusion.....	96

## **Introduction**

CppSim is a general behavioral simulator that leverages the C++ language to achieve very fast simulation times, and a graphical framework to allow ease of design entry and modification. Users may freely use this package for either educational or industrial purposes without restriction. However, the package and all of its components come with ***no warranty or support***.

To install this package, first download it from the web at <http://www.cppsim.com/download>. Upon extraction, several sub-packages will be installed to perform the various tasks required:

- 1) **Sue2**: a free, open source, schematic capture program that is easy to use.
  - Note: in Linux version, this program as run as command **sue2**
- 2) **CppSimView**: a free waveform viewer for plotting signals produced by CppSim and VppSim.
  - Note: in Linux version, this program as run as command **cppsimview**
- 3) **CppSim Classes**: free, open source, C++ classes to allow easy implementation of common system blocks such as filters, noise generators, and PLL/DLL blocks that leverage the techniques described in “Fast and Accurate Behavioral Simulation of Fractional-N Synthesizers and other PLL/DLL Circuits”, M.H. Perrott, DAC, 2002.
- 4) **net2code** function: free (but not open source) executable routine that plays a central role of CppSim and VppSim as it produces C++ and/or Verilog simulation code from a netlist and module descriptions.

- 5) **Hspice Toolbox for Matlab/Octave**: a free, open source set of Matlab/Octave routines to allow straightforward access to Hspice, Ngspice, and CppSim simulation results within Matlab or Octave.
- 6) **CppSim and Ngspice Data Modules for Python**: a free, open source set of Python classes and routines to allow straightforward access to CppSim and Ngspice simulation results within Python.
- 7) **MinGW C++ compiler**: (Windows only) a free GNU-based C++ compiler used by CppSim to automatically compile the simulation code it produces.
- 8) **MSYS**: (Windows only) a free set of routines that allow use of the “make” facility to compile C++ code.
- 9) **Emacs**: (Windows and Mac OS X only) a free, open source, text editor that is especially convenient for writing C++ code.
- 10) **PLL Design Assistant**: a free (but not open source) design tool that allows straightforward design of phase locked loops and other closed loop systems at the transfer function level.
  - Note: in Linux version, this program runs as command **plldesign**
- 11) **Verilator**: a free, open source tool written by Wilson Snyder to translate synthesizable Verilog code into a C++ class.
- 12) **Icarus Verilog**: a free, open source Verilog simulator written by Stephen Williams, which is used as the primary simulator when running VppSim simulations.
- 13) **NGspice**: a free, open source SPICE simulator based on Berkeley Spice3.
- 14) **GTKWave**: (Windows and MAC OS X only) a free, open source waveform viewer written by Tony Bybell for viewing digital (and analog) signals stored within LXT (and other) files.

This document is intended as a primer that covers basic use of CppSim and VppSim in conjunction with Sue2, CppSimView, GTKWave, Verilator, and Matlab (or Octave). CppSim and VppSim both use Sue2 as the schematic editor for entering designs (a different version of CppSim/VppSim is also available at <http://www.cppsim.com> to support Cadence Composer). Simulations are run within either Sue, Python, Octave, or Matlab. Simulation results are then viewed in CppSimView, GTKWave, Python, Octave, or Matlab. CppSimView is the preferred environment for interactively examining analog signals and GTKWave for interactively examining large sets of digital signals accompanied by smaller numbers of analog signals. Python, Matlab, or Octave, in combination with the CppSim Data module for Python or the Hspice Toolbox for Matlab/Octave, offers much more powerful post-processing capabilities than CppSimView or GTKWave, and is the recommended environment for doing more sophisticated CppSim/VppSim simulations.

While this document covers enough information on CppSim and VppSim to get a good idea of their operation, a more full description of the capabilities and functionality of CppSim and its various sub-packages are provided in the manuals available within the **Doc** menu of Sue2. In particular, the CppSim Reference Manual, **cppsimdoc.pdf**, provides details on the underlying operation of CppSim, information on how to create CppSim module and simulation files, and examples of how to use the various CppSim classes. The Sue2, CppSim Data module for Python, and Hspice Toolbox manuals are provided in the files **sue2\_manual.pdf**, **cppsimdata\_for\_python.pdf**, and **hspice\_toolbox.pdf**, respectively, and are available in the Doc menu of Sue2. Note that there is no separate manual for CppSimView – this document contains a full description of CppSimView.

## Installation and Setup

The CppSim/VppSim framework is available for Windows (Windows 2000 and above), Mac OS X (Lion and above), and Linux (Redhat/Centos Enterprise 5). Of the three distributions, the Windows version is the easiest to install and the Linux version is the hardest to install in terms of their requirements for outside packages. Installation details of each distribution are provided below.

### A. Windows

Go to the CppSim web page <http://www.cppsim.com/download>, and then download the file for Windows (which corresponds to the setup file **setup\_cppsim5.exe**). To install, simply run **setup\_cppsim5.exe** in Windows (i.e., double-click on **setup\_cppsim5.exe** in Windows Explorer) and follow the instructions. To run Sue2 or CppSimView, click on their respective Windows icons once the installation process has completed and Windows has restarted.

Note that some computers require installation of the Microsoft Visual C++ 2008 Redistributable Package (x86) in order to run NGspice. This is a small set of DLL files, and can be downloaded from the Microsoft website at: <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=29>

### B. Mac OS X

Go to the CppSim web page <http://www.cppsim.com/download>, and then download the file for Mac OS X (which corresponds to the file **CppSimMacInstall.dmg**). To install, simply open CppSimMacInstall.dmg by double-clicking on it in the Finder window. A window should appear with four icons (**Sue2**, **CppSimView**, **PllDesign**, and **GTKwave**) as well as the **CppSim** folder. As stated in the window, drag the four icons into the **Application** folder and then drag the **CppSim** folder into your home directory by making use of the Finder window.

Unlike the Windows version, the Mac OS X version does not come with a C++ compiler. Instead, you need to obtain the Command Line Tools of Xcode which are contained in a freely available package from the Apple Developer website. For those who do not wish to login to the Apple Developer website, an alternative approach is to install Xcode from the Apple App store. Once Xcode has completed its installation, you should then start Xcode and select the **Preferences** item under the **Xcode** menu (at the very top of the screen, second menu item from the left). If you then click on the **Downloads** tab of the Preferences window that appears, you will see an option to install the Command Line Tools.

### C. Linux

Go to the CppSim web page <http://www.cppsim.com/download>, and then download the file for Linux (which corresponds to the file **cppsim\_linux\_install.tar.gz**). To install, place the file **cppsim\_linux\_install.tar.gz** into your home directory and then type the following command at the Linux shell prompt:

```
> tar zxvf cppsim_linux_install.tar.gz
```

You should then see the **CppSim** directory as well as the file **cppsim\_bashrc\_file\_example**. Assuming you are content keeping the **CppSim** directory, in your home directory, you simply need to update your `~/.bashrc` file according to the example shown in **cppsim\_bashrc\_file\_example**. If you wish to alter the locations of the CppSim and CppSimShared directories, you need to modify the environment variables **CPPSIMHOME** and **CPPSIMSHAREDHOME** as should be evident upon

examination of the **cppsim\_bashrc\_file\_example** file. Upon making your changes to the `~/.bashrc` file, you may then delete the **cppsim\_bashrc\_file\_example** file.

In the case of Linux, several important packages must be installed to achieve proper operation of CppSim and VppSim:

- **g++**: required to compile and run CppSim and VppSim simulations
  - Available within the standard packages of Redhat/Centos
- **Tcl/Tk**: required by the Sue2 program
  - Available within the standard packages of Redhat/Centos
- **Wine**: required by the CppSimView and the PLL Design Assistant programs
  - Available from <http://www.winehq.org>
- **GTKwave**: a useful waveform viewer for examining signals produced by CppSim/VppSim
  - Available from <http://gtkwave.sourceforge.net>
- **Zlib**: required by Icarus Verilog to support LX2 files for GTKwave
  - Available from <http://zlib.net>
- **Emacs**: a convenient graphical text editor for modifying CppSim and VppSim files
  - Available within the standard packages of Redhat/Centos

## **Known Bugs**

- 1) True library support is lacking in Sue2 right now (i.e., name clashing occurs between cells of the same name even though they may be in different libraries). This issue is taken care of by using the Import and Export tools of Sue2.
- 2) The copy to clipboard operations for CppSimView and the PLL Design Assistant do not work in the Mac OS X and Linux versions.
- 3) The undo command in Sue2 is broken.
- 4) Sometimes the history file of CppSimView gets corrupted and does not allow CppSimView to start. If so, within Windows Explorer, go to the SimRuns directory associated with the current cell of Sue2, and then erase the file called **cppsimview\_history.mat** within that directory. As an example, if Sue2 is currently displaying cell **sd\_synth\_fast** within library **Synthesizer\_Examples**, then delete the file **cppsimview\_history.mat** located within directory **c:/CppSim/SimRuns/Synthesizer\_Examples/sd\_synth\_fast** (where **c:/CppSim** corresponds to the base directory location that CppSim was installed at, and may be different for different machines).
- 5) When using electrical elements, there is a limit on how many parameters can be utilized within a given electrical element “bundle”. If you exceed this limit, you will see an error message:

----- Net2Code Version: 5.2 -----

error in 'add\_param\_value': maximum allowable number of  
parameters exceeded  
to fix, change LENGTH\_PAR\_LIST in source code

To deal with this issue, you should examine whether there are parameters of the electrical elements involved which have fixed values. If that is the case, you can create a new version of the given electrical element primitives (see the “Creating New CppSim Primitives” section of this manual on how to create new modules) with a reduced number of parameters (i.e., set the

parameter values in the cppsim code rather than declaring the parameter). By making use of these new electrical element primitives, you can thereby reduce the number of parameters within a given electrical element cluster, and therefore hopefully avoid the above error message.

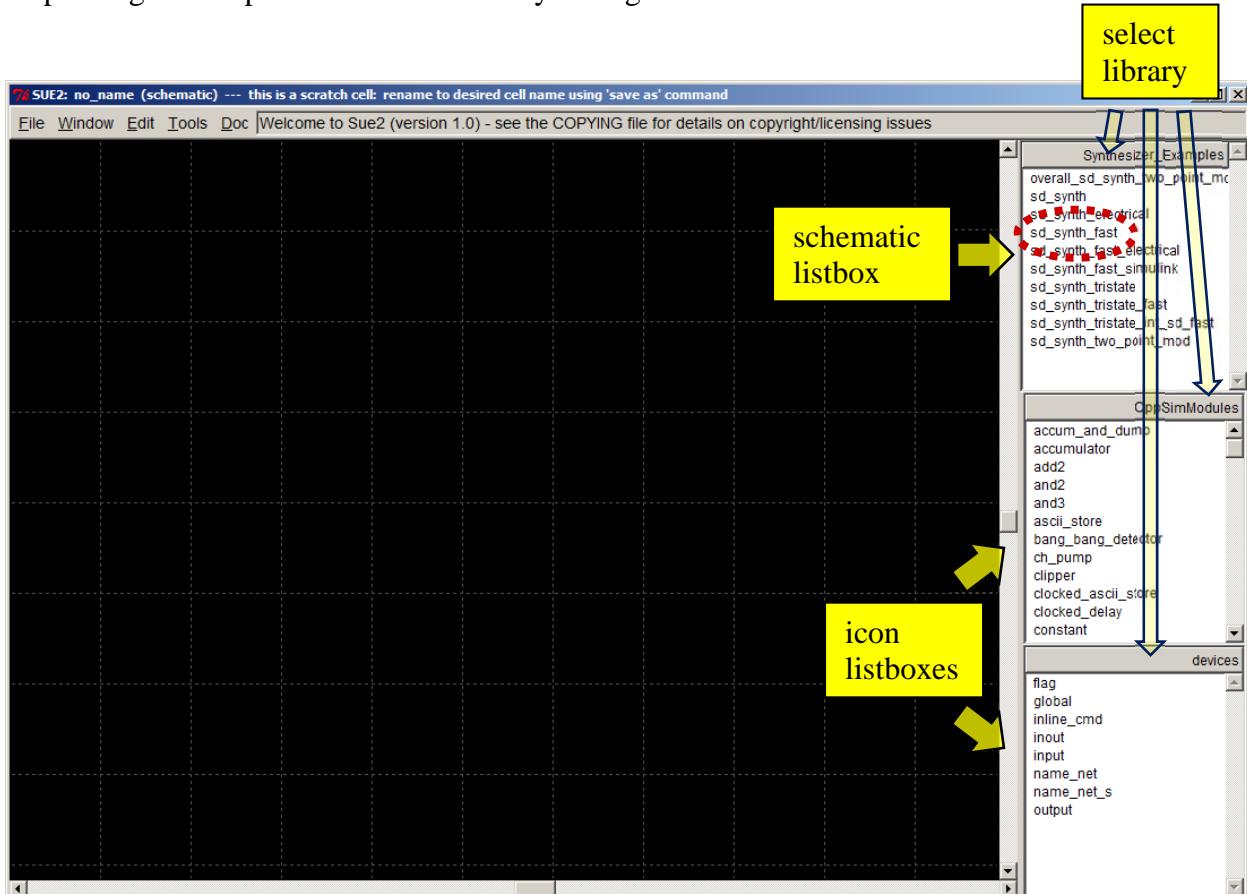
## The Basics of Running CppSim Simulations

To explain the basic operation of running CppSim, let us now walk through an example using **Sue2**. Since **CppSimView** works in conjunction with **Sue2** to view simulation results, we will also step the reader through its operation, as well. For the remainder of this manual, we will use the Windows version for our examples, but the Mac OS X and Linux versions are very similar.

### A. Running a CppSim Simulation in Sue2

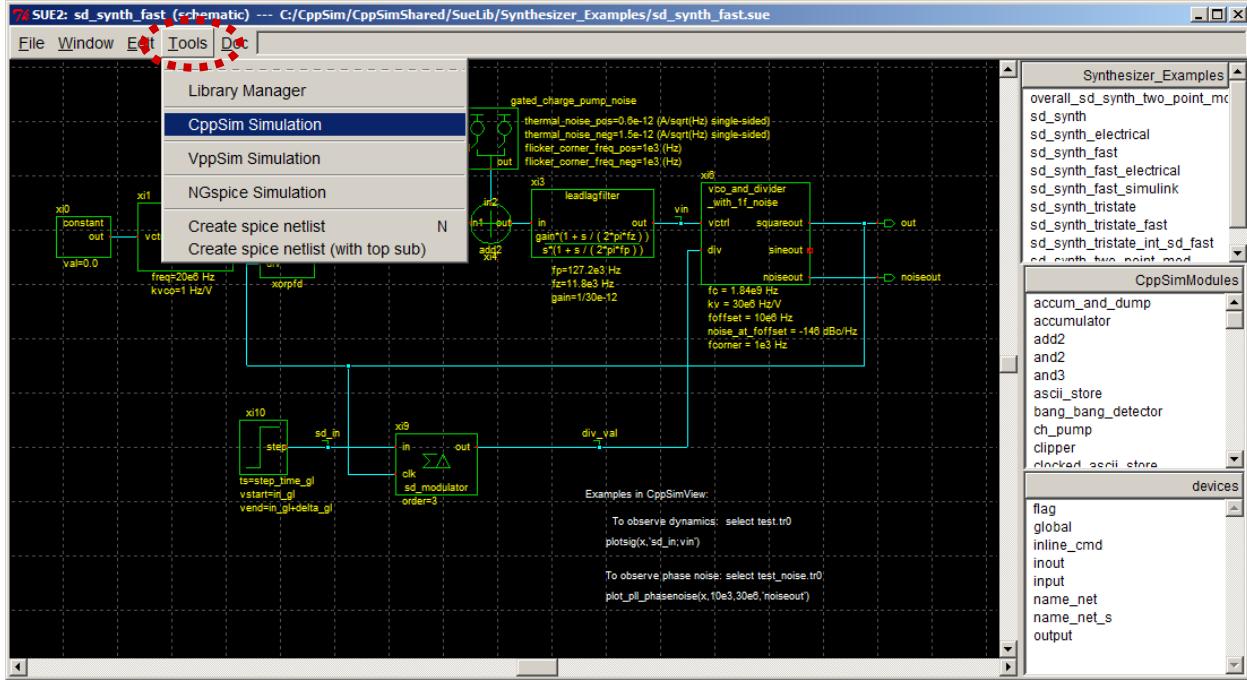
- Start the Sue2 program by performing the following action:
  - Windows:** double-click on the **Sue2** icon on the Windows Desktop.
  - Mac OS X:** double-click on the **Sue2** app in the Applications folder
  - Linux:** at the Linux prompt, run the command **sue2**

You should see a window similar to what is shown below. Note that there is one schematic listbox and two icon listboxes, each of which lists cells from the library that is selected by pushing their top button as indicated by the figure.

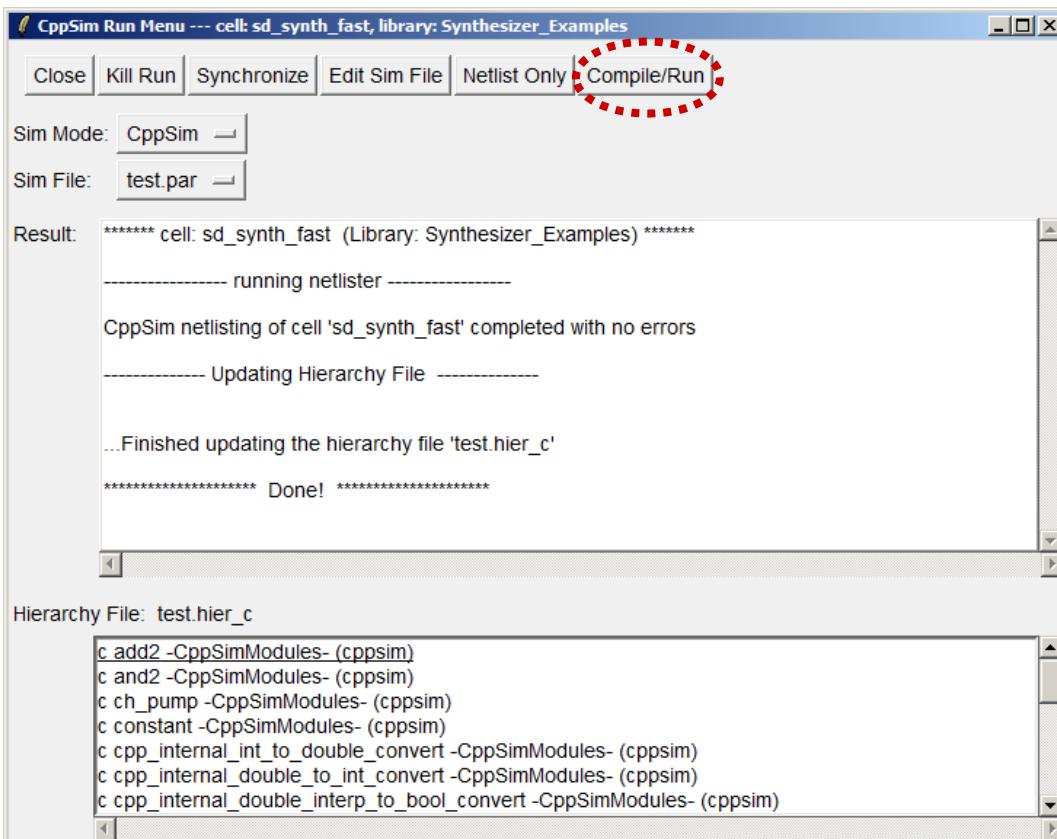


- Select the **sd\_synth\_fast** schematic by clicking on it in the **schematic listbox**. Note that the **icon listboxes** will be used later to add modules to a given schematic. Sue2 should now

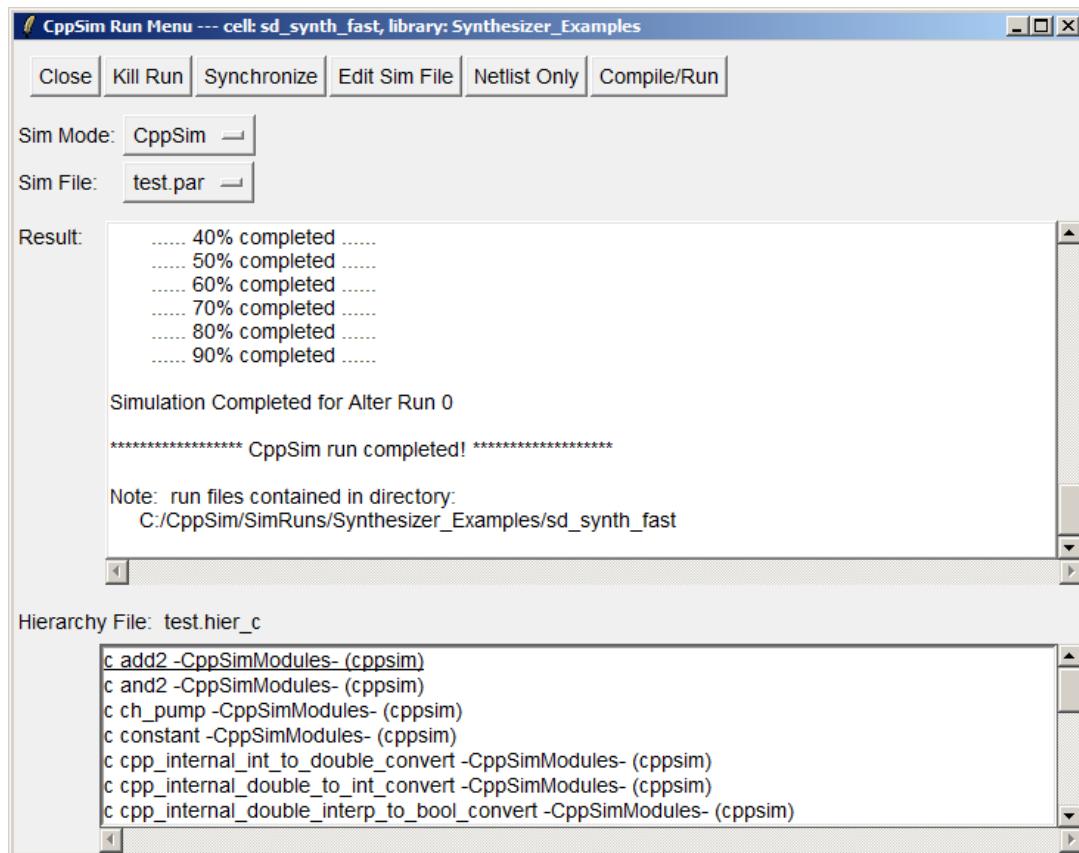
display the **sd\_synth\_fast** schematic as shown below. Note that the **CppSim Simulation** menu item is obtained by clicking on **Tools** (circled in red).



- Clicking on the **CppClassSimulation** menu item, as shown above, yields the **CppClassRun Menu** as shown below. Note the **Compile/Run** button, which is circled in red.

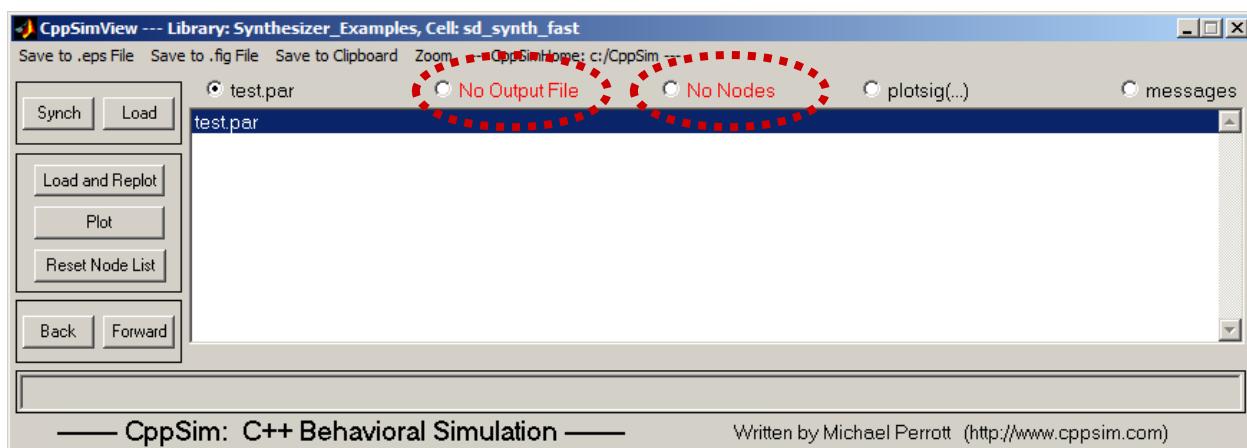


- Run a CppSim simulation on the **sd\_synth\_fast** cell by clicking on the **Compile/Run** button shown in the above figure. You should see some warning messages, and then finally the window should appear as shown below.

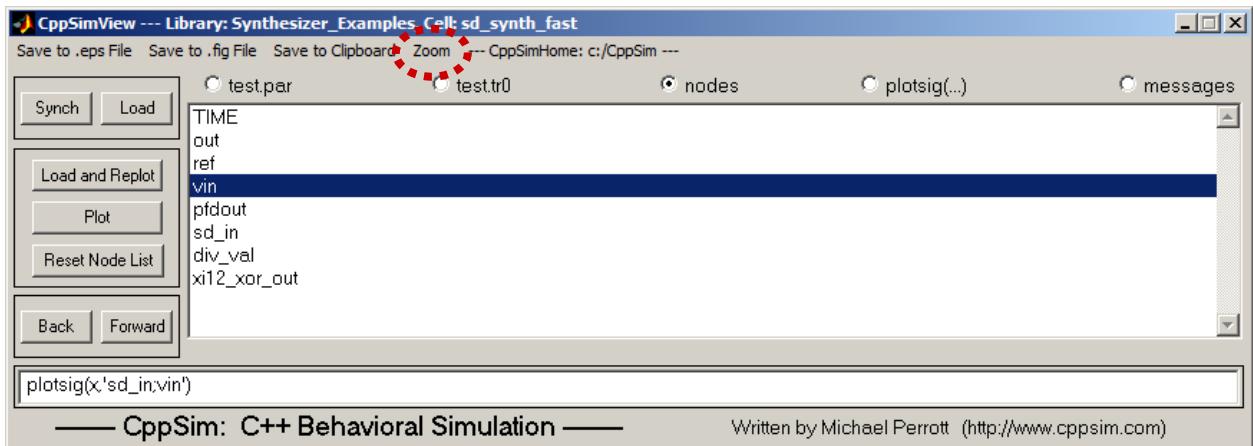


- To view results of the simulation, start **CppSimView** by performing the following action:
  - Windows:** double-click on the **CppSimView** icon on the Windows Desktop.
  - Mac OS X:** double-click on the **CppSimView** app in the Applications folder
  - Linux:** at the Linux prompt, run the command **cppsimeview**

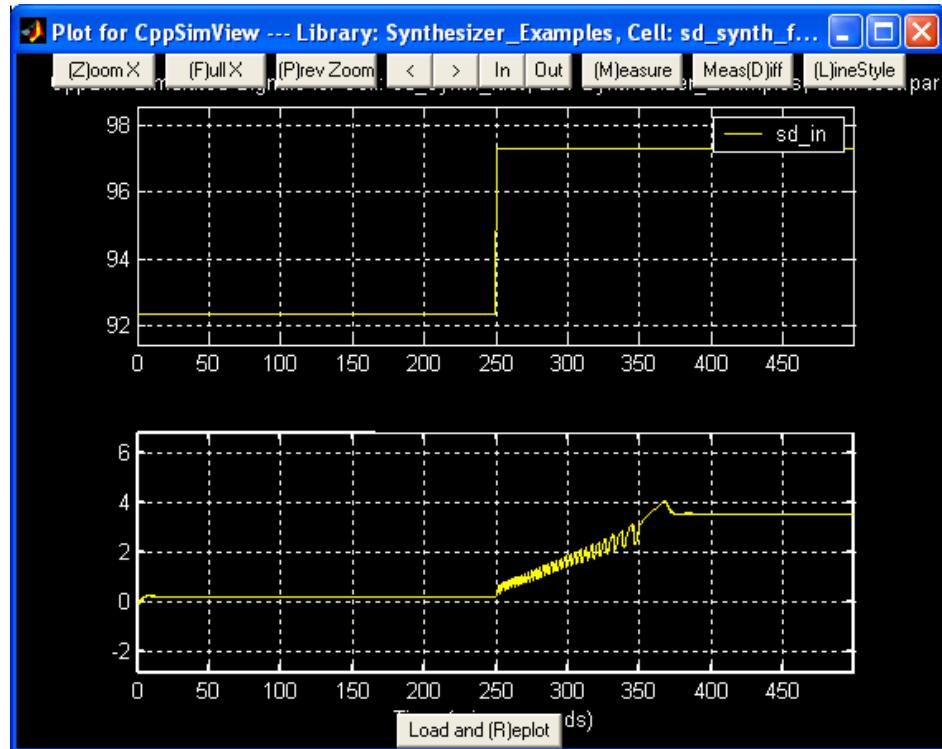
You should see a new window appear as shown below.



- To view simulation results for a given signal within the **sd\_synth\_fast** cell, you need to first choose an appropriate **Output File** and then the **Node** that the signal is associated with. In the window shown above, first click on the **No Output File** radio button, and choose **test.tr0** as the output file. Next click on the **No Nodes** radio button, and then double-click first on node **sd\_in** and then on node **vin**. The resulting CppSimView window should appear as shown below, and the Plot Window should show the corresponding signal waveforms.



- Now click on **Zoom** (circled above in red). The resulting Plot Window should appear as shown below.

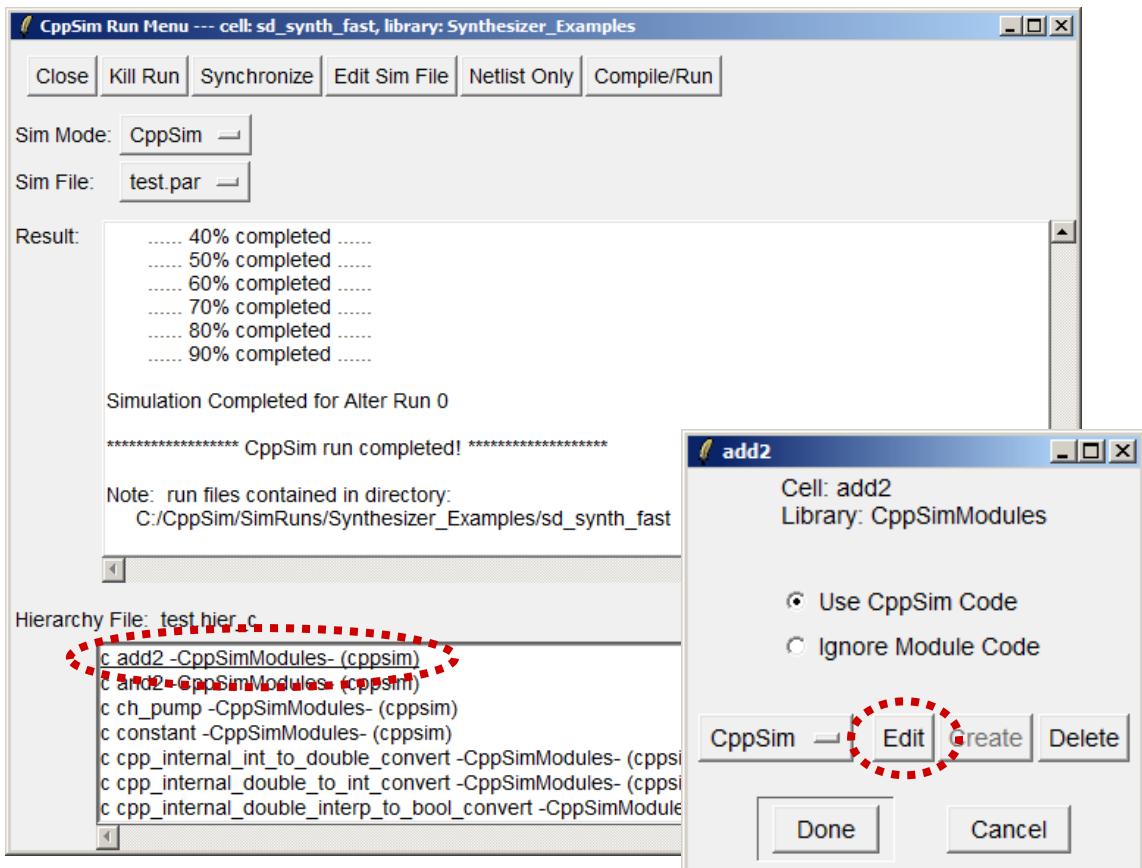


- Consider clicking on different buttons in the Plot Window to zoom into portions of the signals and perform various other operations. One convenient feature is the use of the arrow keys on your keyboard to zoom in, zoom out, and pan left and right. The **down** arrow key zooms in, the **up** arrow key zooms out, and the **left** and **right** arrow keys pan left and right, respectively.

## B. Editing CppSim Module Descriptions

There are several ways to edit CppSim module descriptions of the various blocks used within a given design. Each of them displays a text file in **Emacs** (a freely available text editor) corresponding to the CppSim code of the module, the template of which is described in more detail in the **CppClass Reference Manual** (i.e., **cppsimec.pdf**). Here we will show three different ways of conveniently accessing the module code text file of a given module from the **Sue2** environment.

- Continuing with the example from the previous section, double-click on the **add2** module circled in red within the **Hierarchy File** section as shown below. A pop-up menu will appear, as also shown below, within which you should push the **Edit CppSim code** button (circled in red) to view the CppSim module description code of the **add2** module.



- After performing the above operations, you should see the **Emacs** editor window shown below. Note that the CppSim module code is in template form, with fields such as the module name, parameters, inputs and outputs, etc. The best way to understand the use of these fields is to check out different module code examples. Also, the CppSim Reference Manual, **cppsimec.pdf**, should also be examined.

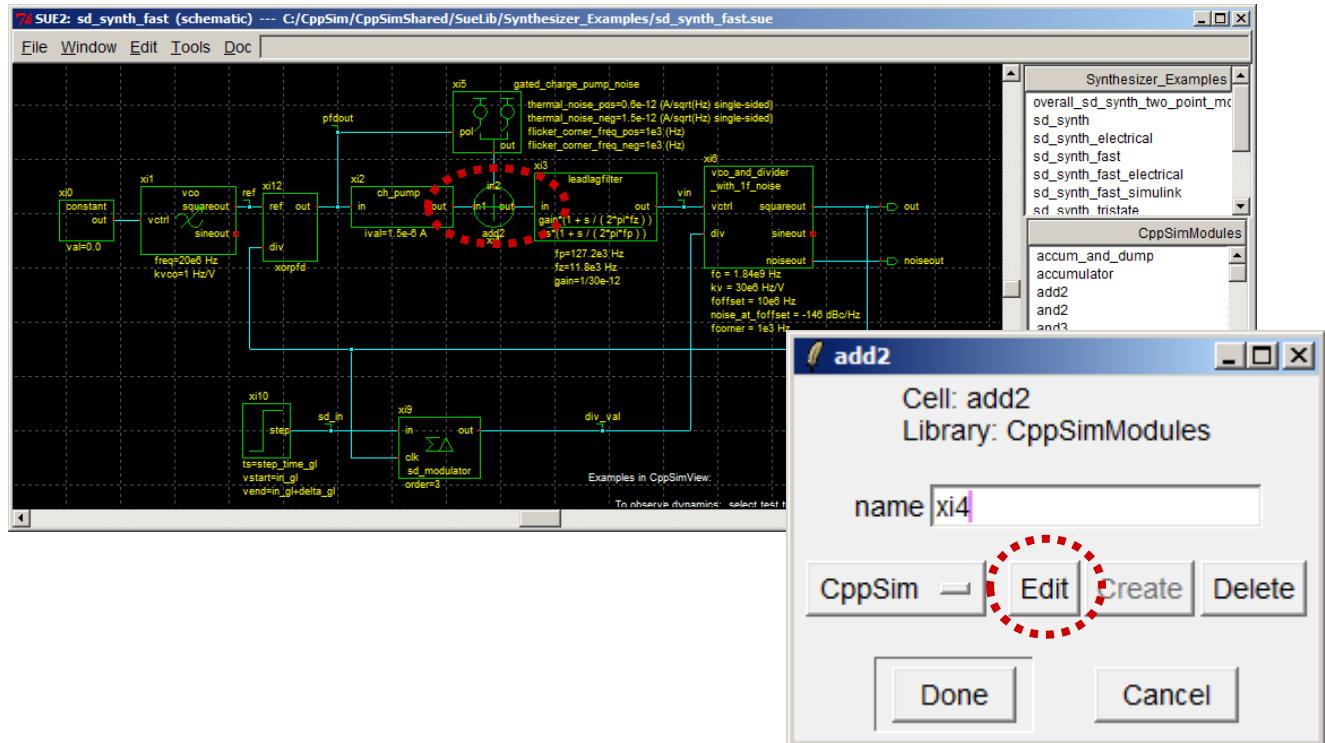
```

C:\CppSim\CppSimShared\CadenceLib\CppSimModules\add2\cp...
File Edit Options Buffers Tools Help
module: add2
parameters:
inputs: double in1, double in2
outputs: double out
static_variables:
classes:
init:
out = in1+in2;
code:
out = in1+in2;

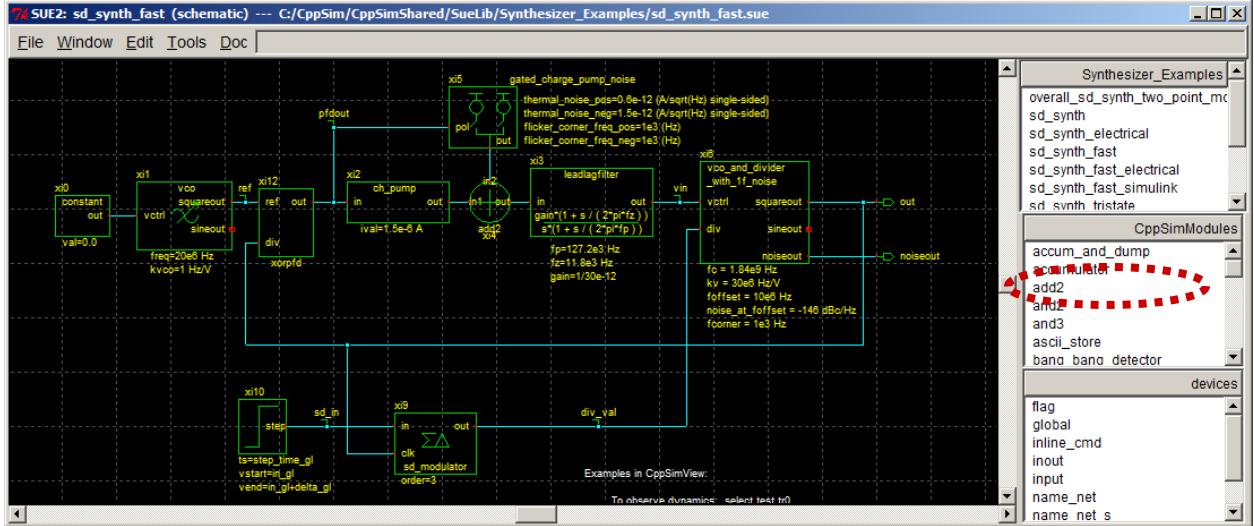
```

--\-- text.txt (Text) --L11--All--\--

- Within the CppSim schematic window, double-click on a given module such as **add2** (circled in red in the figure below). You may then edit the module code by pushing the **Edit CppSim** code button, as circled below.



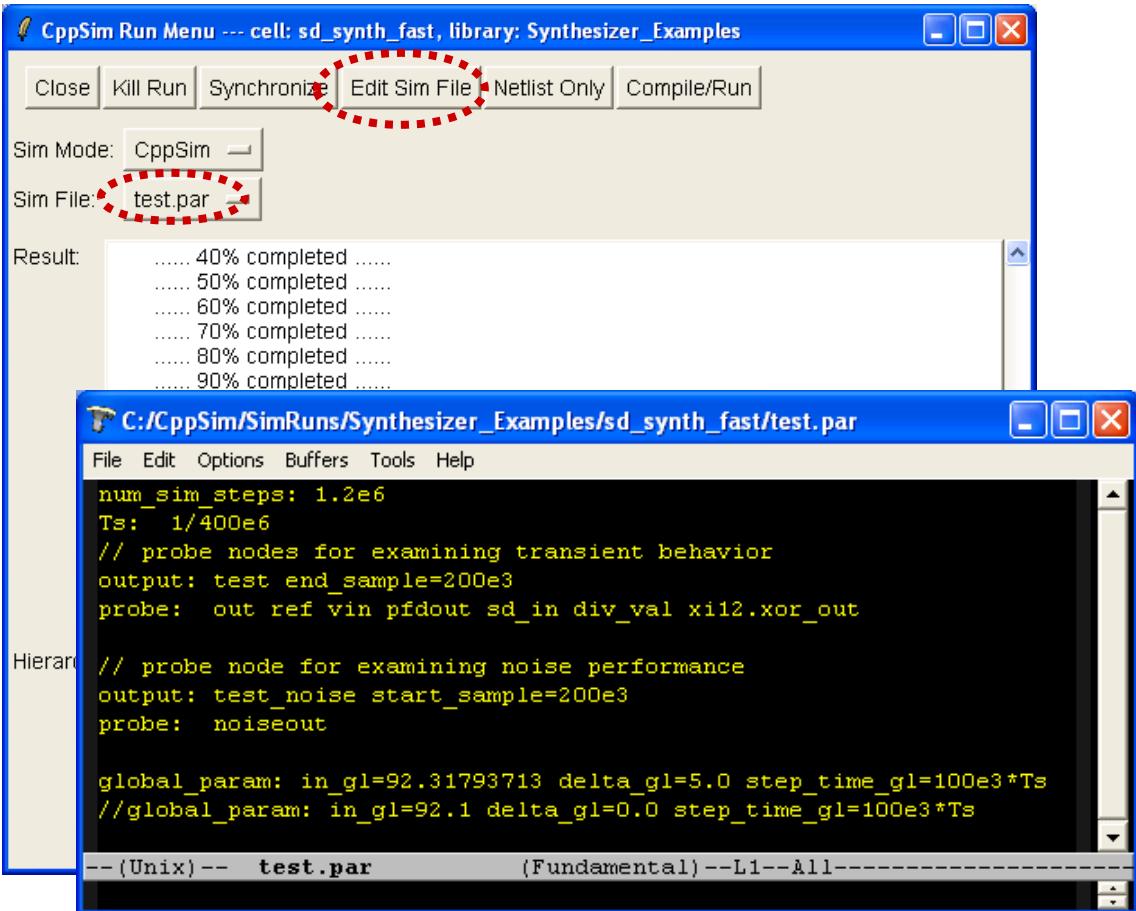
- Finally, within the **icons listbox**, double-click on a given module such as **add2** as shown below. A window will again appear offering an **Edit CppSim** code button for the module.



## C. Editing Simulation Files (i.e., test.par files)

While the CppSim module code dictates the behavior of individual modules, there needs to be simulation specifications specific to a given cell schematic (i.e., interconnection of modules) when performing a CppSim simulation. Such specifications include the time step of the simulator, the number of simulation samples to be calculated, and the value of any top level or global parameters. CppSim allows multiple sets of such specifications, each contained within an individual **Sim File**. Typically, there is only one **Sim File**, which is labeled by default as **test.par**.

- Continuing with the example of the previous section, click on the **Edit Sim File** button within the **CppClassSim Run Menu** window as shown below. An **Emacs** window will then appear that displays the **Sim File** specified in the field as circled below. As shown in the Emacs window below, the **Sim File** is specified in template form to indicate the simulation time step (**Ts**), the number of time steps (**num\_sim\_steps**), etc. For more information on these fields, check out the **CppClassSim Reference Manual (cppsimdoc.pdf)**



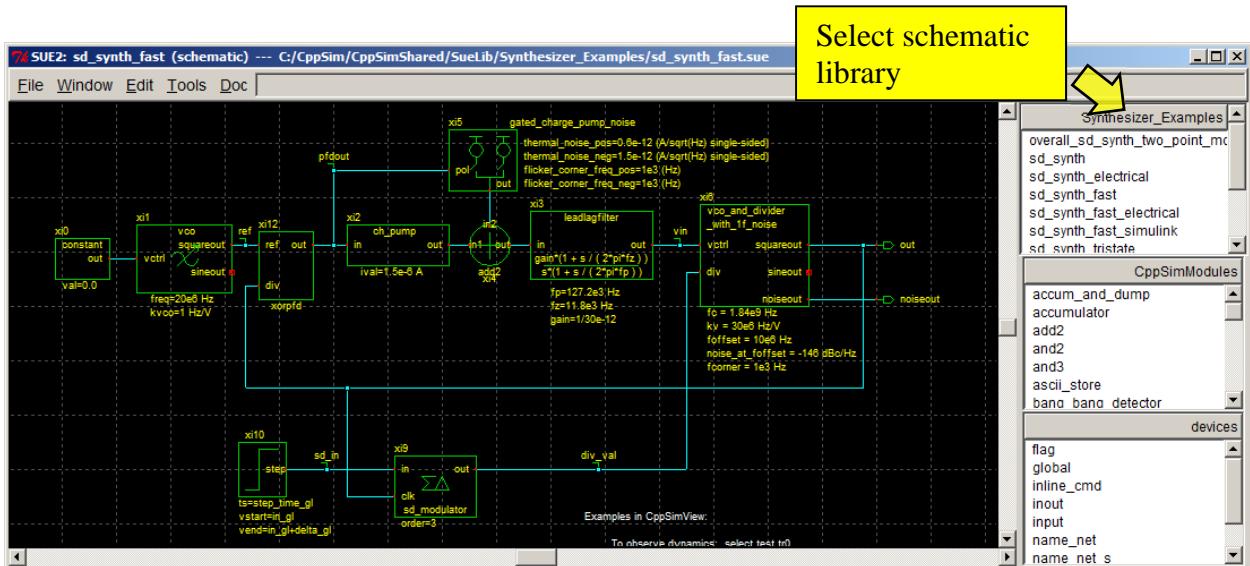
- Note that if no **Sim File** exists, then pressing the **Edit Sim File** button within the **CppSim Run Menu** will create a new sim file (with name **test.par**) that can be edited and saved.

## CppSim versus VppSim for Simulation of Verilog Modules

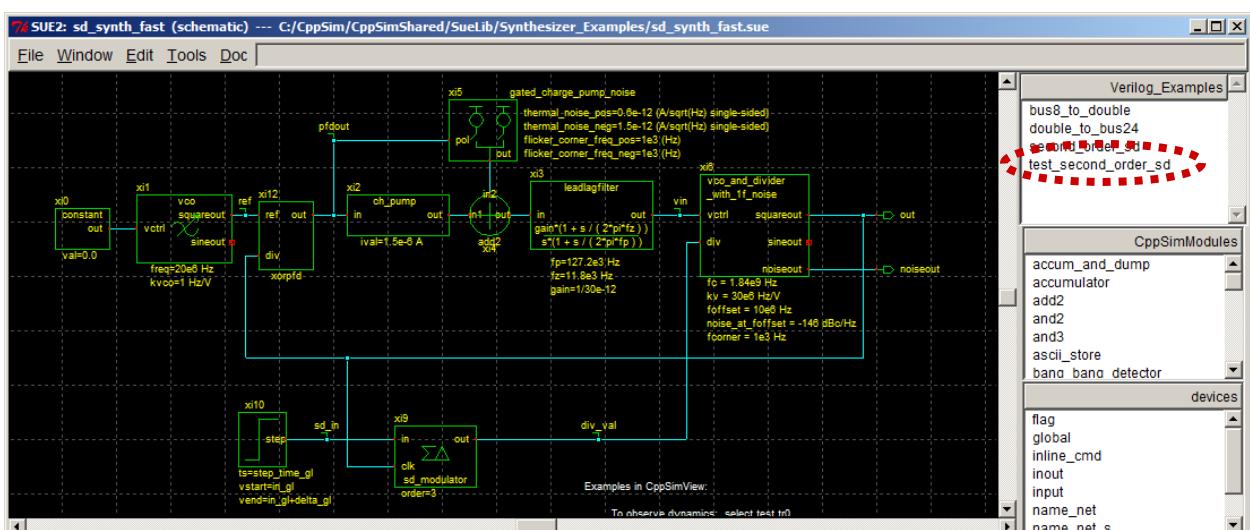
We now discuss inclusion of Verilog modules within CppSim by walking through an example using **Sue2**, **CppSimView**, and **GTKWave**. We then repeat the same example using VppSim, and compare and contrast CppSim versus VppSim for simulation of digital systems. In the exercises to follow, we will assume that the reader has already read the previous section on the basics of running CppSim simulations.

### A. Running a CppSim Simulation with a Verilog Module

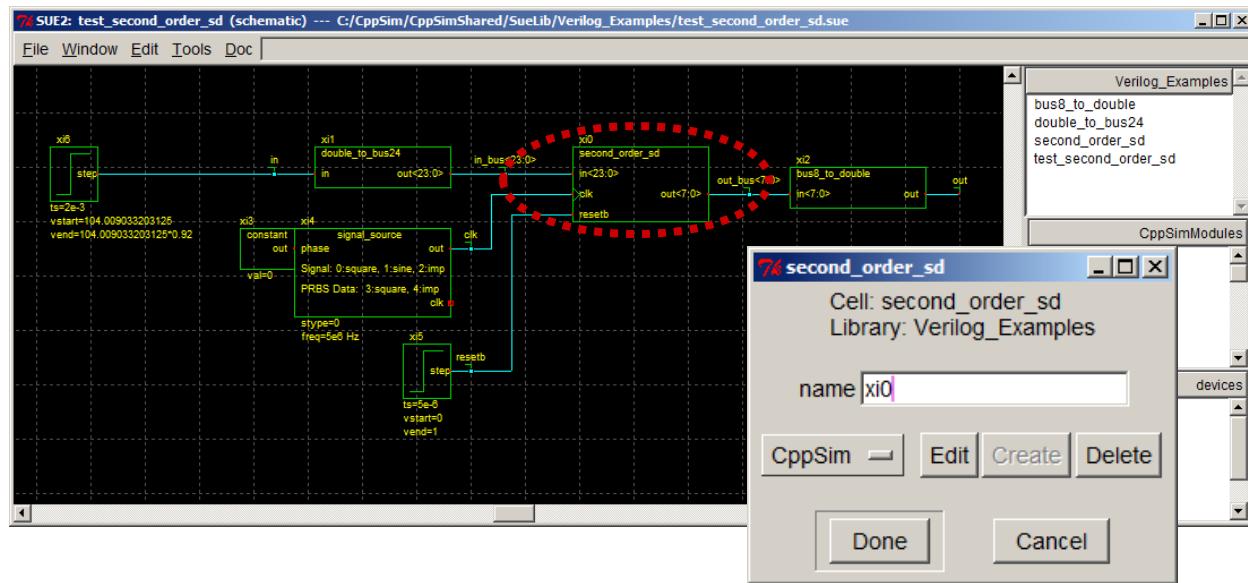
- Within **Sue2**, click on the schematic library button as indicated in the figure below. You should see a list of libraries appear. Choose **Verilog\_Examples** as the library.



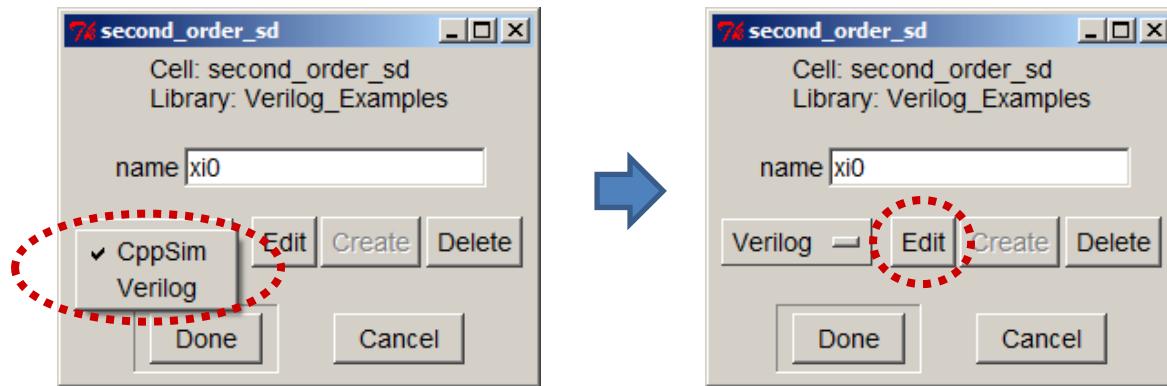
- Within the schematic listbox, which now indicate **Verilog\_Examples** as its library, select the schematic **test\_second\_order\_sd** as indicated below.



- Within the **test\_second\_order\_sd** schematic, double-click on the **second\_order\_sd** module as indicated below. The property menu should appear for the **second\_order\_sd** module as also shown in the figure.



- Click on the **CppSim** button in the property menu of the **second\_order\_sd** module. You should see **Verilog** appear as an option as shown below. Select **Verilog** and then click on **Edit** as indicated below.



- The resulting editor window displays Verilog code for the **second\_order\_sd** module as shown below. One should note that you can also include sub-modules that are required for the main module (**second\_order\_sd**, in this case) to work. Once you have completed examination of the code, close the editor window as well as the property editor window.

```

C:/CppSim/CppSimShared/CadenceLib/Verilog_Examples/second_order_sd/verilog/verilog.v
File Edit Options Buffers Tools Help
// To include other Verilog files, fill in 'verilog_include_dir:' statements
// -> Note that these statements should be preceded with '/' as shown below
// verilog_include_dir:
// verilog_include_dir:

`define IN_BITS 24
`define OUT_BITS 8

module second_order_sd(in, clk, resetb, out);

output [`OUT_BITS-1:0] out;
input [`IN_BITS-1:0] in; //24
input clk, resetb;

wire [`OUT_BITS-1:0] out;
wire [`IN_BITS-1:0] error, fb, residue;

assign error = in + fb;
assign out = error[`IN_BITS-1:`IN_BITS-'OUT_BITS];
assign residue = {'OUT_BITS'h0,error[`IN_BITS-'OUT_BITS-1:0]};
mod_h mod_h_inst(fb, residue, clk, resetb);

endmodule

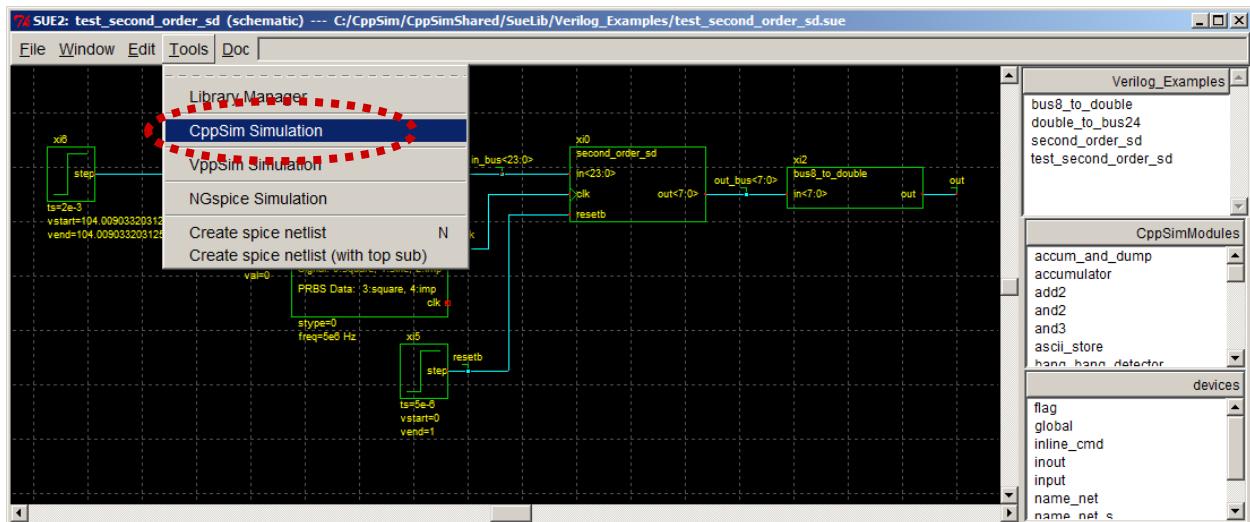
module mod_h(out, in, clk, resetb);

output [`IN_BITS-1:0] out;
input [`IN_BITS-1:0] in;
input clk, resetb;

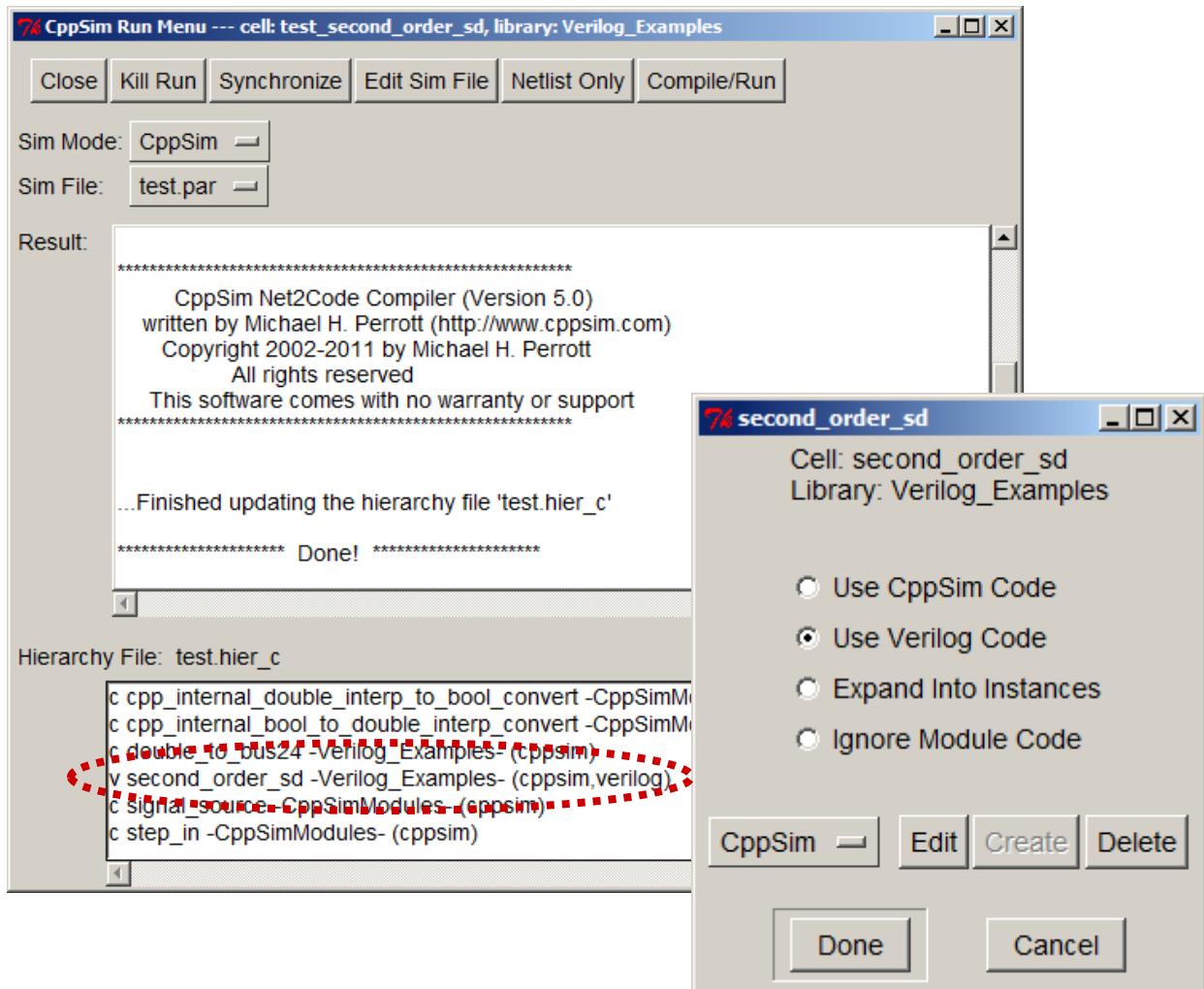
--\-- verilog.v      (Fundamental)--L26--Top--

```

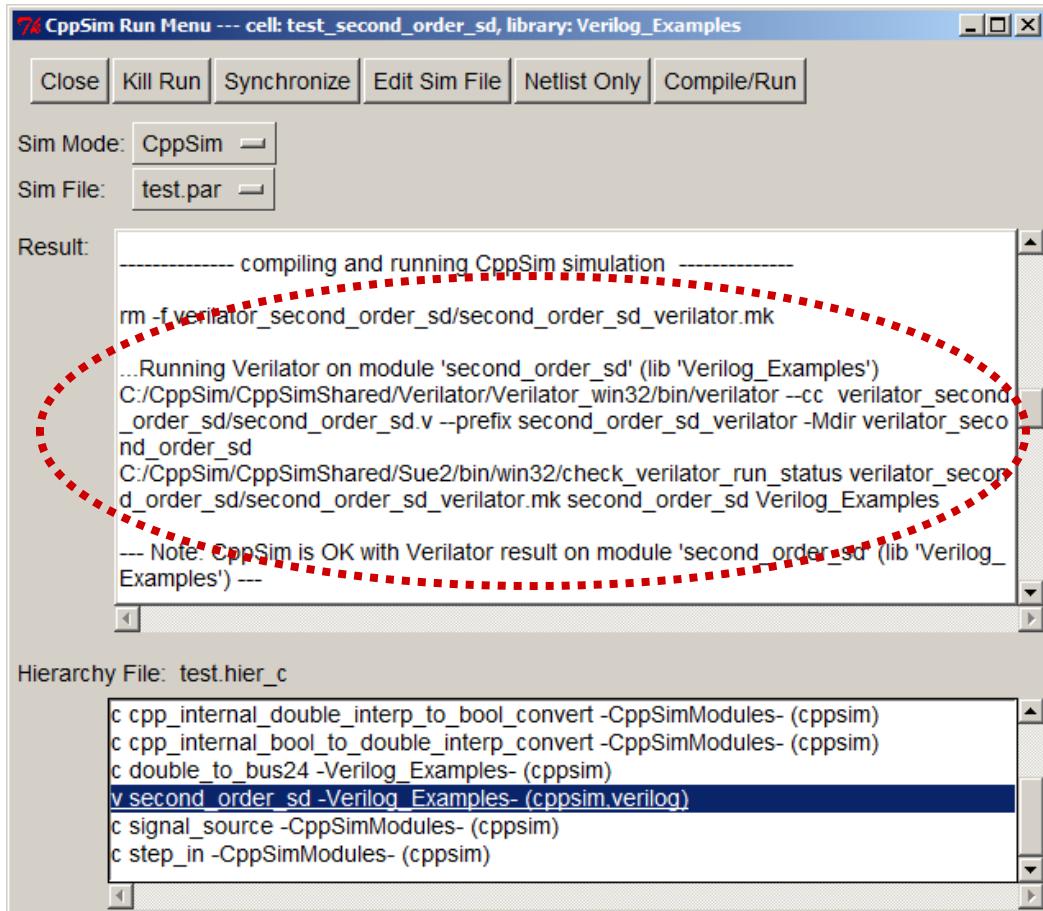
- Now click on the **CppSim Simulation** item under the **Tools** menu item as shown below.



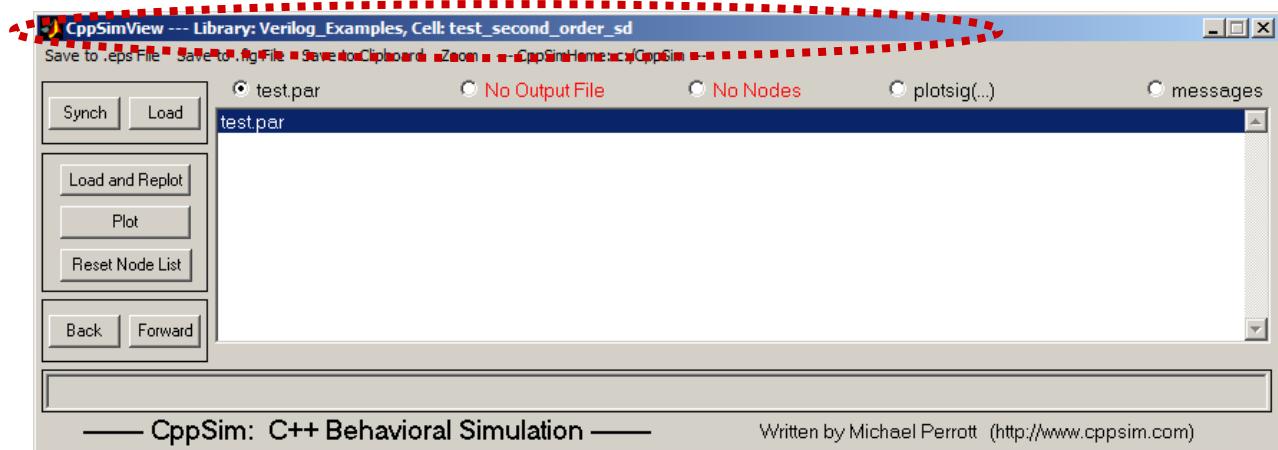
- Within the **Hierarchy File** section of the **CppSim Run Menu** that appears, scroll down to the **second\_order\_sd** entry and double-click on it as indicated below. A new window will appear, as also shown below, which allows selection of the module code for the simulation. Keep the code type as **Verilog** as shown below, and click on **Done** to terminate the code selection window.



- Click on **Compile/Run** in the **CppSim Run Menu** in order to run the CppSim simulation with the included Verilog module. Upon completion of the simulation, scroll up the Result window and examine the simulation messages. You should notice that **Verilator** was run on the **second\_order\_sd** module, which turned its Verilog code into an equivalent C++ class. CppSim then seamlessly incorporated this C++ class into the simulation. A key restriction of Verilator is that it does not support behavioral Verilog code. Rather, the Verilog code must be synthesizable. As we will see, this is one of the key differences between CppSim and VppSim since VppSim allows both behavioral and synthesizable Verilog code to be utilized.

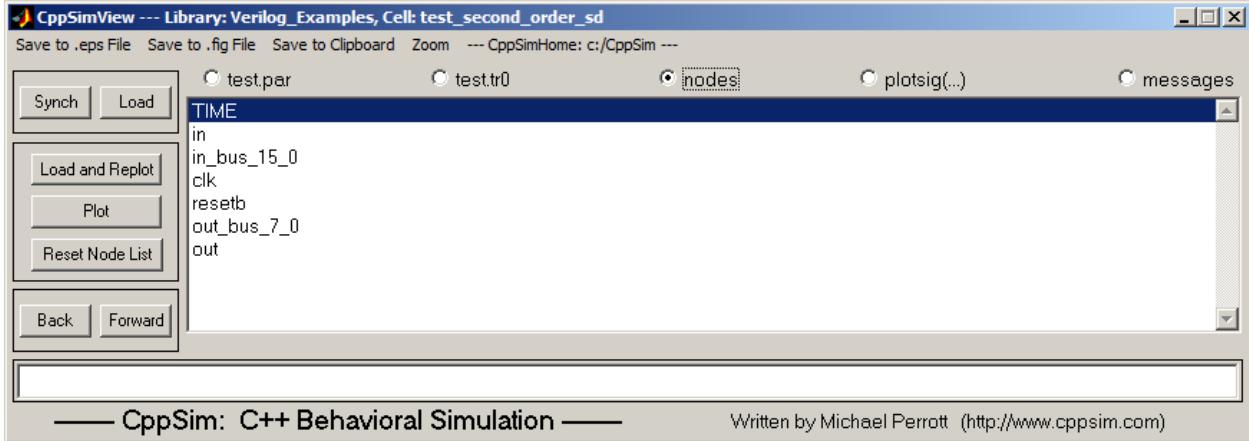


- If you are already running CppSimView, push the Synch button. Otherwise, click on the CppSimView icon to open it. It should appear as shown below with **test\_second\_order\_sd** in its title banner.

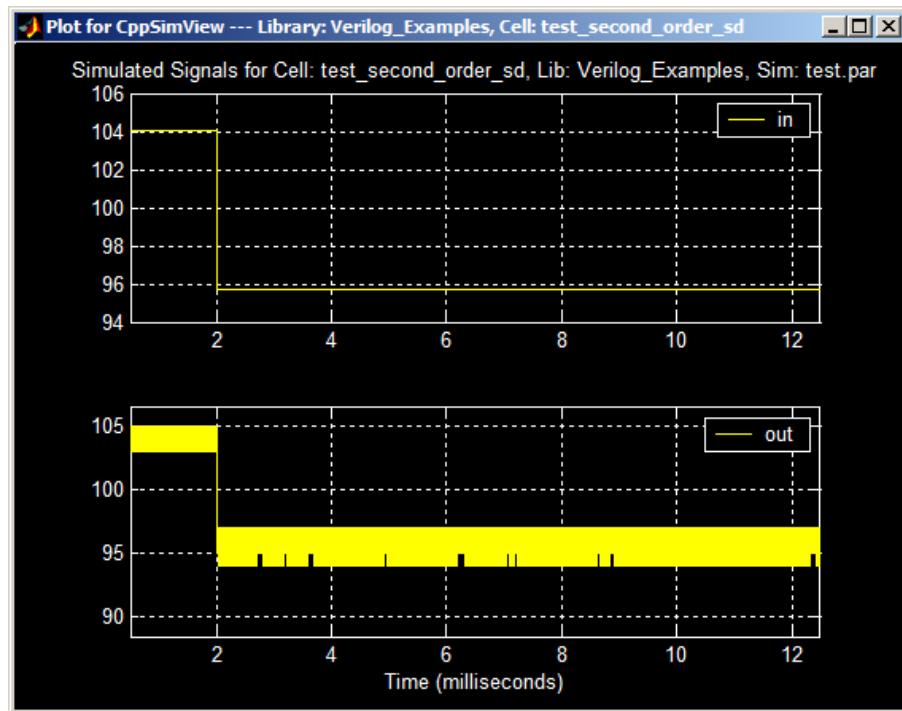


- Click on the **Output File** and **Nodes** radio buttons to obtain a list of nodes as shown below. One should notice that the bussed signals in the schematic are represented by their equivalent integer values for display by CppSimView. As an example, **in\_bus<15:0>** is represented by the integer value **in\_bus\_15\_0** (note that the **probe:** statement in the **test.par** file indicated a

subset of the 24 bits to be chosen), and **out\_bus<7:0>** is represented by the integer value **out\_bus\_7\_0**.



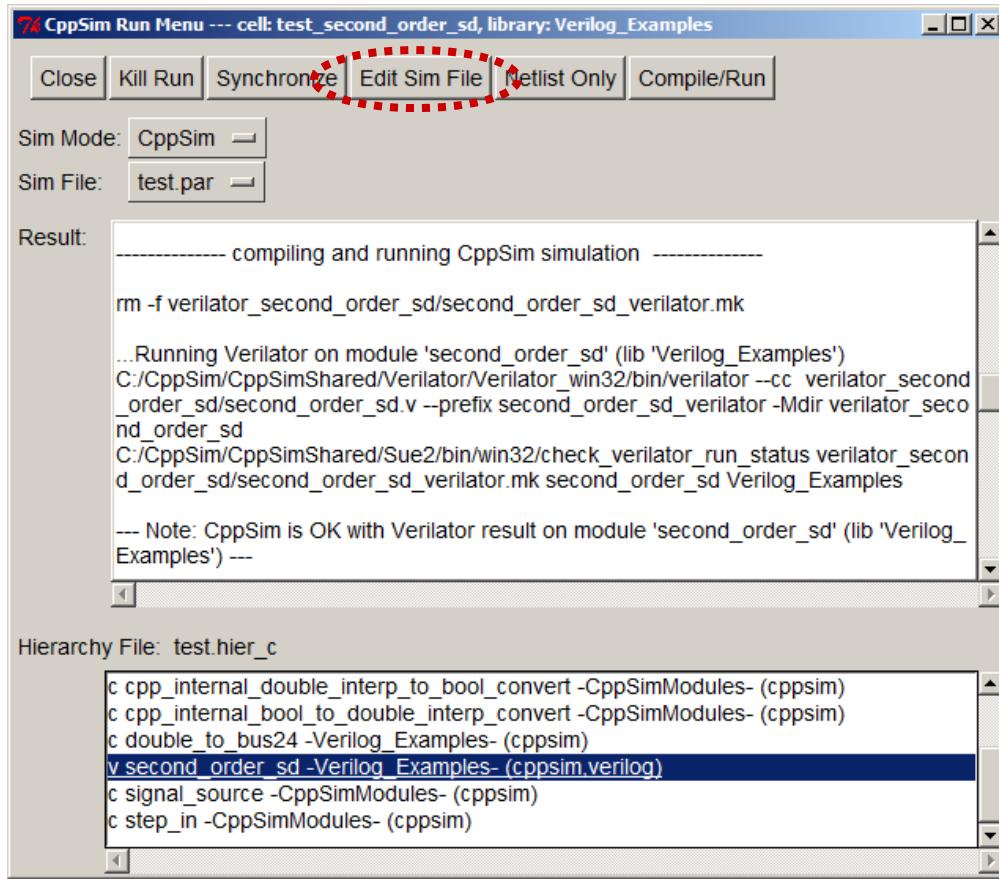
- Double-click on nodes **in** and **out**. The resulting plot window should appear as shown below.



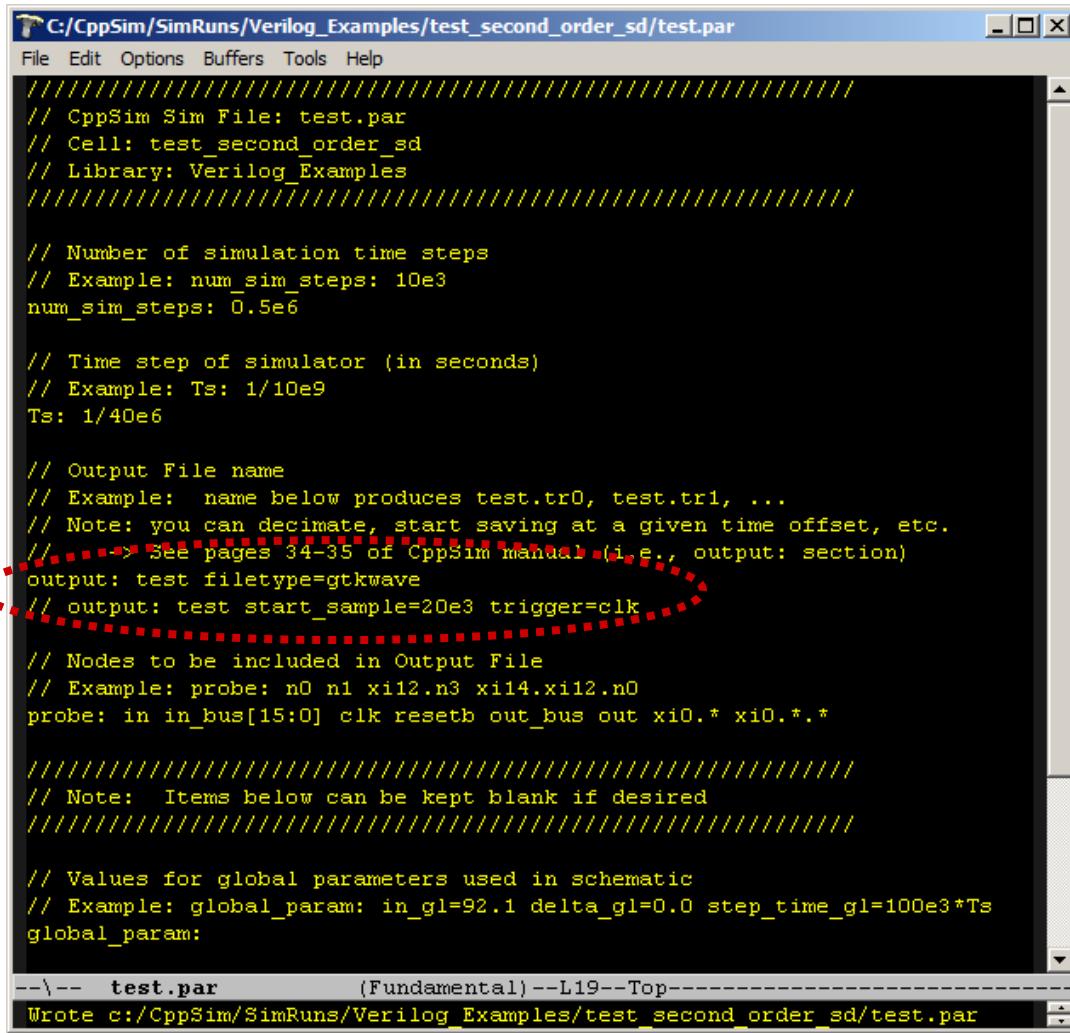
## B. Utilizing GTKWave to view CppSim Simulations

In contrast to CppSimView, GTKWave offers a much more flexible means of viewing digital signals. Here we illustrate its use with CppSim when bussed signals are included in the system simulation.

- Assuming you have completed the previous section, click on the **Edit Sim File** button within the **CppClass Run Menu** as shown below.



- Within the editor window that appears, make the following alterations such that the file appears as below
  - Comment out the line **output: test start\_sample=20e3....**
  - Uncomment the line **output: test filetype=gtkwave**
  - Save the file



```

C:\CppSim\SimRuns\Verilog_Examples\test_second_order_sd\test.par
File Edit Options Buffers Tools Help
///////////
// CppSim Sim File: test.par
// Cell: test_second_order_sd
// Library: Verilog_Examples
///////////

// Number of simulation time steps
// Example: num_sim_steps: 10e3
num_sim_steps: 0.5e6

// Time step of simulator (in seconds)
// Example: Ts: 1/10e9
Ts: 1/40e6

// 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 filetype=gtkwave
// output: test start_sample=20e3 trigger=clk

// Nodes to be included in Output File
// Example: probe: n0 n1 xi12.n3 xi14.xi12.n0
probe: in in_bus[15:0] clk resetb out_bus out xi0.* xi0.*.*

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

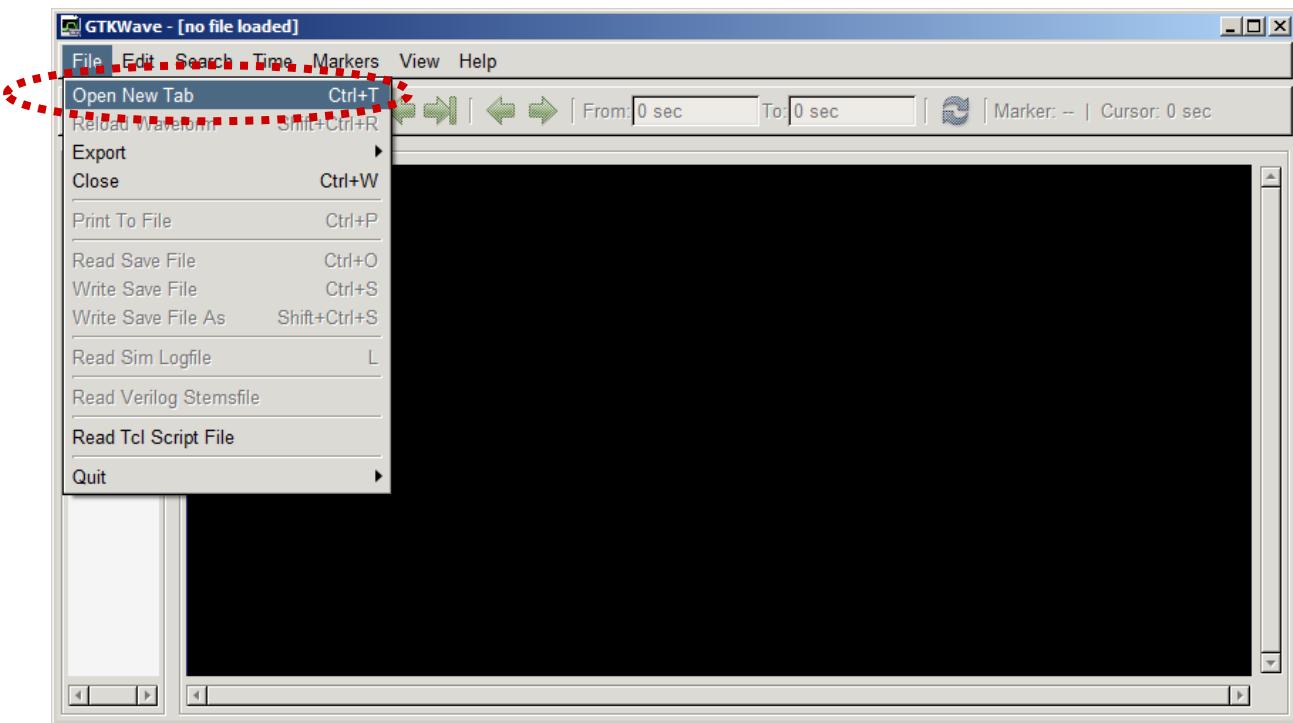
// Values for global parameters used in schematic
// Example: global_param: in_g1=92.1 delta_g1=0.0 step_time_g1=100e3*Ts
global_param:

--\-- test.par (Fundamental)--L19--Top--
Wrote c:/CppSim/SimRuns/Verilog_Examples/test_second_order_sd/test.par

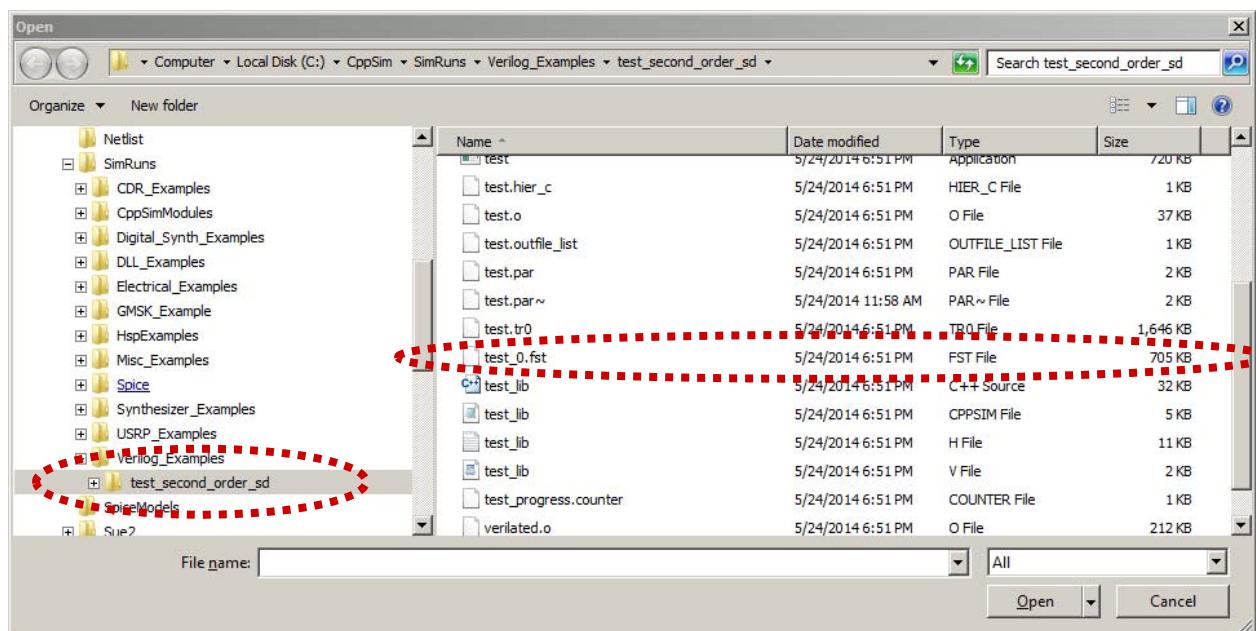
```

- Click on the **Compile/Run** button in the **CppSim Run Menu** to run the CppSim Simulation. You may notice that the simulation runs slower, which is due to the fact that compression algorithm used to store data for GTKWave is computationally intensive.
- To view results of the simulation, start GTKWave by performing the following action:
  - **Windows:** double-click on the **GTKWave** icon on the Windows Desktop.
  - **Mac OS X:** double-click on the **gtkwave** app in the Applications folder
  - **Linux:** at the Linux prompt, run the command **gtkwave**

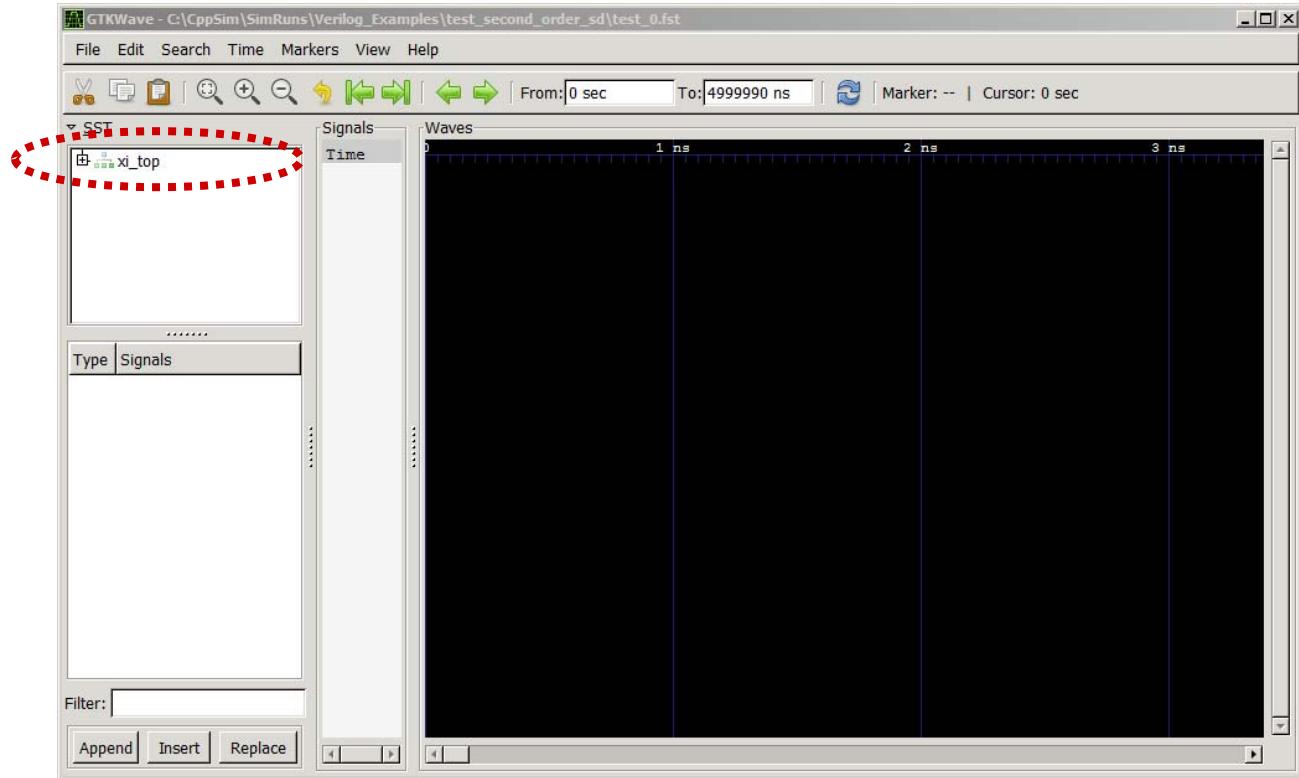
In the new window that appears, click on the **Open New Tab** entry under the **File** menu as shown below.



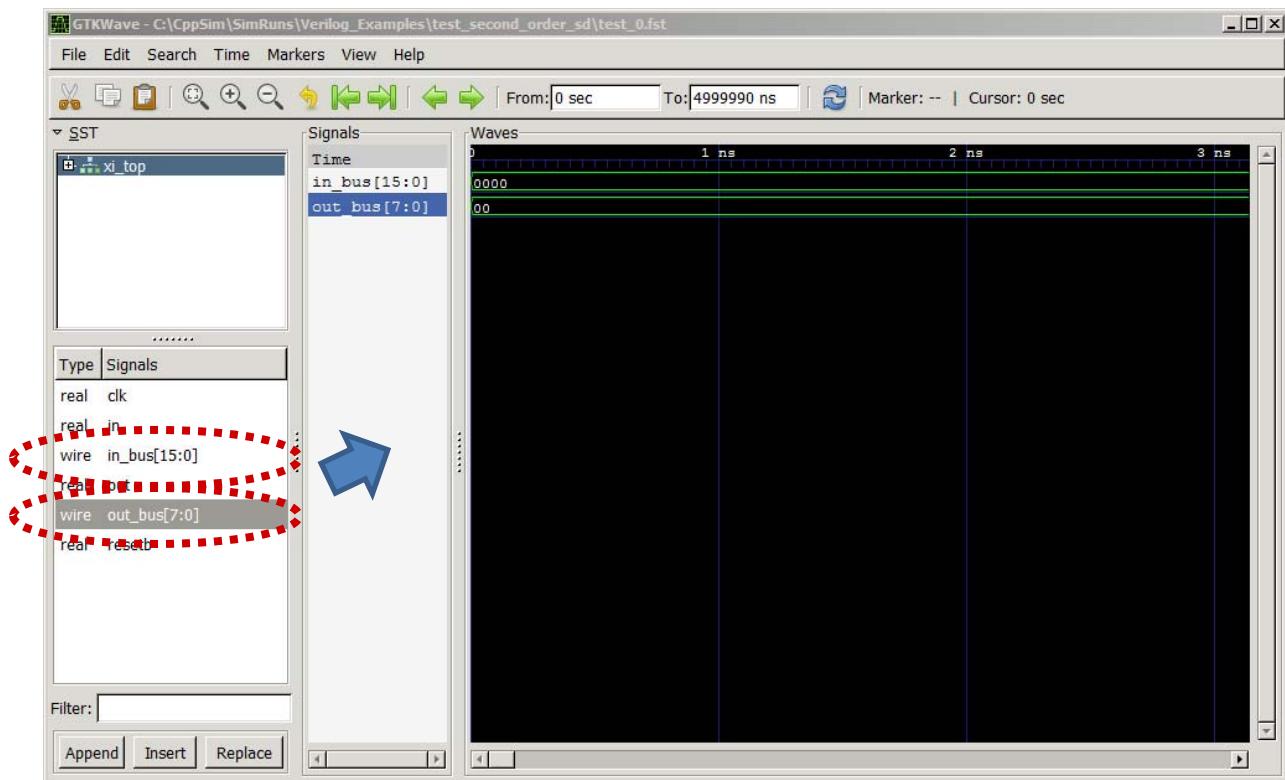
- In the **Open** file window that appears, go to directory **c:\CppSim\SimRuns\Verilog\_Examples\test\_second\_order\_sd**, and then select file **test\_0.fst** as indicated below. Note that, in general, all files produced by CppSim or VppSim for GTKWave will have the suffix **.fst**.



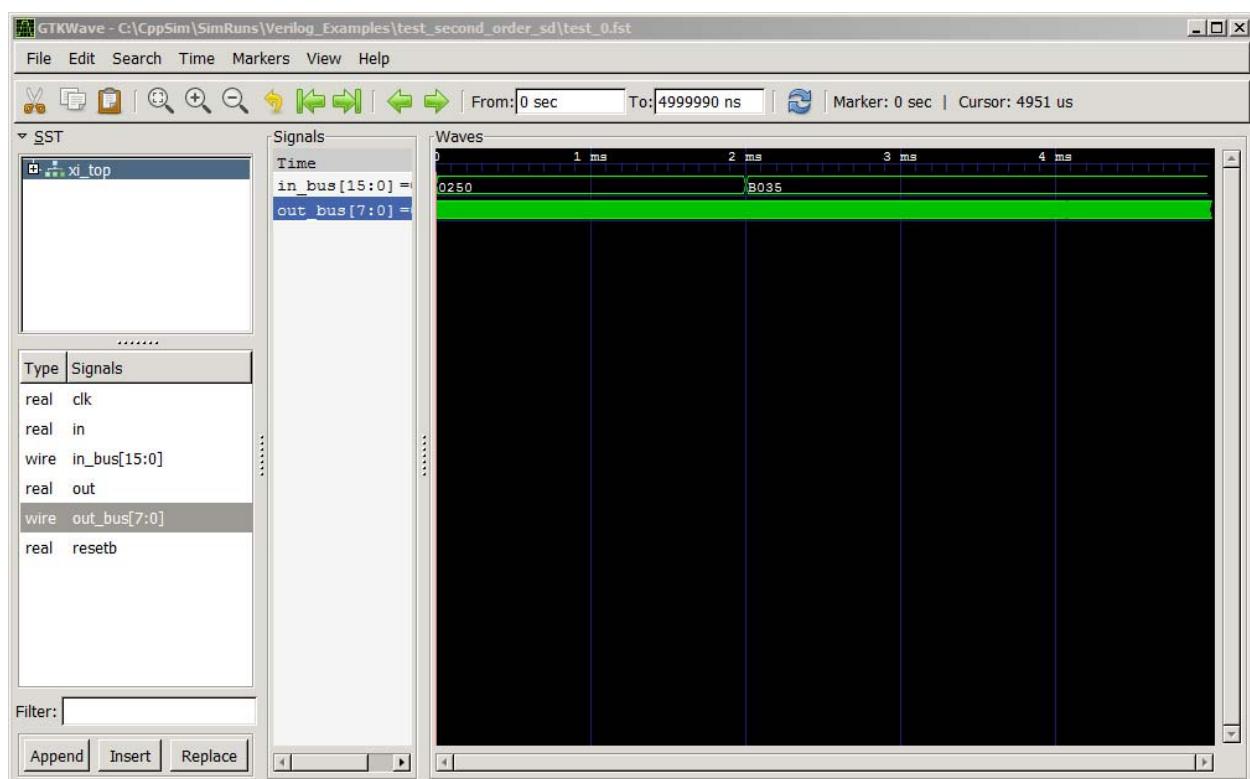
- Upon opening the test\_0.fst file, the top level of the module will appear in the main GTKWave window as shown below. Clicking on the + sign to the left of **xi\_top** will expand into the list of instances for which **probe:** data has been specified in the simulation file (i.e., test.par file). Clicking on **xi\_top** itself will show the signals in the top level which have been specified by the **probe:** statement.



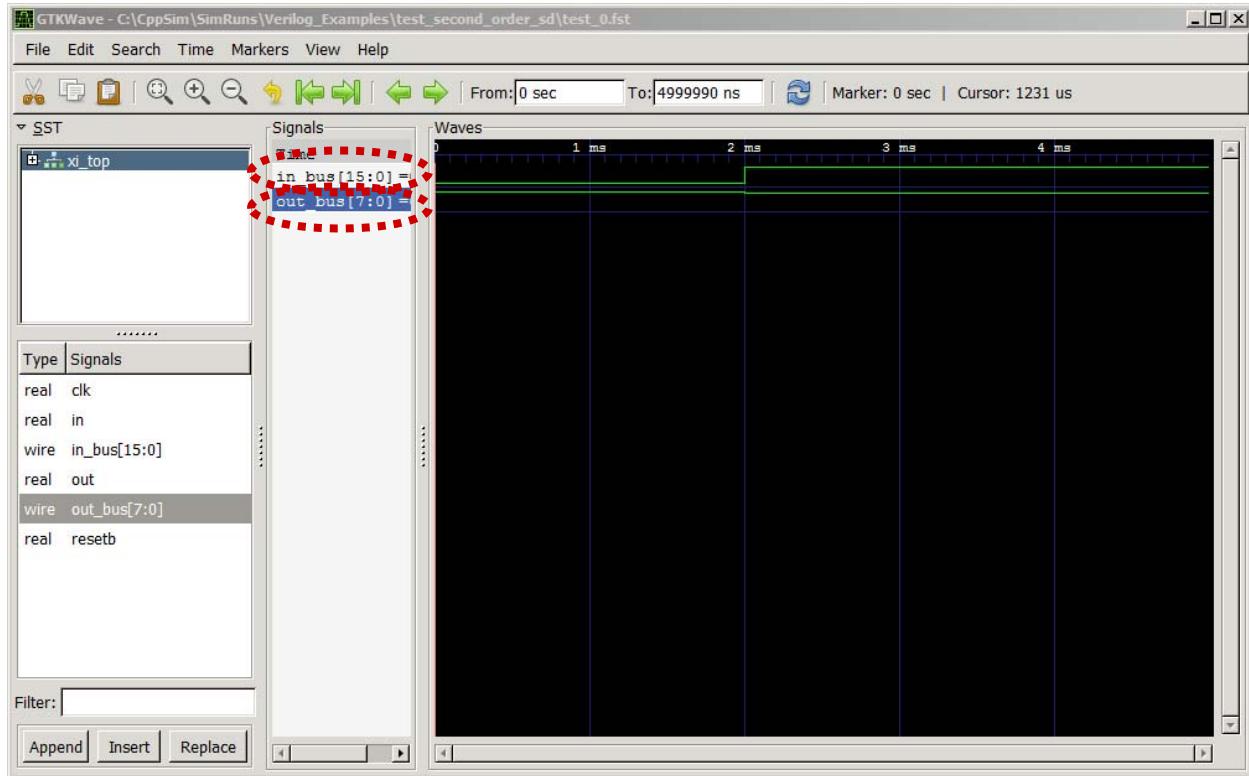
- After exploring the above options for viewing signals, click on **xi\_top** to see the top level signals as shown below. Drag **in\_bus[15:0]** and **out\_bus[7:0]** from the **Signals** box into the **Waves** section in order to see their waveforms as shown below.



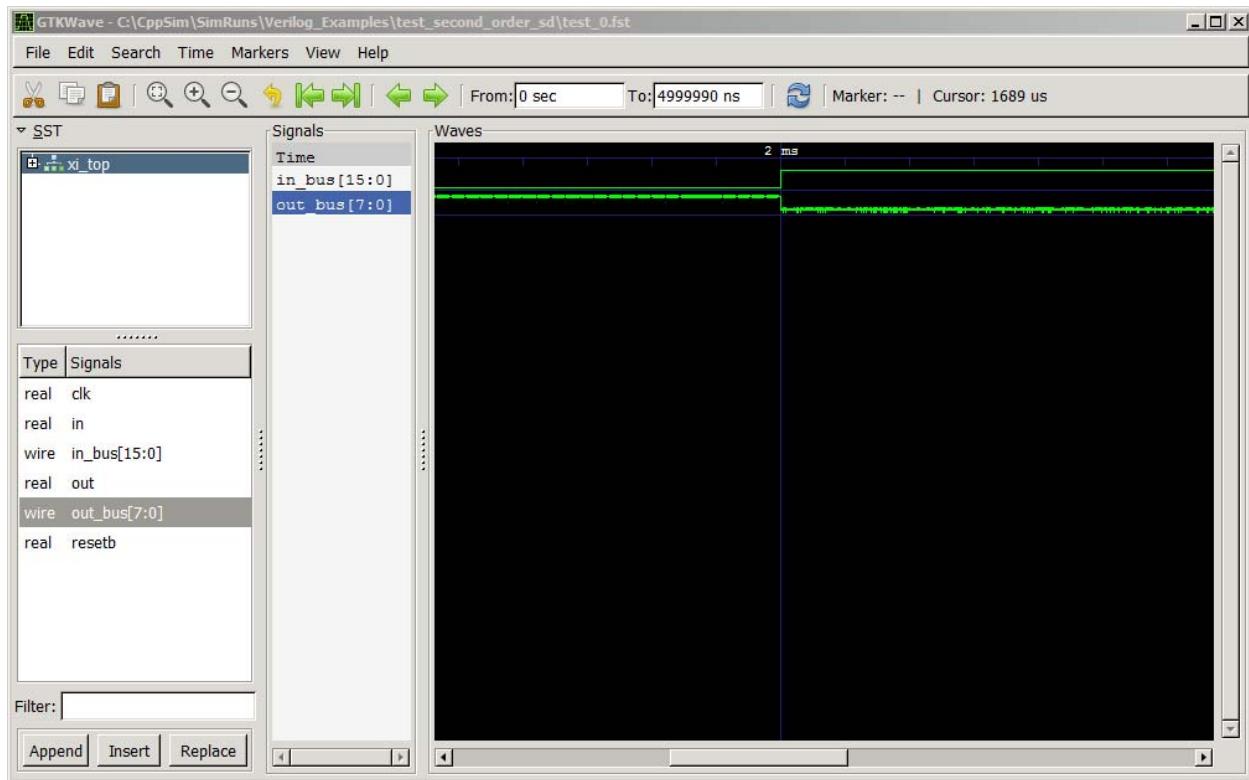
- While in the GTKWave window, hit the **f** key to increase the time span to display the entire simulation time frame.



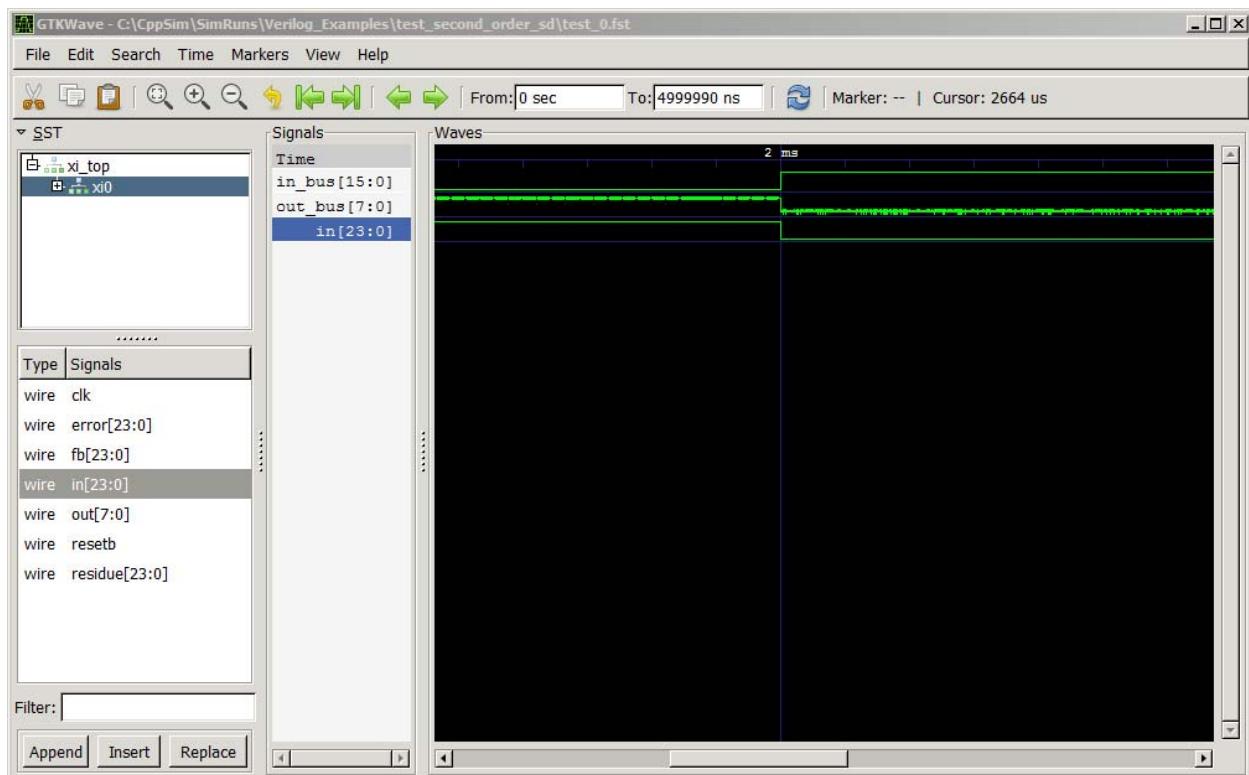
- Left click on the **in\_bus[15:0]** in the **Signals** box, as shown below, and push the **a** key to change its display to analog mode. Do the same to the **out\_bus[7:0]** signal to get the plots shown below. Note that right clicking on these signals in the Signals box allows various representations (i.e., decimal, hex, binary, etc.) to be displayed for the waveform.



- Within the **Waves** part of the window, right-click with the mouse and then slide the mouse pointer across a region to zoom into. Upon releasing the mouse button, the waveform will be re-plotted according to the selected zoom region. As shown in the figure below, it appears that a rising step at the input leads to a falling step at the output. However, it is important to recall that the actual input has 24 rather than 16 bits.

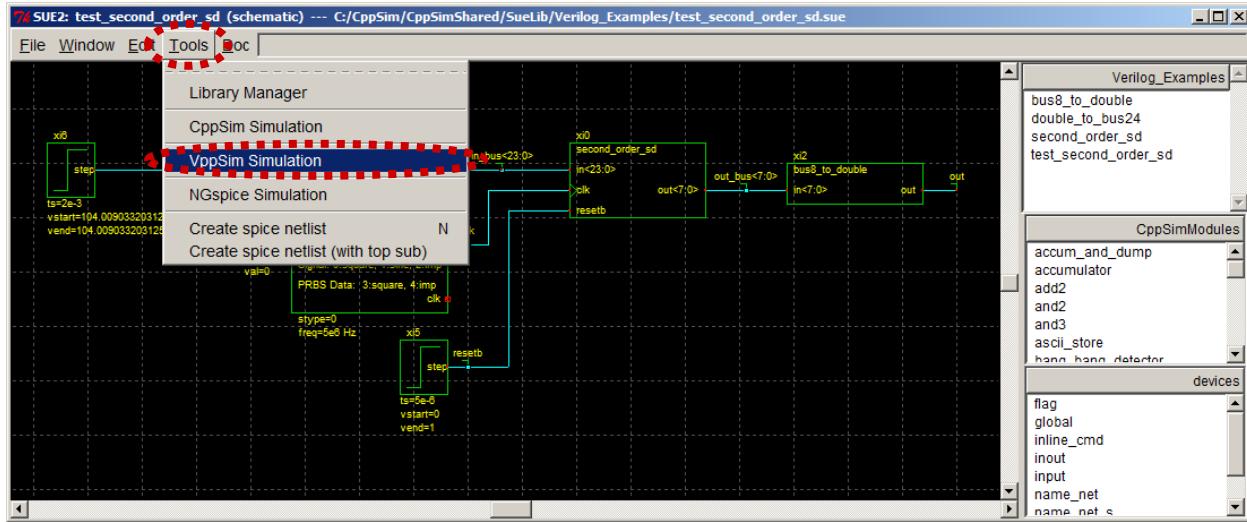


- A more accurate picture is seen by looking at the full 24 bits, which is available within instance **xi0**. As shown below, we see that the full input actually has a falling edge which corresponds to the falling edge seen at the output.

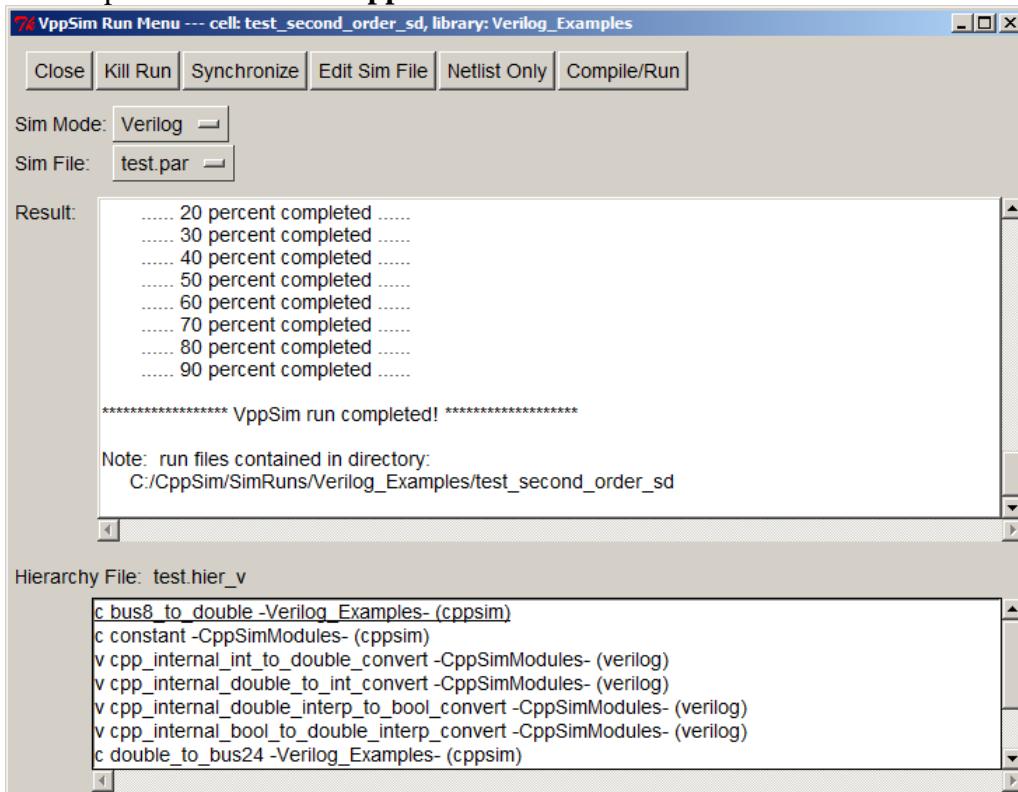


## C. Running a VppSim Simulation with a Verilog Module

- Returning to Sue2, click on **VppSim Simulation** within the **Tools** menu as shown below. A **VppSim Run Menu** window should appear.

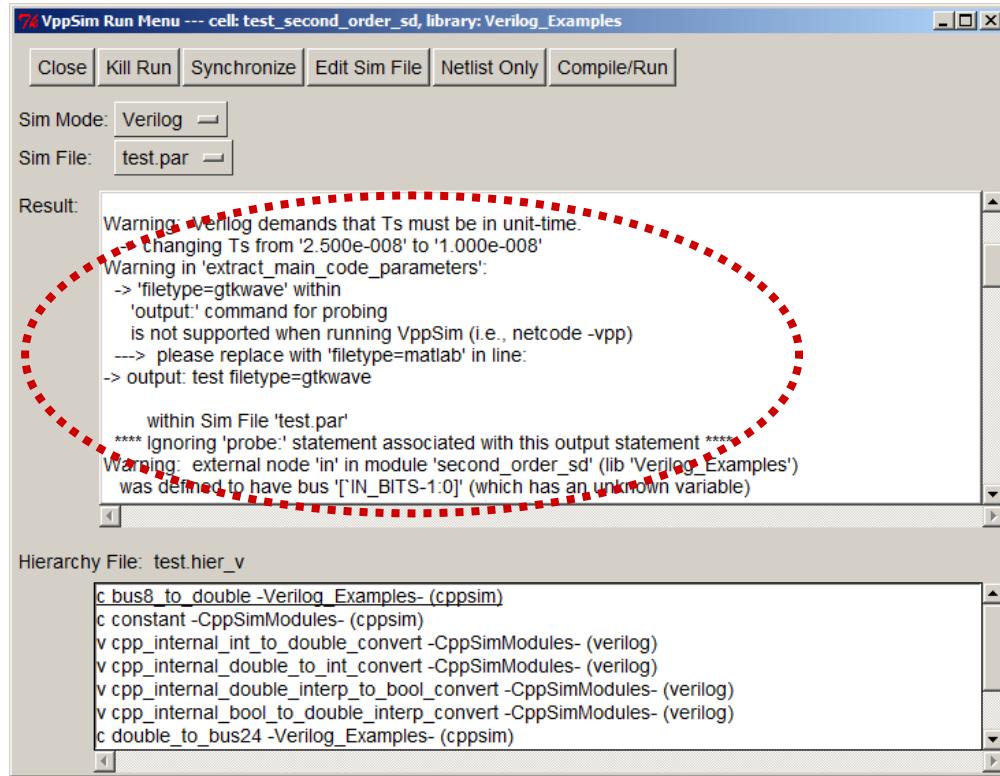


- Click on the **Compile/Run** button within the **VppSim Run Menu** window. The simulation should complete such that the **VppSim Run Menu** looks similar to below.

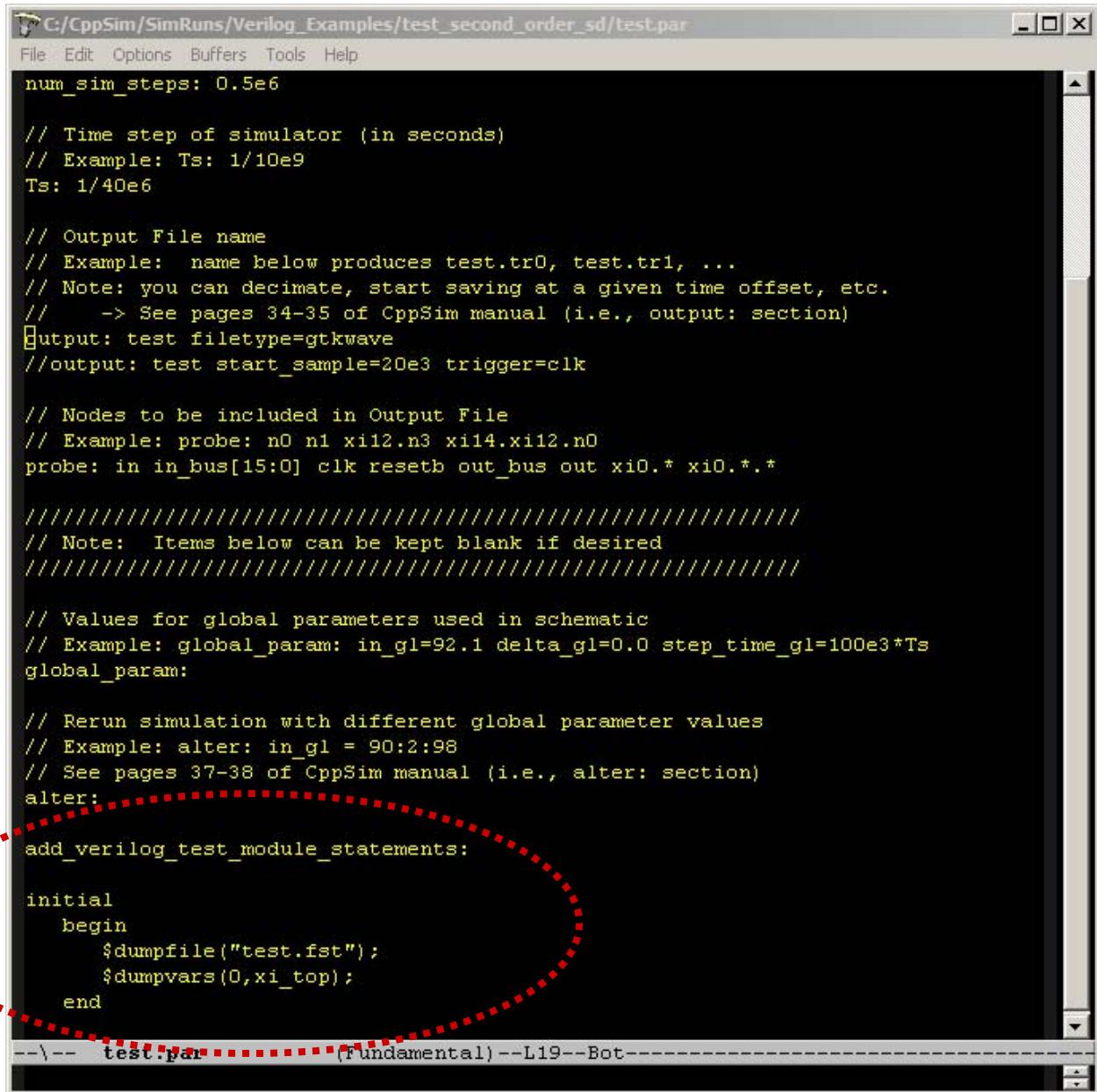


- Scroll up the **Result** section in the **VppSim Run Menu** to see the various messages produced during the simulation. One key message is shown below, namely that **filetype=gtkwave** is not

supported for **output:** statements when using VppSim. Note that **filetype=matlab**, which is supported, is the default setting for **output:** commands. We will return to this issue soon.



- Since **filetype=gtkwave** is not supported for VppSim, a different method must be used to save signals to be viewed by GTKWave. Fortunately, this is straightforward. To see how this is done, click on **Edit Sim File** in the **Run VppSim Menu** window shown above. Scroll to the bottom of the editor window that appears and you will see **\$dumpfile** and **\$dumpvars** commands within the **add\_verilog\_test\_module\_statements:** command as shown below. The **add\_verilog\_test\_module\_statements:** command adds statements to the test module portion of a Verilog testbench that is autogenerated by VppSim. As such, we simply use the standard Verilog testbench commands to save signals for viewing by GTKWave.



```

C:/CppSim/SimRuns/Verilog_Examples/test_second_order_sd/test.par
File Edit Options Buffers Tools Help
num_sim_steps: 0.5e6

// Time step of simulator (in seconds)
// Example: Ts: 1/10e9
Ts: 1/40e6

// 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 filetype=gtkwave
//output: test start_sample=20e3 trigger=clk

// Nodes to be included in Output File
// Example: probe: n0 n1 xi12.n3 xi14.xi12.n0
probe: in in_bus[15:0] clk resetb out_bus out xi0.* xi0.*.*

///////////////////////////////
// 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:

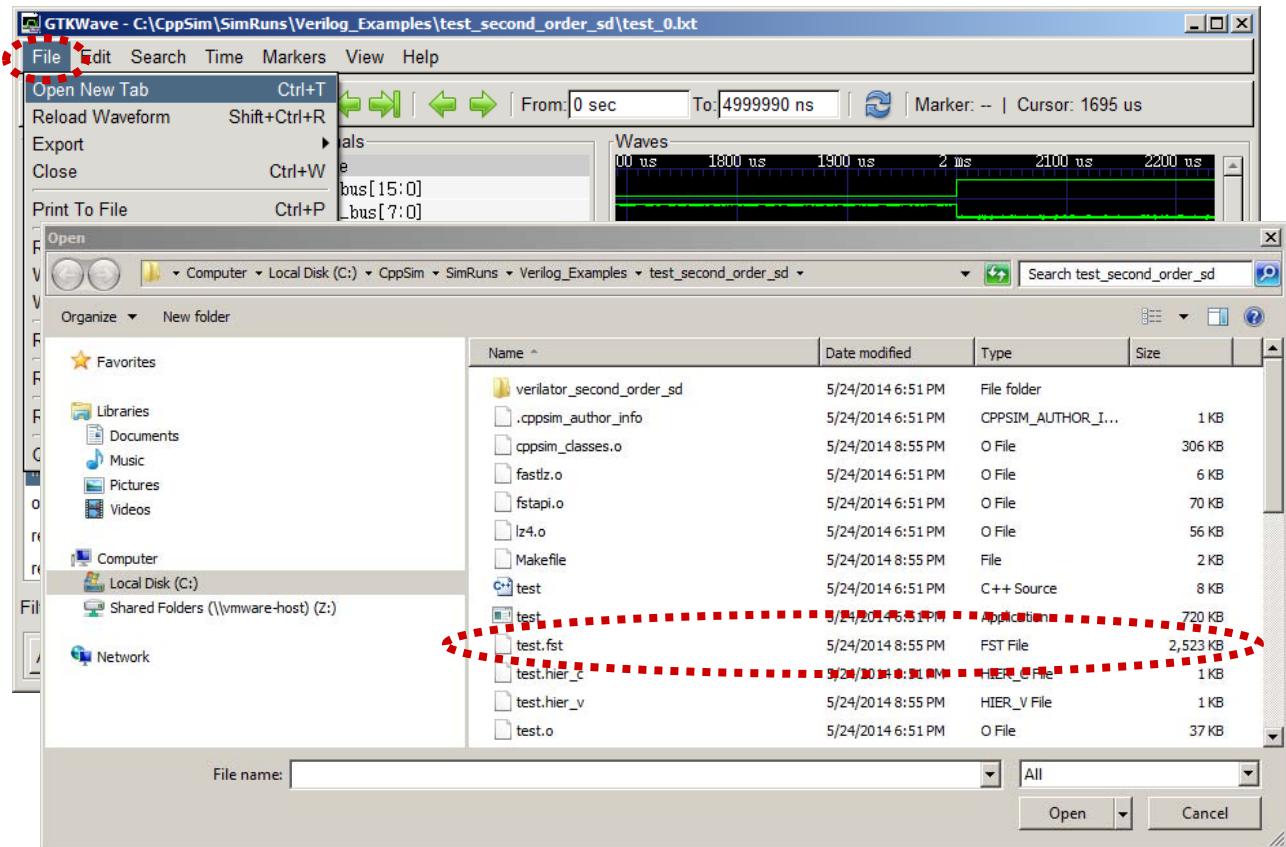
add_verilog_test_module_statements:

initial
begin
    $dumpfile("test.fst");
    $dumpvars(0,xi_top);
end

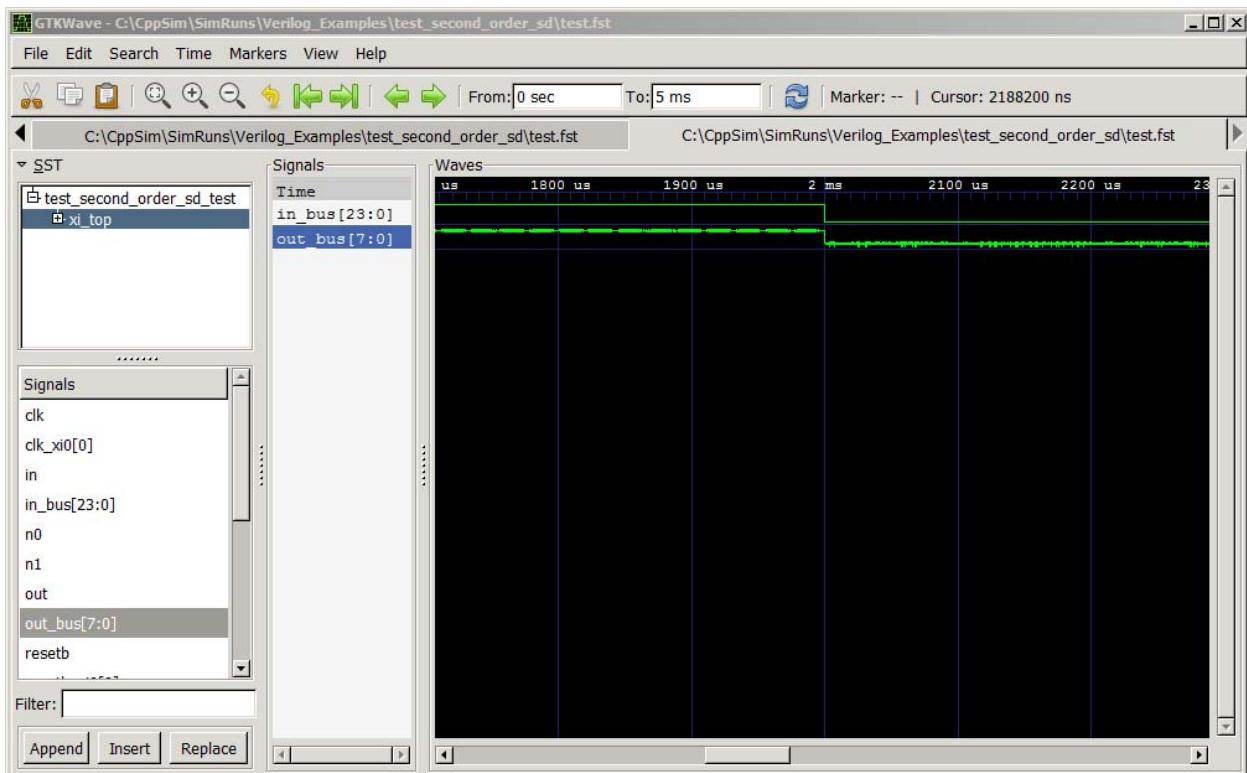
--\-- test.par (Fundamental) --L19--Bot--

```

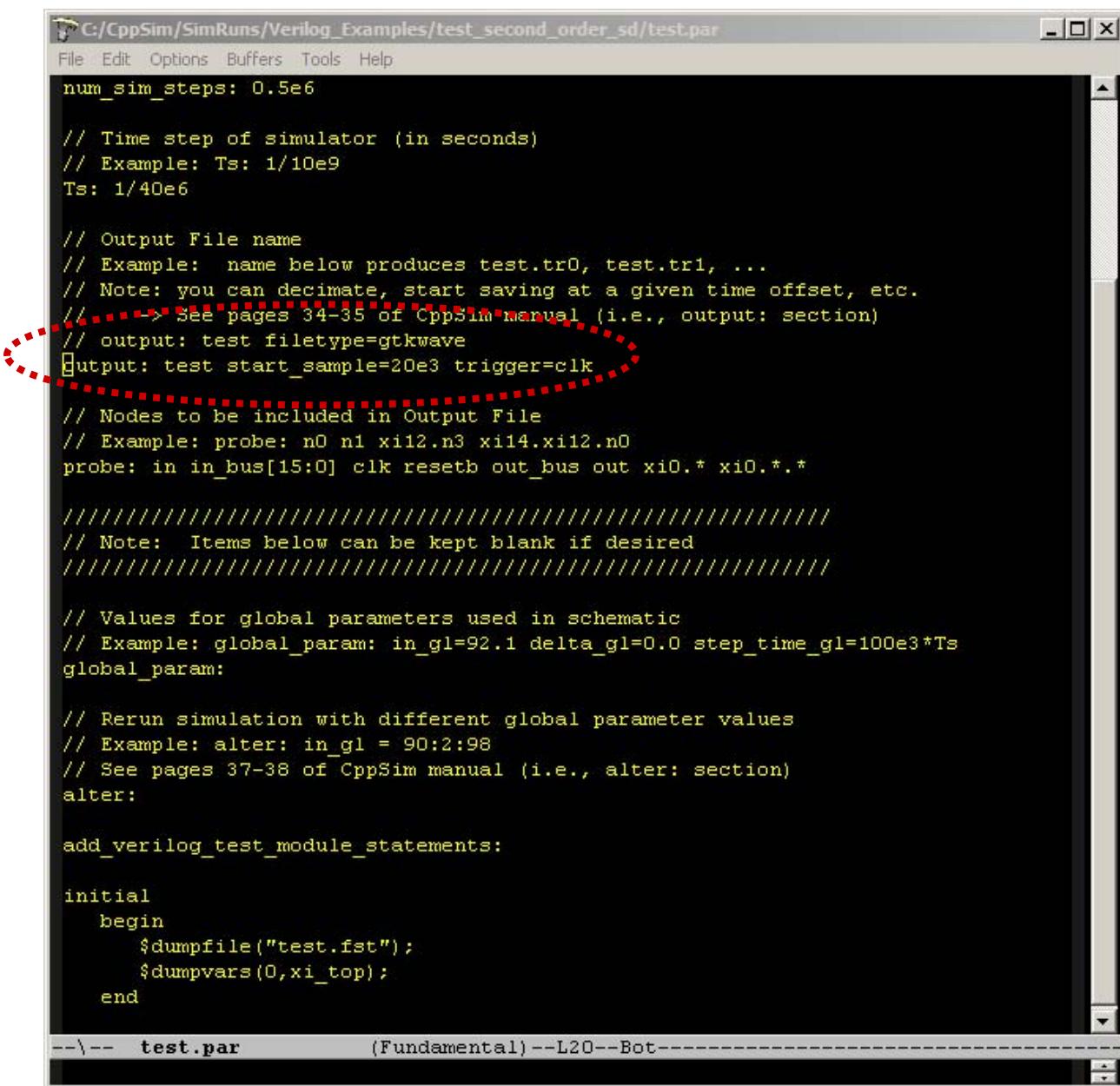
- Now that we have seen how VppSim saves signals for GTKWave, let's move forward with viewing them. Within GTKWave, click on **Open New Tab** and now select **test.fst** (not **test\_0.fst**) as the file to view.



- Using the same methods discussed in the previous section, you should be able to obtain a zoomed-in version of the input and output signals as shown below.



- One can also save signals for CppSimView, Matlab/Octave, and Python. To do so, click on **Edit Sim File** in the **VppSim Run Menu** (or simply edit the file if you still have it open from the previous exercise) and perform the following modifications to arrive at the file as shown below:
  - Comment out the line **output: filetype=gtkwave**
  - Uncomment the line **output: test start\_sample=20e3 ....**



```

C:/CppSim/SimRuns/Verilog_Examples/test_second_order_sd/test.par
File Edit Options Buffers Tools Help
num_sim_steps: 0.5e6

// Time step of simulator (in seconds)
// Example: Ts: 1/10e9
Ts: 1/40e6

// 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 filetype=gtkwave
// output: test start_sample=20e3 trigger=clk
// Nodes to be included in Output File
// Example: probe: n0 n1 xi12.n3 xi14.xi12.n0
probe: in in_bus[15:0] clk resetb out_bus out xi0.* xi0.*.*

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

// Values for global parameters used in schematic
// Example: global_param: in_g1=92.1 delta_g1=0.0 step_time_g1=100e3*Ts
global_param:

// Rerun simulation with different global parameter values
// Example: alter: in_g1 = 90:2:98
// See pages 37-38 of CppSim manual (i.e., alter: section)
alter:

add_verilog_test_module_statements:

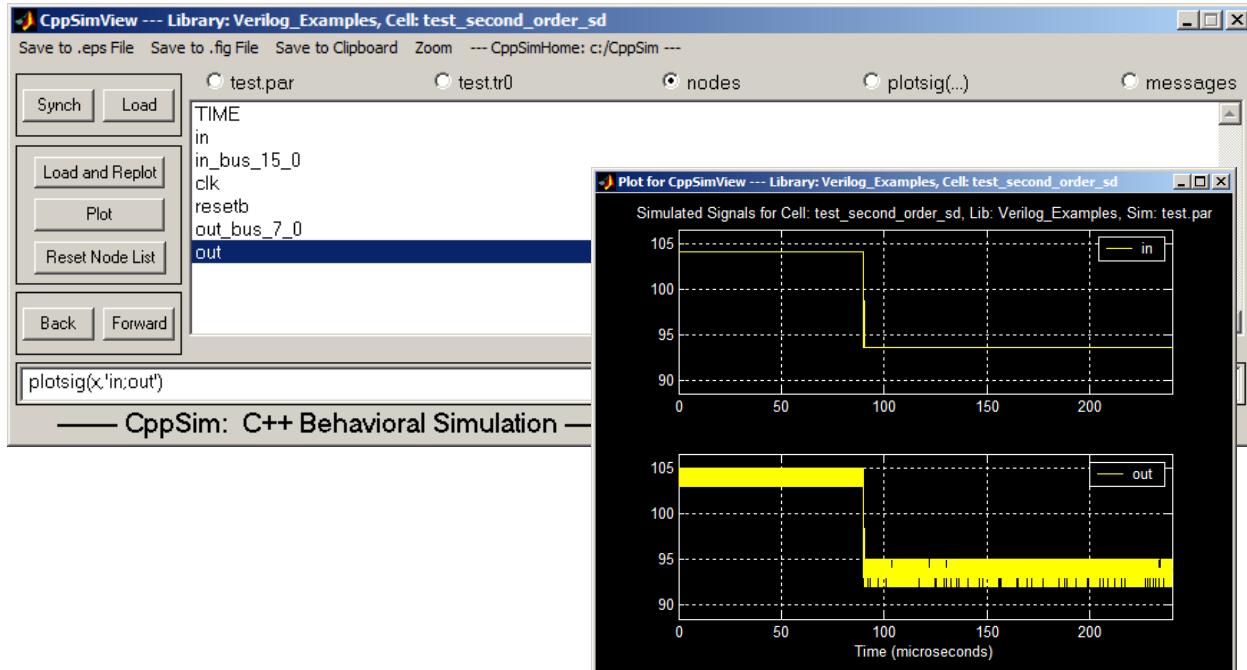
initial
begin
  $dumpfile("test.fst");
  $dumpvars(0,xi_top);
end

--\-- test.par          (Fundamental)--L20--Bot--

```

- Click on Compile/Run in the VppSim Run Menu to re-run VppSim with the new output: command settings. The simulation should completely as normal.

- Either open CppSimView and click on the **Load** button, or simply click on the **Load** button if it is still open from the previous exercise in this manual. You will see many of the same signals as were present for CppSim, although some are missing since VppSim doesn't support the \* command in saving signals. Click Reset Node List and then double-click on signals in and out in order to see the waveforms shown below.



## D. Key Distinguishing Features between CppSim and VppSim

The above exercises illustrated that either CppSim or VppSim could be used to run simulations with combined CppSim and Verilog blocks. Also, either CppSimView or GTKWave can be used to view results, with GTKWave being more handy when digital signals are present. Switching between CppSim and VppSim is fairly seamless as the same Simulation file (i.e., test.par) can be utilized. As such, a natural question to ask is why one might want to prefer one versus the other for simulating systems?

At a high level, we compare CppSim and VppSim as follows:

- CppSim produces C++ code for the overall simulation. When including Verilog modules, Verilator (written by Wilson Snyder) is used to convert the Verilog modules to C++ classes, which are directly incorporated within the overall simulation. As such, Verilog support is limited to what Verilator can handle, which is currently restricted to synthesizable code. Verilog testbenches are not supported since the main simulation engine is custom written and compiled C++ code.
- VppSim produces Verilog code for the overall simulation, and therefore directly supports Verilog testbenches as well as synthesizable and functional Verilog code. In the Windows version of CppSim, Icarus Verilog (written by Stephen Williams) is used as the main Verilog simulator, and VppSim is used to seamlessly incorporate CppSim modules and create the overall Verilog description. To incorporate the CppSim modules, VppSim first creates their

corresponding C++ classes and connects Verilog wrapper modules which access the C++ classes using the Verilog PLI. In doing so, full CppSim module support is achieved without translation, and standard Verilog simulation engines are supported since they universally include the PLI interface. To include Verilog testbench statements, two Simulation file (i.e., `test.par`) commands are available (as further described in the CppSim Reference Manual):

- **add\_verilog\_test\_file\_statements:** includes statements at the very top level of Verilog simulation, which is especially useful for ``define` and ``include` Verilog statements.
- **add\_verilog\_test\_module\_statements:** includes statements within the top module of the Verilog code, which is especially useful for controlling top level signal values and creating GTKWave dumpfiles as shown earlier in this section.

In addition to including testbench statements using the above commands, one can also include Verilog modules which are not included within the schematic. Two commands are provided for this:

- **add\_top\_verilog\_library\_file\_statements:** places the included Verilog modules at the beginning of the Verilog simulation file.
- **add\_bottom\_verilog\_library\_file\_statements:** places the included Verilog modules at the end of the Verilog simulation file. This is useful for avoiding statements such as `timescale` in the included modules for impacting the schematic level modules in the simulation.

The key issue between CppSim and VppSim is the use of custom C++ code versus a Verilog simulator, respectively, for the simulation engine. Some key comparison points between these two approaches are as follows:

- The C++ code generated by CppSim corresponds to constant time step simulation of the overall system. This method is quite useful for analog blocks such as filters and signals such as noise which must be updated on a regular basis, and directly compiled C++ code is quite efficient at doing such regular operations.
- The Verilog code generated by VppSim corresponds to event driven simulation of the overall system based on signal transitions. Since signal transitions within digital systems are typically sparse, event-driven simulation can be much faster than constant-time simulation for such systems. However, the extra overhead of scheduling events is a burden when dealing with blocks that need constant update such as filters.

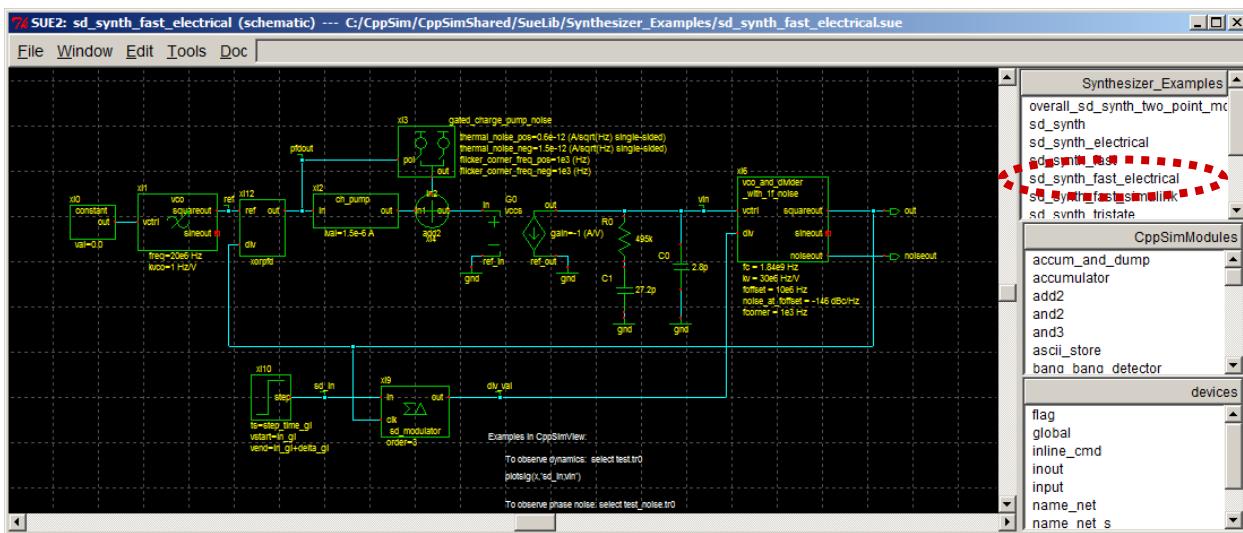
Given the above, we see that CppSim is best for analog-intensive systems that include digital blocks for support of the analog functionality. In contrast, VppSim is best for digital-intensive systems in which analog blocks play a less significant role, especially for cases where Verilog testbenches are desirable. Since the main tradeoff is simulation speed, the combination of CppSim and VppSim allows a complementary environment between analog and digital designers in which each side can easily share blocks with the other.

## The Basics of Including Electrical Elements in CppSim/VppSim Simulations

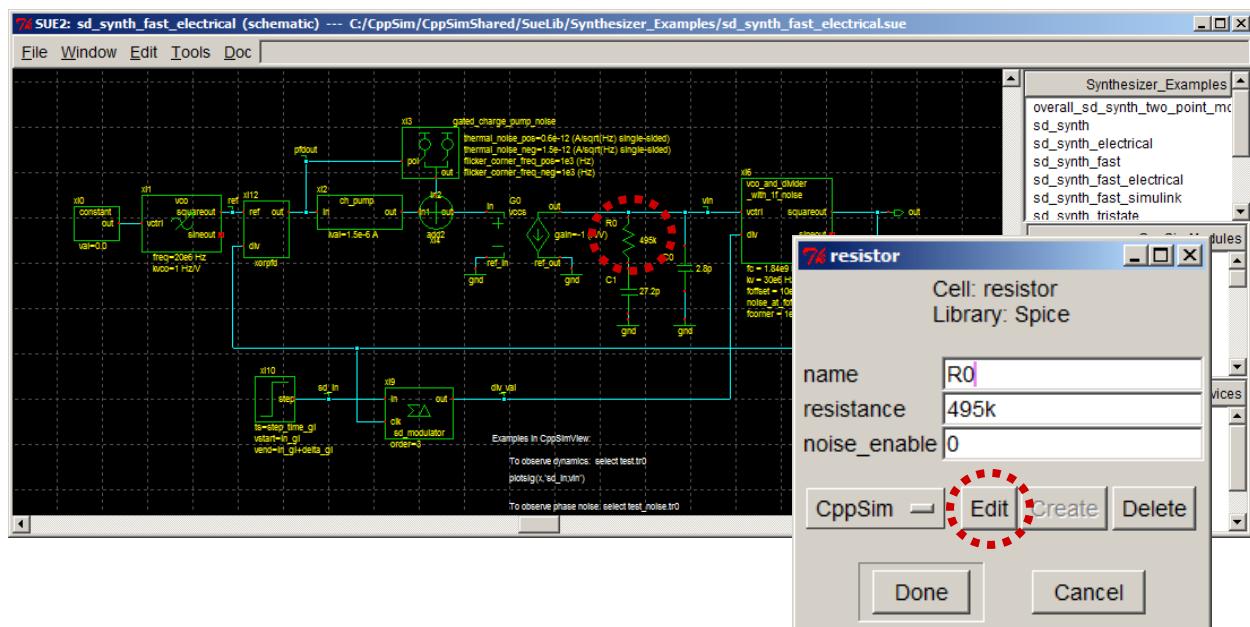
Both CppSim and VppSim support the inclusion of electrical elements within simulations in which nodal analysis is used to solve for the nodal signal values. The key issues associated with including such elements are discussed here, with examples focusing on CppSim since VppSim will be identical from the user standpoint.

### A. Including Electrical Elements within System Descriptions

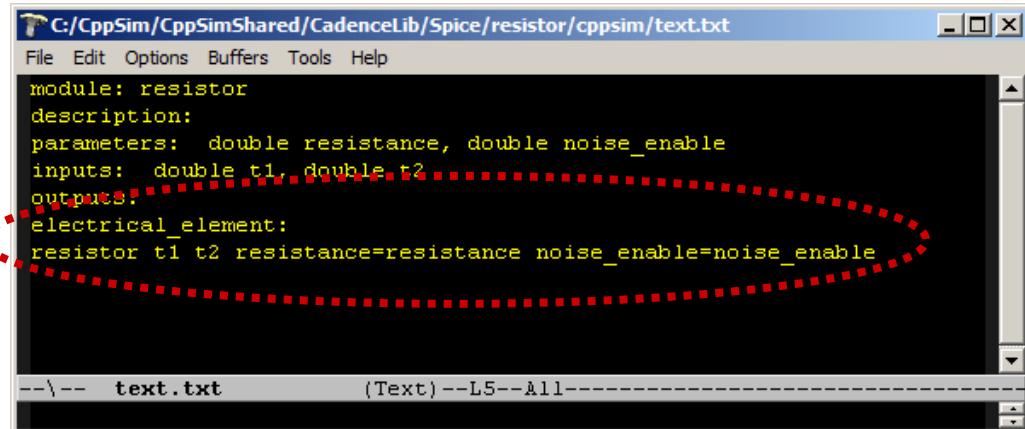
- Within Sue2, go to the **sd\_synth\_fast\_electrical** schematic by selecting this cell within the **Synthesizer\_Examples** library using the schematic listbox as indicated below.



- Double-click on element **R0** and then click on the **Edit** button within the properties window that appears.

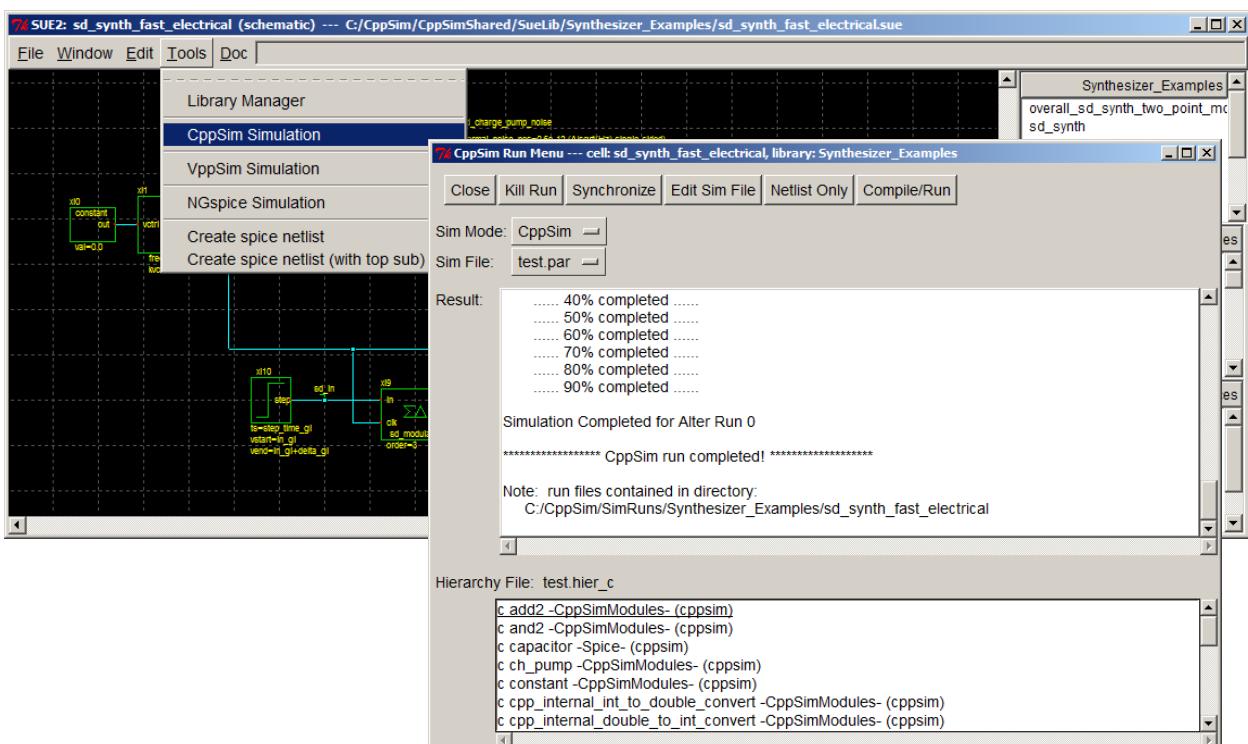


- As we see from the code below, **R0** corresponds to a module named **resistor**. Rather than having a **code:** section, **resistor** is instead described with an **electrical\_element:** command. This command can be composed of one or many lines which include electrical primitives such as resistors, capacitors, inductors, voltage\_controlled voltage sources, etc. Please see the **electrical\_element:** description within the CppSim Reference Manual for more details on this command. In this case, we see that the **electrical\_element:** command consists of only one element – a resistor with terminals and parameters matching that of the module. Note that for **electrical\_element:** modules, it does not matter if the terminals correspond to input or output nodes since nodal analysis is used to solve for the signal values.

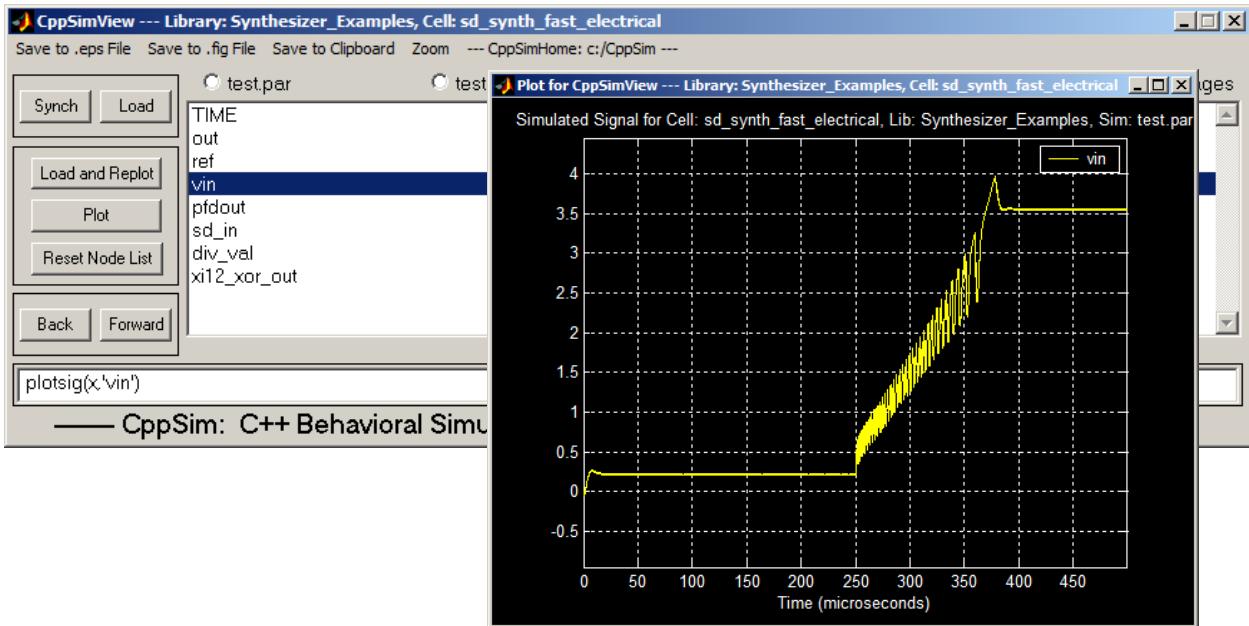


```
C:/CppSim/CppSimShared/CadenceLib/Spice/resistor/cppsim/text.txt
File Edit Options Buffers Tools Help
module: resistor
description:
parameters: double resistance, double noise_enable
inputs: double t1, double t2
outputs:
electrical_element:
resistor t1 t2 resistance=resistance noise_enable=noise_enable
--\-- text.txt (Text)--L5--All--
```

- Open the **CppSim Run Menu** window and then click on **Compile/Run** to run the CppSim simulation on the schematic. You will notice that it runs quite quickly. In fact, if you compare the time for completing the simulation versus running CppSim on the **sd\_synth\_fast** schematic, you'll notice that there is little speed penalty in using the nodal version of the loop filter as opposed to a **code:** based cppsim module such as **leadlagfilter** as used within schematic **sd\_synth\_fast**.



- Click on the CppSimView icon to start the viewer, and then examine the **vin** signal as shown below. In general, you can probe nodes associated with nodal elements in the same fashion as those produced by CppSim modules containing a **code:** section.



## B. Key Constraints When Using Electrical Elements

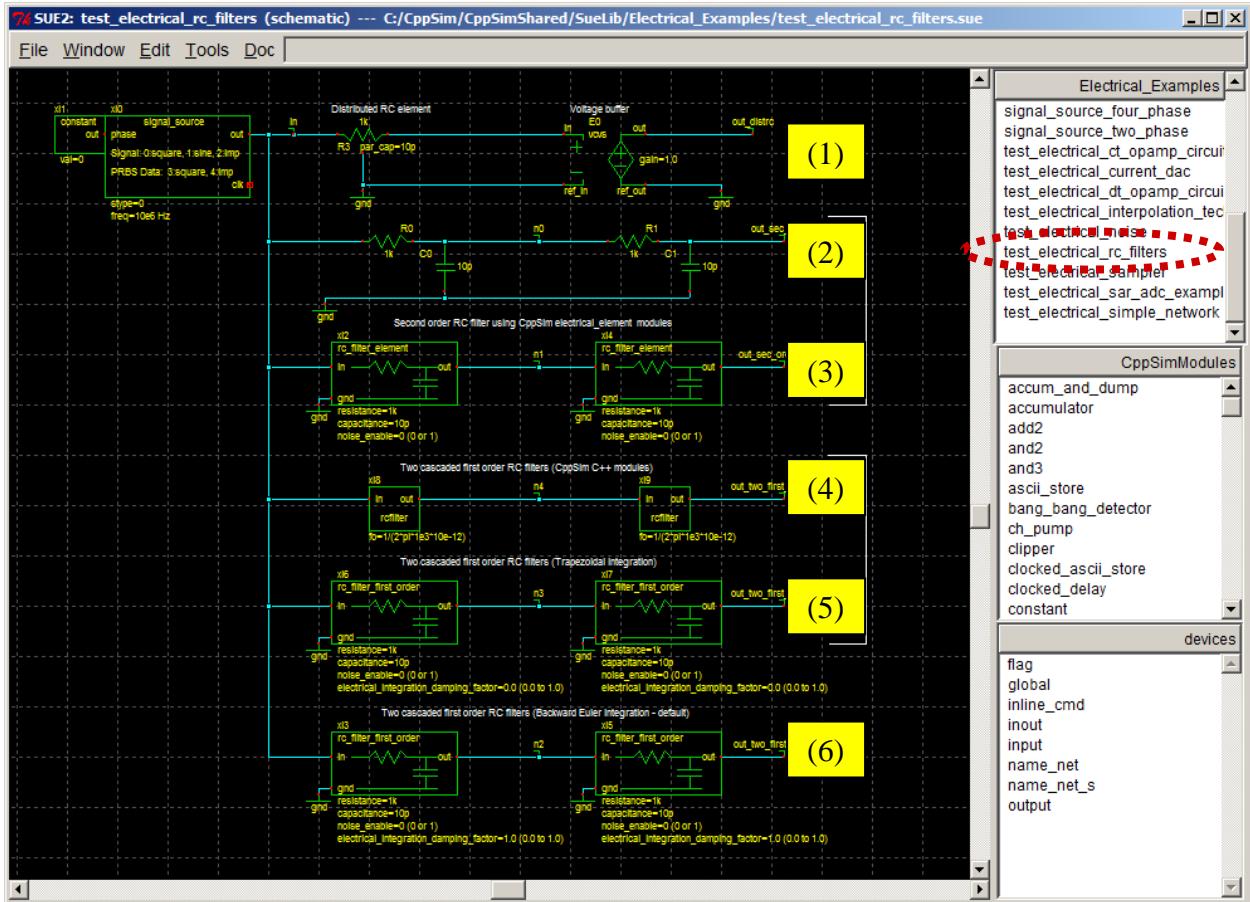
CppSim and VppSim are focused on achieving fast simulation at the system level. In general, nodal analysis of large systems becomes prohibitively expensive in terms of computation time. To maintain fast speed, CppSim and VppSim apply the following constraints when using electrical elements:

- Only linear electrical primitives and switches are supported. Such primitives include resistors, capacitors, inductors, voltage-controlled voltage and current sources, current-controlled current sources, independent voltage and current sources, and electrical switches with finite on and off resistance. The **electrical\_element:** command description within the CppSim Reference Manual contains more details on these supported primitives.
- Coupling between electrical elements does not extend beyond a single schematic level in the system hierarchy. When going between levels of hierarchy, ideal voltage buffers are inserted at the schematic input and output boundaries.
- All **electrical\_element:** primitives of a given module are inserted flat into the schematic within which the corresponding instance is placed. As such, even though coupling is not supported between electrical elements at different schematic levels, one can create higher level elements such as detailed linear two-port modules by placing their description within an **electrical\_element:** based module as opposed to a schematic-based module.
- Numerical integration of the nodal analysis formulation of electrical elements provides one parameter to the user, **electrical\_integration\_damping\_factor**, which takes on values between 0 and 1. The default value of 1 corresponds to backward Euler integration, whereas a value of 0 corresponds to trapezoidal integration. As the parameter varies between 1 and 0, the numerical integration method transitions between these two methods. The value of this parameter for the overall simulation is set by placing the **electrical\_integration\_damping\_factor:** command within the simulation file. However, it

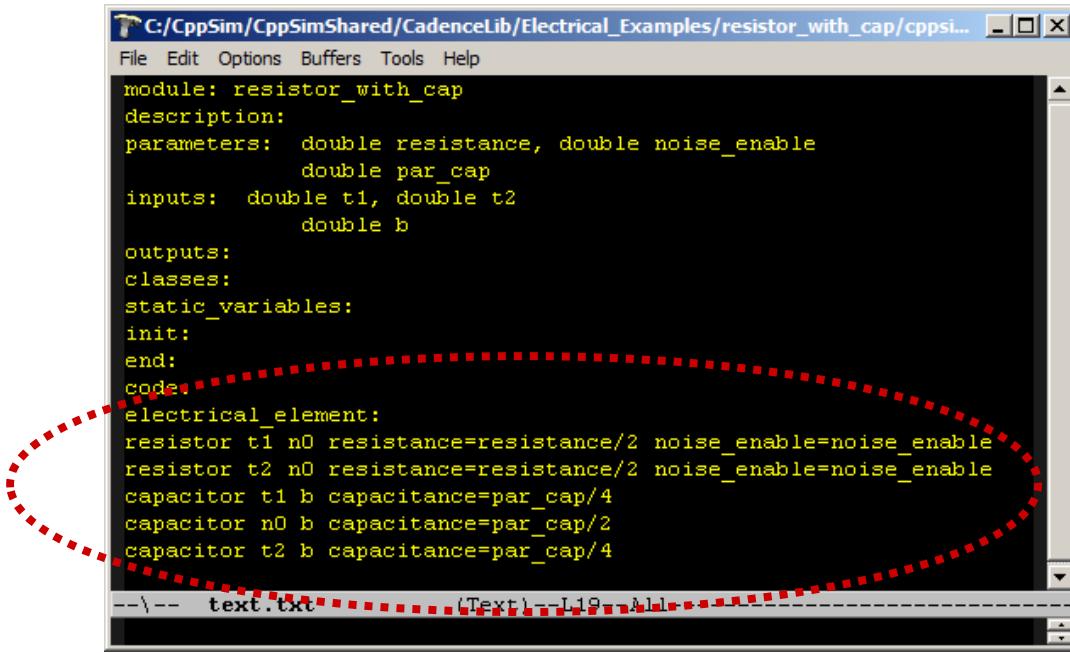
can also be set individually for modules containing electrical elements at the schematic level at lower levels of hierarchy. In such case, the module must simply contain a parameter called **electrical\_integration\_damping\_factor** which is set in the range of 0 to 1.

To better understand the above constraints, we will look at an example that shows different approaches to representing cascaded RC filters.

- Within Sue2, go to the **test\_electrical\_rc\_filters** schematic by selecting this cell within the **Electrical\_Examples** library using the schematic listbox as indicated below.



- The above example contains various approaches to representing cascaded RC filter sections:
  - 1) A resistor element with distributed capacitive loading that is implemented with the **electrical\_element:** command (Note that the voltage buffer included in the above schematic is simply there to illustrate how voltage-controlled voltage sources can be utilized).
    - To see the code as shown below, double-click on **R3** in the schematic and then click on **Edit** in the properties menu that appears. Note that the **electrical\_element:** command consists of five lines, the first two of which correspond to the resistance being split into two, and last three being the capacitors that connect to each side of the split resistance. Also note that the parameter values in each line can consist of expressions which involve the overall module parameters.

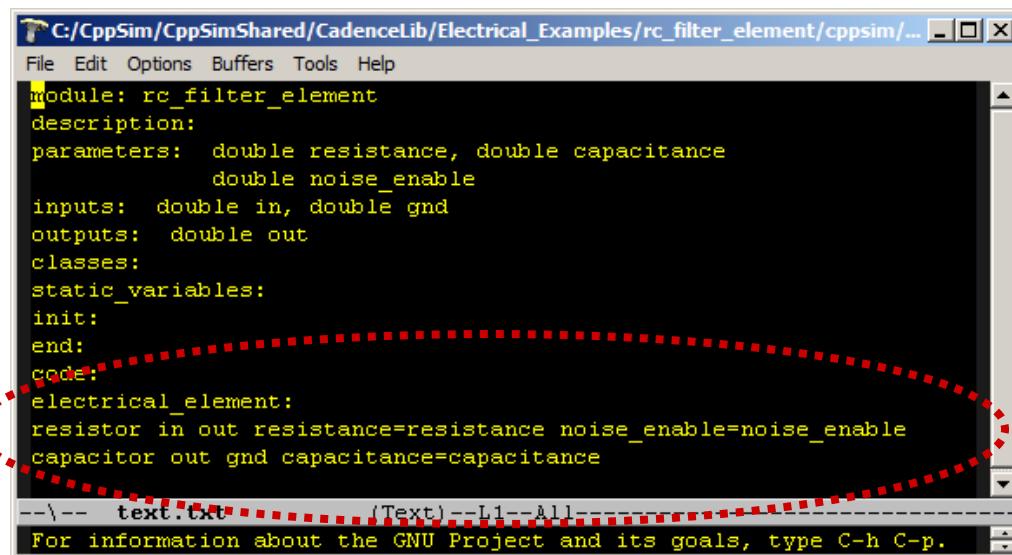


```

C:/CppSim/CppSimShared/CadenceLib/Electrical_Examples/resistor_with_cap/cppsim...
File Edit Options Buffers Tools Help
module: resistor_with_cap
description:
parameters: double resistance, double noise_enable
            double par_cap
inputs: double t1, double t2
        double b
outputs:
classes:
static_variables:
init:
end:
code:
electrical_element:
resistor t1 n0 resistance=resistance/2 noise_enable=noise_enable
resistor t2 n0 resistance=resistance/2 noise_enable=noise_enable
capacitor t1 b capacitance=par_cap/4
capacitor n0 b capacitance=par_cap/2
capacitor t2 b capacitance=par_cap/4
--\-- text.txt (Text)--L19--All-

```

- 2) A cascaded two-stage RC filter section. In this case, the two stages are coupled such that the poles formed by the network are altered by their mutual loading.
- 3) A cascaded two-stage RC filter network that has the same frequency response as directly drawing the RC stages as done in (2). Each RC section is implemented as a CppSim module using the **electrical\_element:** command.
  - o To see the code as shown below, double-click on **xi2** in the schematic and then click on **Edit** in the properties menu that appears. Note that the **electrical\_element:** command consists of two lines in this case, one of which corresponds to the resistor and the other to the capacitor in the first order RC network represented by the module.



```

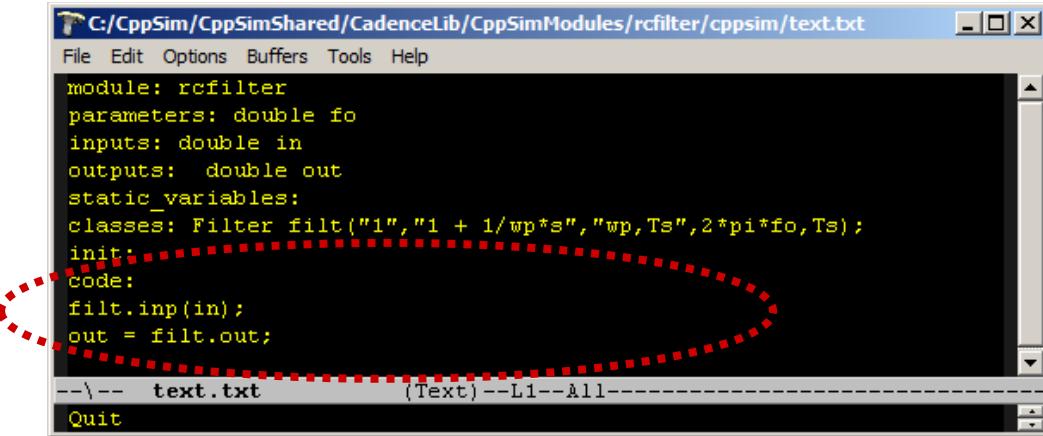
C:/CppSim/CppSimShared/CadenceLib/Electrical_Examples/rc_filter_element/cppsim...
File Edit Options Buffers Tools Help
module: rc_filter_element
description:
parameters: double resistance, double capacitance
            double noise_enable
inputs: double in, double gnd
outputs: double out
classes:
static_variables:
init:
end:
code:
electrical_element:
resistor in out resistance=resistance noise_enable=noise_enable
capacitor out gnd capacitance=capacitance
--\-- text.txt (Text)--L1--All-
For information about the GNU Project and its goals, type C-h C-p.

```

- 6) A cascaded two-stage RC filter consisting of two first order RC stages implemented as CppSim modules with the **code:** command. Unlike cases (2) and (3) above, the two first order

stages are un-coupled such that the frequency responses consists of two poles at the same frequency (i.e.,  $1/(2\pi RC)$ ).

- To see the code as shown below, double-click on **xi8** in the schematic and then click on **Edit** in the properties menu that appears. Note that the **code:** command makes use of the CppSim **Filter** class to easily realize the lead-lag filter operation.



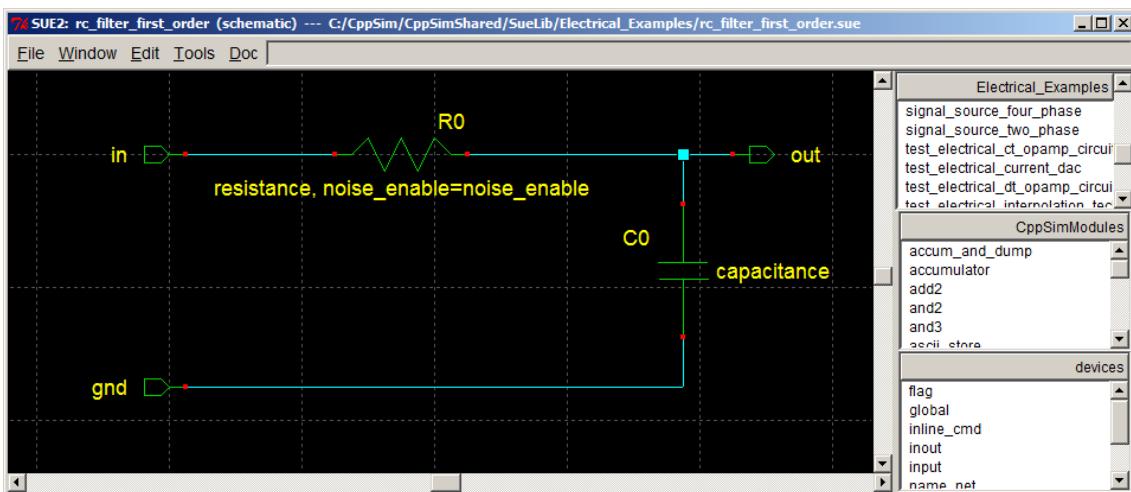
```

C:/CppSim/CppSimShared/CadenceLib/CppSimModules/rcfilter/cppsim/text.txt

File Edit Options Buffers Tools Help
module: rcfILTER
parameters: double fo
inputs: double in
outputs: double out
static_variables:
classes: Filter filt("1","1 + 1/wp*s","wp,Ts",2*pi*fo,Ts);
init:
code:
filt.inp(in);
out = filt.out;
--\-- text.txt (Text)--L1--All--
Quit

```

- A cascaded two-stage RC filter consisting of two first order RC stages implemented as RC schematics. Since the RC schematics occur at a lower level of hierarchy, they are uncoupled from each other. As such, the frequency response is the same as that of case (4).
  - To see the schematic as shown below, single-click on **xi6** and then press **e**. For this schematic, it is important that the **input** and **output** pins be configured as shown since it will impact how unity gain voltage buffers will effectively be inserted when the module is instantiated within other schematics.



- A cascaded two-stage RC filter consisting of two first order RC stages implemented as RC schematics. This is the essentially the same as case (5) above except for the setting of parameter **electrical\_integration\_damping\_factor**. In case (5), this parameter was set to 0 which corresponds to trapezoidal integration for nodal analysis. It turns out that trapezoidal integration matches the frequency response of the CppSim **Filter** class which uses the bilinear transform to create its filter response. In this case, the parameter **electrical\_integration\_damping\_factor** is instead set to 1 (the default value for nodal

analysis in CppSim), which corresponds to backward Euler integration. Case (6) will therefore give somewhat different results than cases (4) and (5), though the differences will become negligible for sufficiently small sample time,  $T_s$ , of the simulation. In general, backward Euler is chosen as the default method since it avoids artificial ringing in simulations which can occur when using the trapezoidal integration method.

## **Using The CppSim Library Manager**

The **CppSim Library Manager** provides the ability to do the following tasks related to libraries:

- Creation and deletion,
- Inclusion in or removal from the **sue.lib** file
- Importing and exporting to allow easy transfer to other users

To explain the second item in more detail, note that **Sue2** only pays attention to libraries that are contained in its **sue.lib file** (located in c:/CppSim/Sue2/sue.lib).

In this section we will discuss the key points related to the above operations. One key issue to consider for all library operations is that **Sue2** considers libraries only as a means to organize modules, and *not* as a means to distinguish between modules. In other words, all modules must have a unique name in **Sue2** – you cannot have more than one module with the same name, even if those modules are in different libraries. As such, the **CppSim Library Manager** is set up to check for such name clashing and to take steps to deal with it. This issue is explained in more detail in the sections to follow.

Another key issue is that the CppSim package separates libraries into **Private** and **Shared** categories. **Private** libraries are located in c:/CppSim/SueLib and c:/CppSim/CadenceLib for the Sue2 modules and their corresponding CppSim module code, respectively. **Shared** libraries are located in c:/CppSim/CppSimShared/SueLib and c:/CppSim/CppSimShared/CadenceLib for the Sue2 modules and their corresponding CppSim module code, respectively. Note that the CppSimShared directory can be placed at an arbitrary location on the system in order to facilitate sharing of modules between several users, and is therefore not constrained to its default location of c:/CppSim/CppSimShared.

One important point in working with **Private** and **Shared** libraries is that **Private** libraries take precedence over **Shared** ones in the case of having a common library name. As an example, in the case where there is a **CppSimShared** library in both the **Private** and **Shared** locations, the **Shared** version of the library will be completely ignored (i.e., all of its modules will be ignored) and only the **Private** version considered. The CppSim Library Manager circumvents the precedence relationship by automatically renaming the **Private** version of the library in case the **Shared** version is desired. This renaming will only be observed if one looks at the actual **Private** library directory name (i.e., using Windows Explorer), and will be invisible to the user if they are doing all operations within the **CppSim Library Manager**.

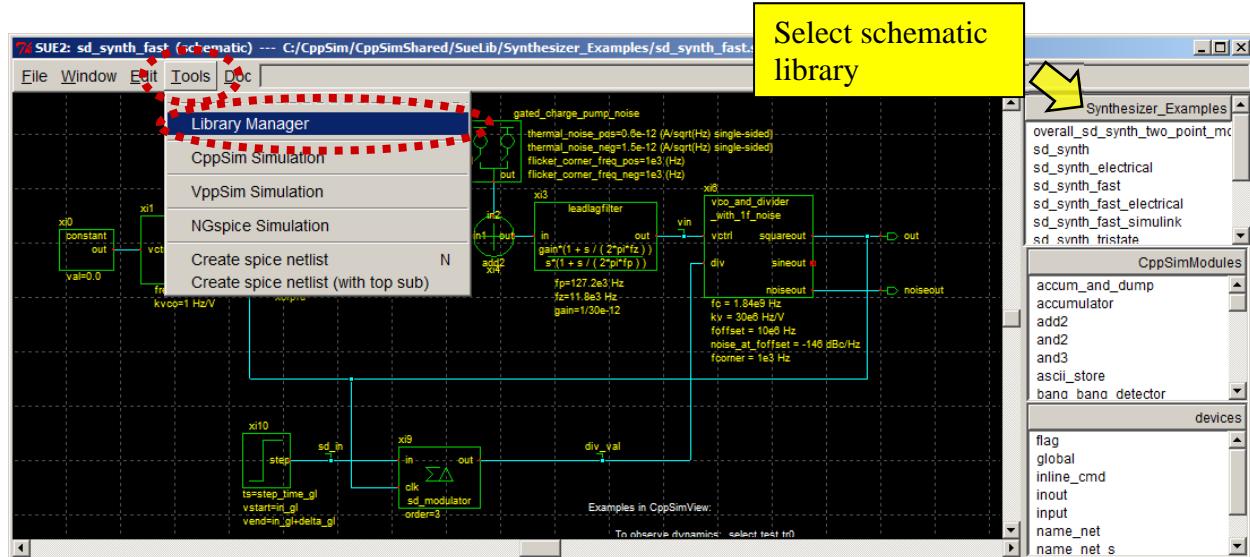
We now proceed with discussing the **CppSim Library Manager** in more detail.

### **A. Basic Operations**

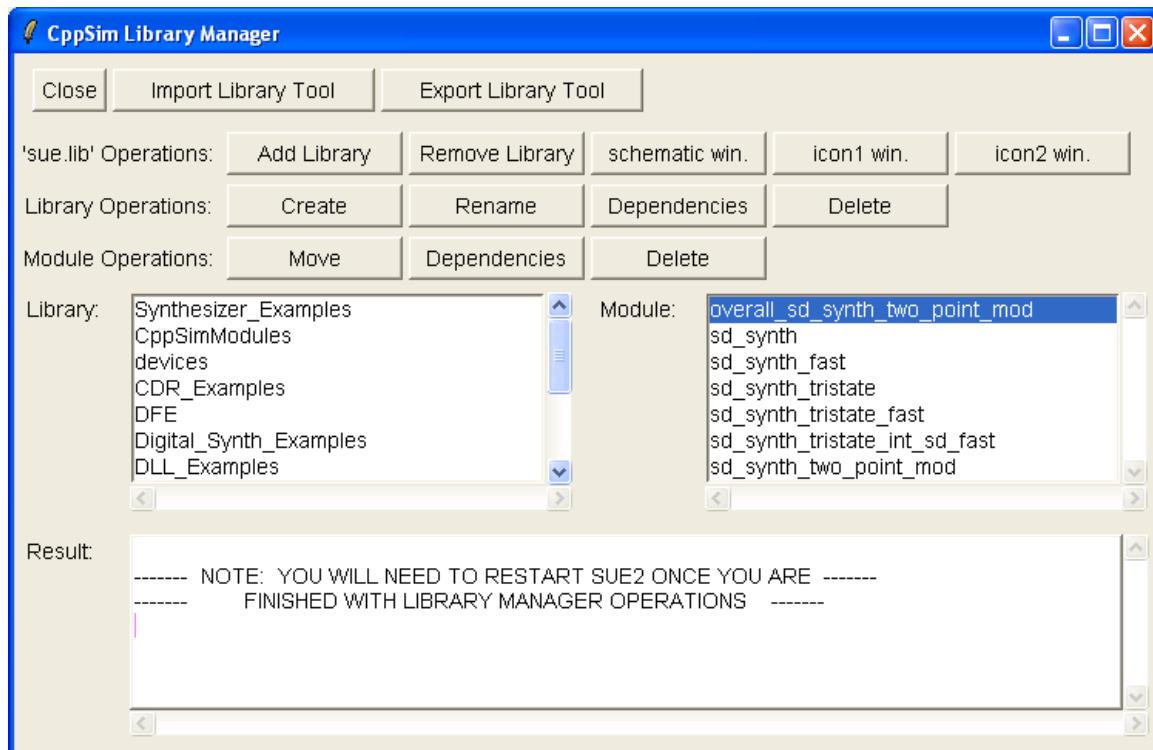
The CppSim library manager is part of the **Sue2** package, and must be run from its **Tools** menu item.

- Within **Sue2**, select the schematic library as **Synthesizer\_Examples** and the schematic module as **sd\_synth\_fast** as indicated in the figure below.

- Now click on the **Library Manager** option under the **Tools** menu item as also shown in the figure below.



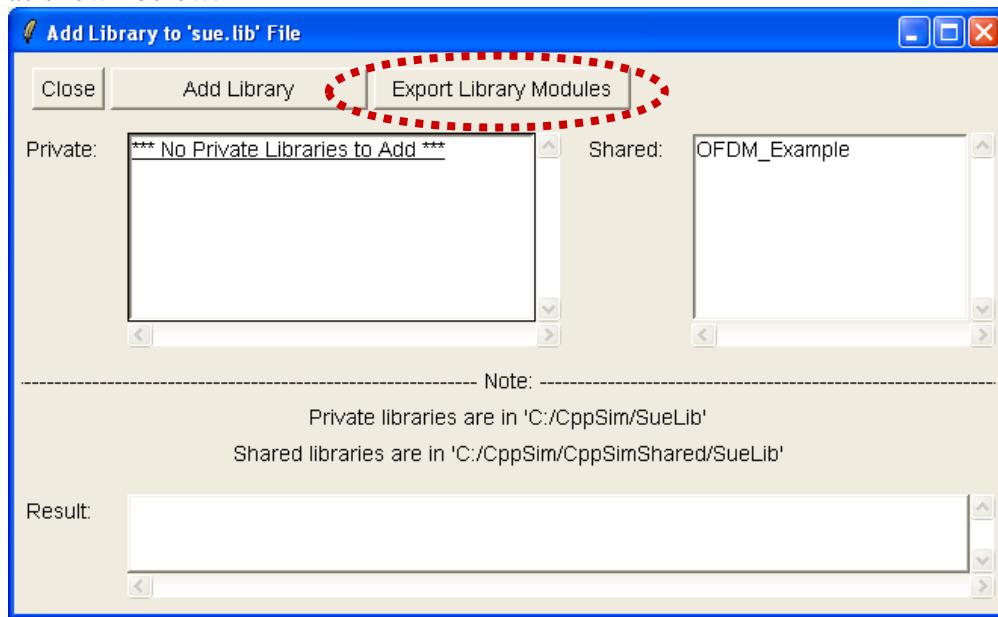
- The CppSim Library Manager window should appear as shown below.



Since many of the above operations are fairly obvious, we will simply discuss the key points of the overall interface:

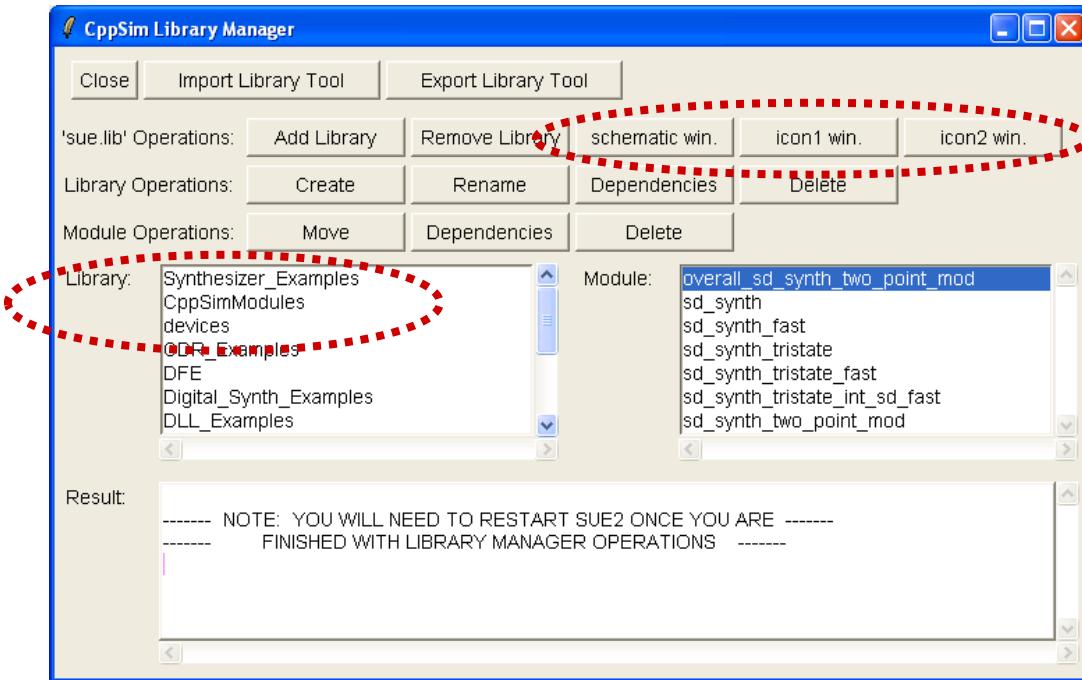
## sue.lib Operations

- Adding or removing a library from the **sue.lib** file has no impact on the actual files contained in that library. Instead, **Sue2** is simply directed to either consider or ignore those files. This feature is mainly directed toward streamlining **Sue2** operations such that only those libraries of key interest are considered at a given time even though more libraries are present in the package.
- Name clashes between cells are considered for all modules that are included in the **sue.lib** file. As such, if you attempt to add a library that has modules which will name clash with existing modules, the Library Manager will not allow the library to be added.
- In the case where there are name clashes and you do indeed want to access the modules in the library, one possibility is to enable automatic renaming of the modules by first exporting them and then importing them using the **Import Library Tool** discussed later in this document. The export operation is done by first clicking on the **Add Library** button in the CppSim Library Manager, and then by clicking on the **Export Library Modules** button in the resulting window as shown below.



- Libraries to be added are separated into **Private** and **Shared** versions, which are distinguished by their directory locations (see the discussion of **Private** and **Shared** libraries in the beginning portion of this section). Using the **Add Library** command allows use of either the **Private** or **Shared** version or neither one of them. Note that you cannot include *both* **Private** and **Shared** versions of the same library. Note that if the **Shared** version of a library is chosen instead of the **Private** version, the **CppSim Library Manager** changes the directory name of the **Private** version. This name change will only be observed if one views the **Private** directory name directly (such as by using Windows Explorer), and will be invisible when working within the **CppSim Library Manager**.
- The **schematic win.**, **icon1 win.**, and **icon2 win.** buttons in the **CppSim Library Manager** window (circled below) allow default settings for the starting library shown in each of those windows. Note that in the **Library:** list, the first entry (i.e., **Synthesizer\_Examples**) corresponds to the starting **schematic window** library, the second entry (i.e.,

**CppClassModules**) corresponds to the starting **icon1 window** library, and the third entry (i.e., **devices**) corresponds to the starting **icon2 window** library.



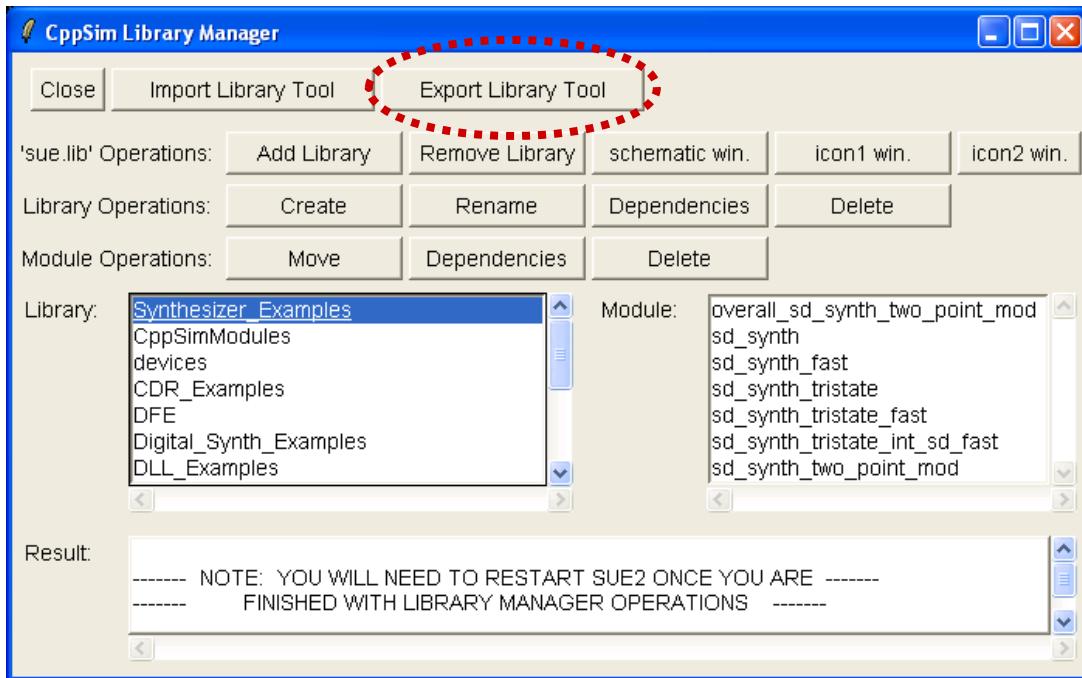
## Library Operations and Module Operations

These are fairly straightforward to figure out, so that the user is encouraged to play around with the various buttons to better understand their functionality.

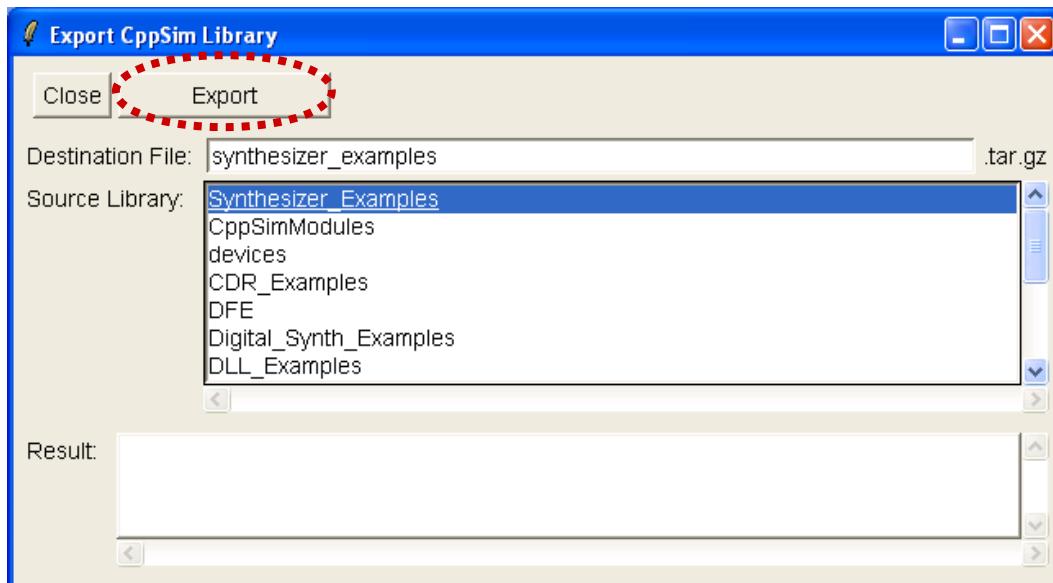
### B. Exporting CppSim Libraries

The **Export Library Tool** allows a library to be archived into a .tar.gz file, which can then be easily passed on to other users or archived for backup. A key feature of this tool is that it not only exports the modules within the selected library, but also included any modules from other libraries that the targeted library modules depend on. By doing so, every dependency of the library is included in the exported file, so that other users can reliably use its corresponding modules without needing to install additional supporting libraries.

- To access the **Export Library Tool**, click on its associated button within the **CppClass Library Manager** window (as shown below).



- To export a given library, simply select it in the **Source Library:** list and then press the **Export** button, as indicated in the figure below. The resulting export file (i.e., **synthesizer\_examples.tar.gz** in this case) will be located in `c:/CppSim/Import_Export`.



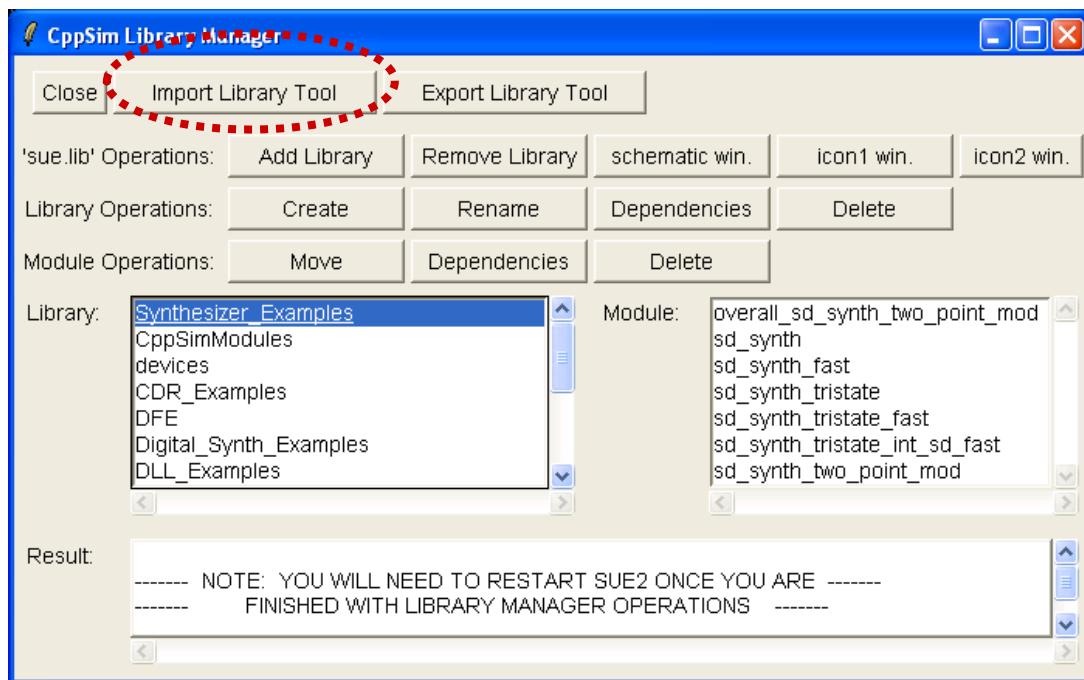
### C. Importing CppSim Libraries Generated from the Export Library Tool

The **Import Library Tool** allows a CppSim library to be imported from either a **.tar.gz** file created by the **Export Library Tool** (as discussed in the previous section). One should note that this operation is fairly sophisticated in order to avoid name clash issues between newly imported versus previously existing modules. In particular, all modules to be considered for import will be checked to see if their name coincides with any existing modules in the package (regardless of their library

location). If there is indeed a name clash, the Sue2 file and CppSim code file of the to-be-imported and existing module are compared to see if there are any differences between them. If there are no differences (i.e., the Sue2 and CppSim files of the to-be-imported and existing modules match each other), then the module to be imported is ignored since it already exists in the package. If there are indeed differences, then the name of the module to be imported is changed (i.e., **add2** would become **add2\_2**, **vco\_2** would become **vco\_3**, etc.) and then imported into the package. The name changes are then propagated to all imported schematics that use the relevant cells.

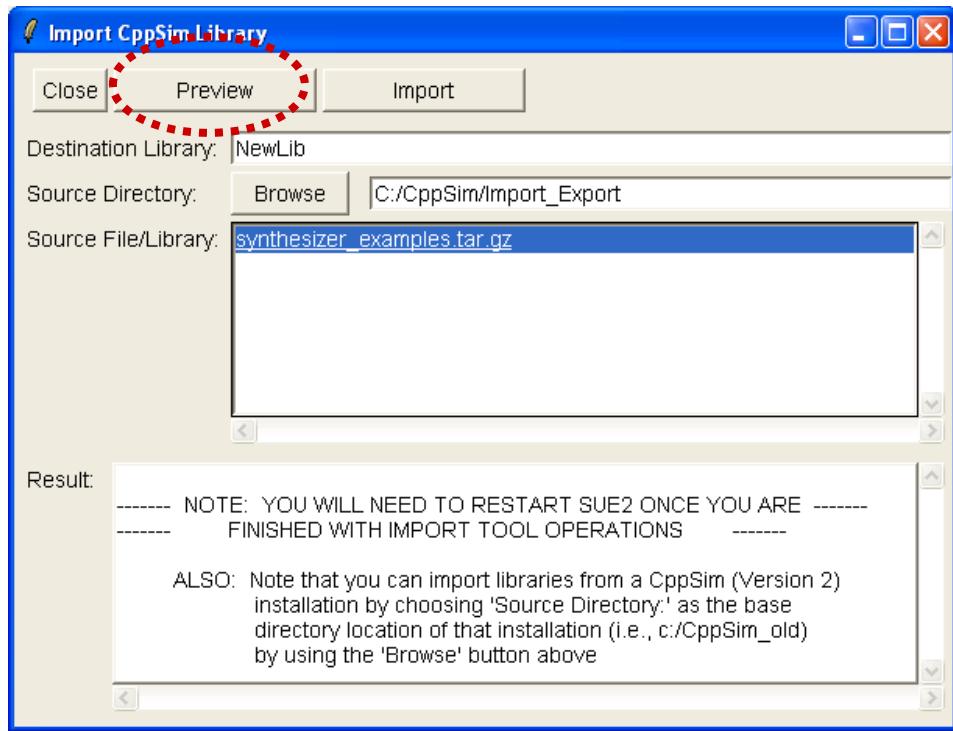
We now explain basic operations of using the **Import Library Tool**

- Within the **CppSim Library Manager** window, click on the **Import Library Tool** button as shown below.

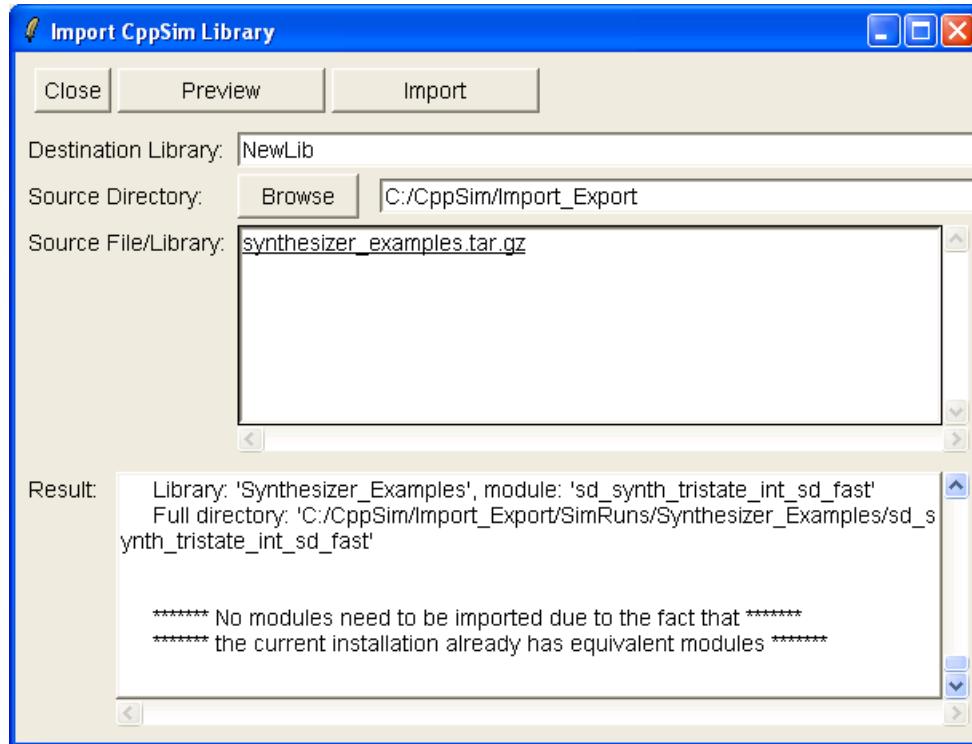


- After performing the above operation, the **Import CppSim Library** window should appear as shown below. In this case, assuming that you exported the **Synthesizer\_Examples** library while playing with the **Export Library Tool** in the previous section, you should see the option of importing **synthesizer\_examples.tar.gz**.
  - The **Destination Library** corresponds to the library name that the imported modules will be brought into. If the library does not currently exist, it will be automatically created. If it does exist, then the Import tool will add the imported modules to whatever modules are currently part of the library.
  - The Source directory is nominally set at c:/CppSim/Import\_Export, which is the place that Exported libraries are sent. When dealing with CppSim Version 3, the Source directory should not be changed from this value. However, if you want to import modules from a CppSim Version 2 installation, this directory should be changed to the base location of that installation (i.e., c:/CppSim\_old, as an example). This will be discussed in more detail in the next subsection.

- The **Preview** and **Import** buttons are identical in most of the operations they perform, with the key difference being that no modules are actually imported when using **Preview**. The value of **Preview** is to observe error messages, which typically relate to issues associated with name clashes, without having the Import operation follow through. Any error messages could then be examined without having the extra issue of cleaning up a problematic import operation.



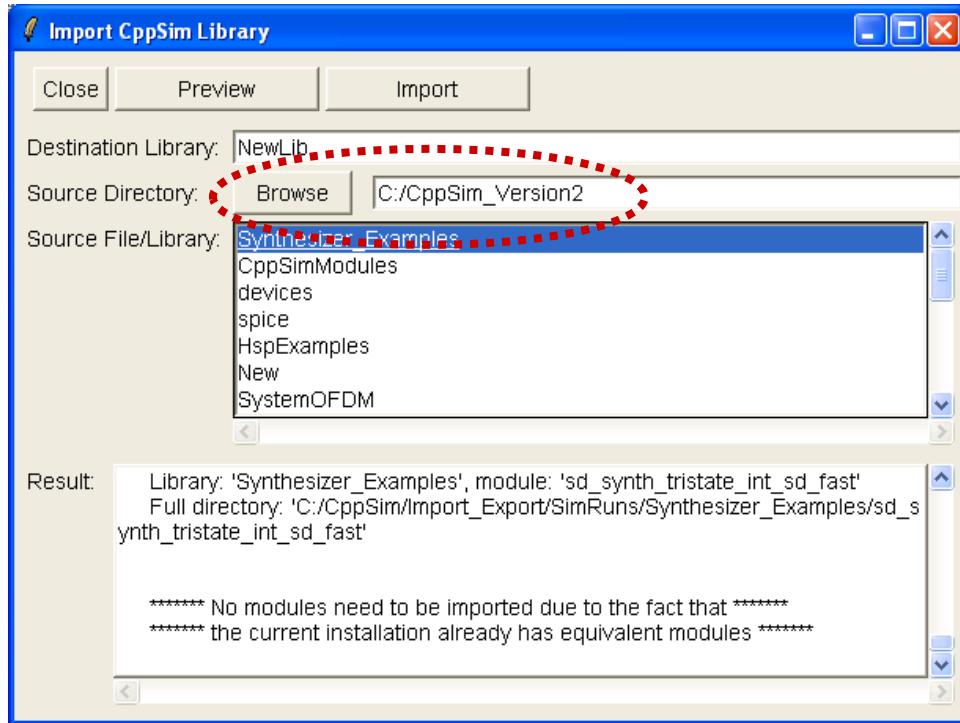
- As an example, click on the **Preview** button (as circled above) with **synthesizer\_examples.tar.gz** chosen as the **Source File/Library**. You should see several messages appear in the **Result** window, with the final message being displayed as shown below. Here we see that no modules would be imported since they are already contained in the distribution.



## D. Importing CppSim (Version 2) Libraries

The **Import Library Tool** also allows a CppSim (Version 2) library to be imported by simply specifying the installation's base directory location (i.e., c:/CppSim\_old, as an example) as the **Source Directory**. Note that this option only works with CppSim Version 2 installations, and not with alternate CppSim Version 3 installations.

- Continuing with the example from the previous subsection, push the **Browse** button in the **Import CppSim Library** window, and then select the base directory of a CppSim (Version 2) installation. In the example shown in the figure below, we have chosen c:/CppSim\_Version2.



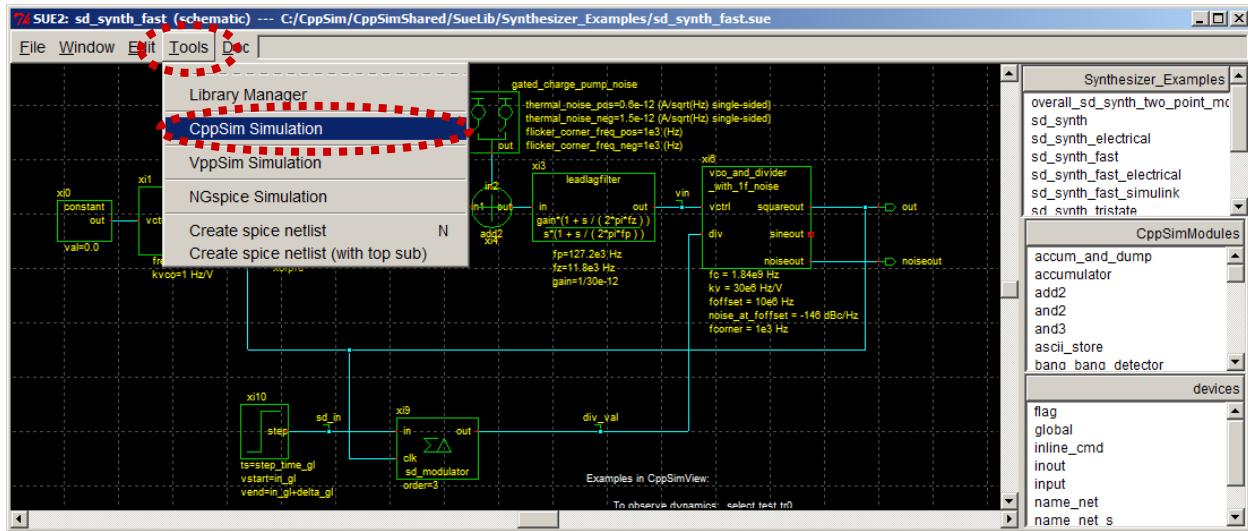
- Now choose a **Source Library**, and then either **Preview** or **Import** as desired.

## Creating Matlab Mex Functions and Simulink S-Functions

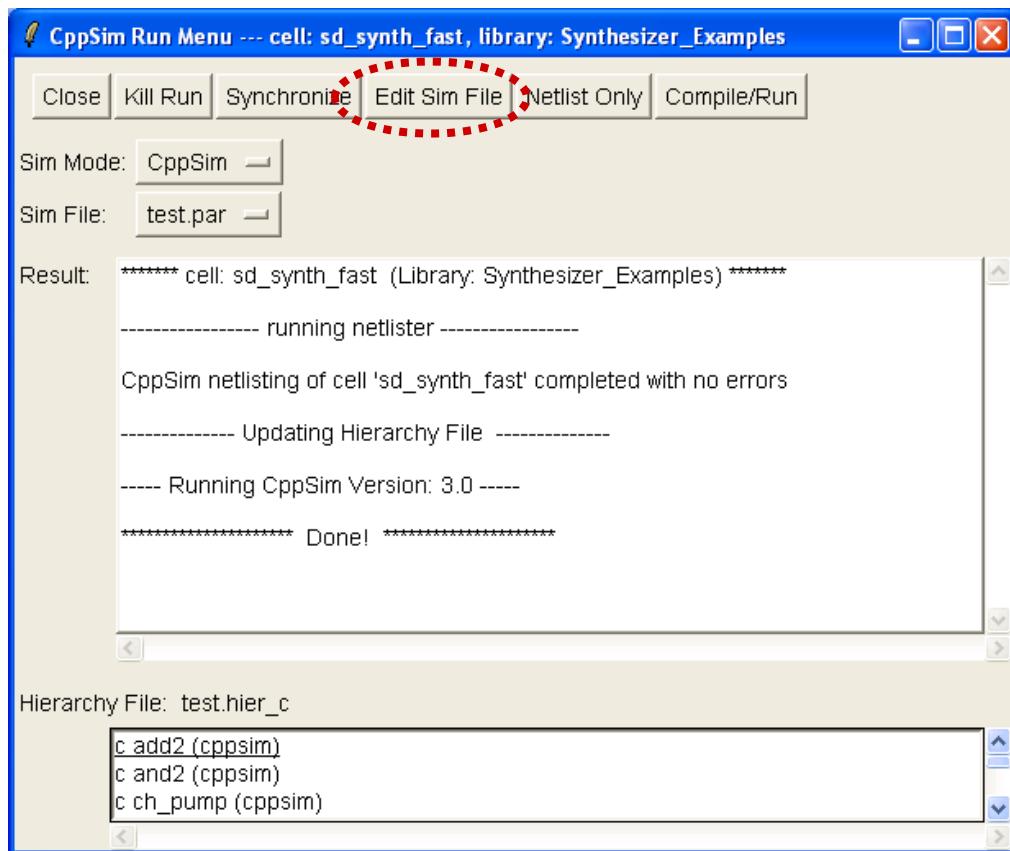
One tremendous benefit of using C++ as a behavioral language is that the resulting code is portable and can be accessed from other packages. In particular, it is relatively straightforward to embed CppSim simulation files within other software packages such as Matlab and Simulink. In particular, CppSim provides such support by automatically generating “wrapper” code to allow a given CppSim system to be compiled into Matlab as a mex function or into Simulink as an S-function. In this section, we will provide examples of creating a Matlab mex function and Simulink S-function.

### A. Matlab Mex Function Generation

- Select the schematic **sd\_synth\_fast** so that Sue2 contains this cell as shown in the figure below.
- Select the **CppClass Simulation** option within the **Tools** menu window, as also shown in the figure below.



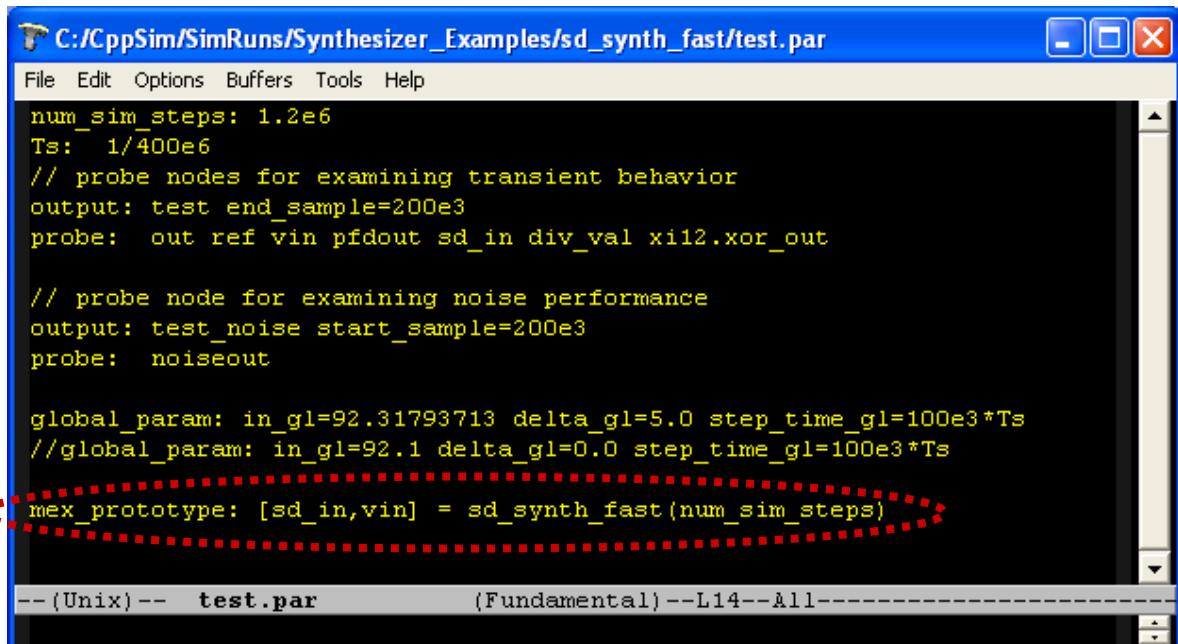
- Within the **CppSim Run Menu** that pops up, click on the **Edit Sim File** button as shown in the figure below.



- Notice the **mex\_prototype:** statement in the **Emacs** window that appears from clicking the **Edit Sim File** button. This statement must be included in the **Sim File** in order to create a mex function. The format for this function is:

**mex\_prototype: [out1,out1] = func\_name(in1,in2,param1,param2)**

In this case, the outputs are chosen as signals **sd\_in** and **vin**, the function name is **sd\_synth\_fast**, and the input parameter is **num\_sim\_steps**. Note that the **mex\_prototype:** statement is explained in more detail in the CppSim Reference Manual (i.e., **cppsimdoc.pdf**).



```

C:\CppSim\SimRuns\Synthesizer_Examples\sd_synth_fast\test.par
File Edit Options Buffers Tools Help
num_sim_steps: 1.2e6
Ts: 1/400e6
// probe nodes for examining transient behavior
output: test end_sample=200e3
probe: out ref vin pfdout sd_in div_val xi12.xor_out

// probe node for examining noise performance
output: test_noise start_sample=200e3
probe: noiseout

global_param: in_g1=92.31793713 delta_g1=5.0 step_time_g1=100e3*Ts
//global_param: in_g1=92.1 delta_g1=0.0 step_time_g1=100e3*Ts
mex_prototype: [sd_in,vin] = sd_synth_fast(num_sim_steps)

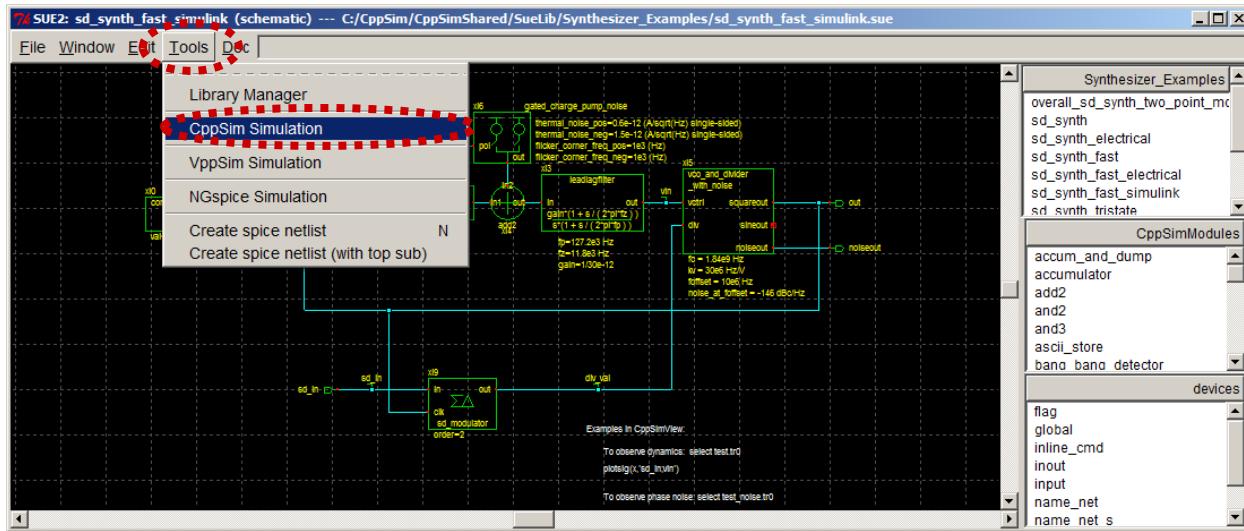
-- (Unix) --  test.par          (Fundamental) --L14--All--

```

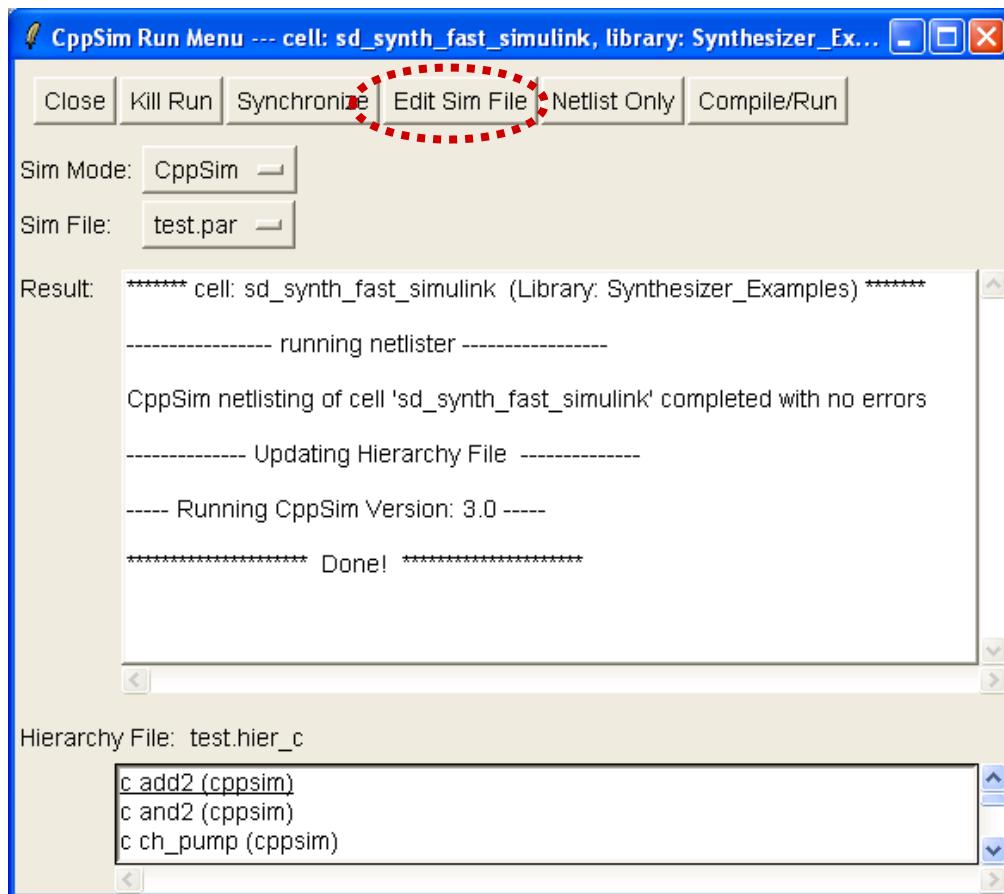
- Within the CppSim Run Menu window, select **Matlab** as the **Sim Mode** option, and then click on the **Compile/Run** button as shown in the figure below. As also shown in the figure, the resulting messages indicate how to compile and run the mex function. Note that you need to have a C++ compiler installed on your computer to enable the compile operation. Microsoft provides a free "Express" version of their C++ compiler, and instructions for utilizing this with Matlab are found at: <http://www.mathworks.com/matlabcentral/fileexchange/22689>

## B. Simulink S-Function Generation

- Select the schematic **sd\_synth\_fast\_simulink** so that Sue2 contains this cell as shown in the figure below.
- Select the **CppSim Simulation** option within the **Tools** menu window, as also shown in the figure below.



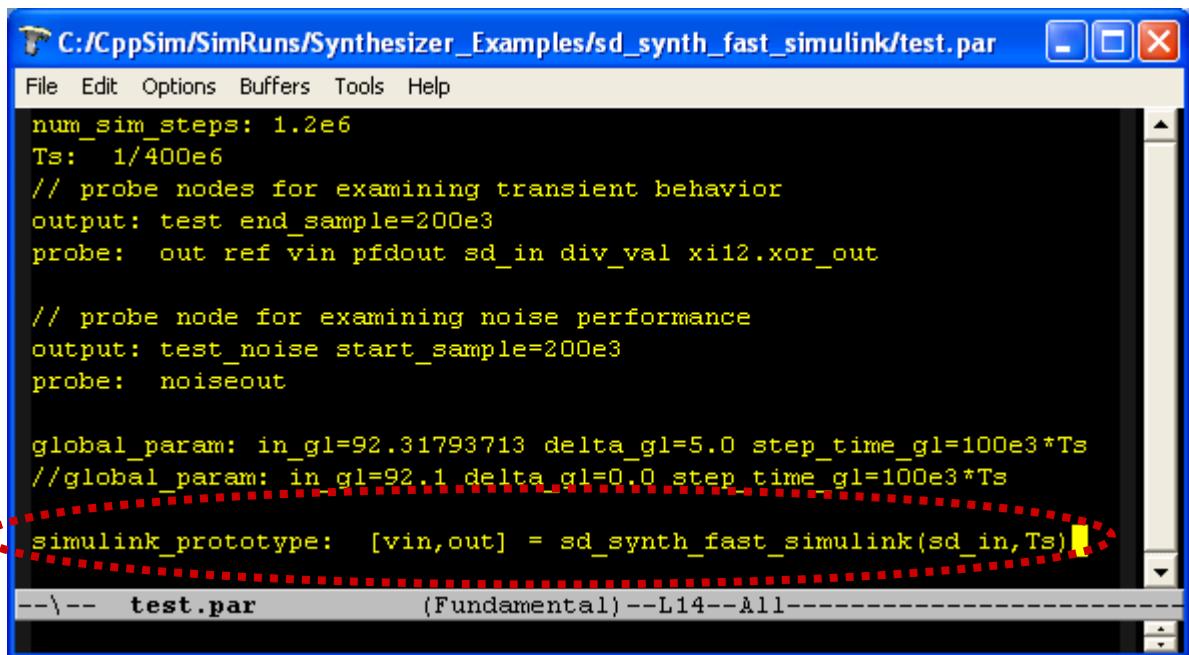
- Within the **CppClass Run Menu** that pops up, click on the **Edit Sim File** button as shown in the figure below.



- Notice the **simulink\_prototype:** statement in the **Emacs** window that appears from clicking the **Edit Sim File** button. This statement must be included in the **Sim File** in order to create an S-function. The format for this function is:

```
simulink_prototype: [out1,out1] = func_name(in1,in2,param1,param2)
```

In this case, the outputs are chosen as signals **vin** and **out**, the function name should be specified as is **sd\_synth\_fast\_simulink**, the input is **sd\_in**, and the parameter is **Ts**. Note that the **simulink\_prototype:** statement is explained in more detail in the CppSim Reference Manual (i.e., **cppsimdoc.pdf**).



```
C:/CppSim/SimRuns/Synthesizer_Examples/sd_synth_fast_simulink/test.par
```

```
File Edit Options Buffers Tools Help
```

```
num_sim_steps: 1.2e6
Ts: 1/400e6
// probe nodes for examining transient behavior
output: test end_sample=200e3
probe: out ref vin pfdout sd_in div_val xi12.xor_out

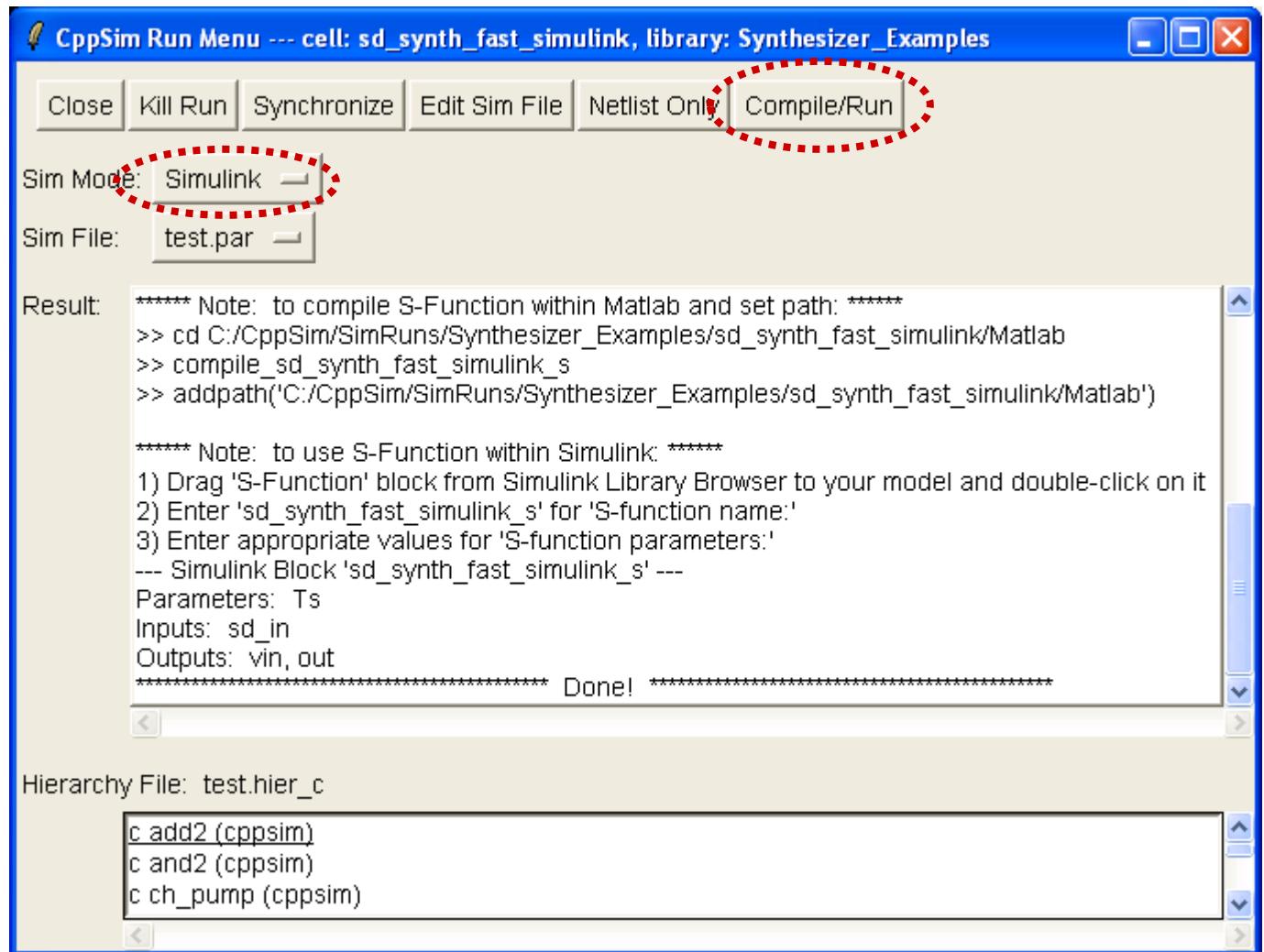
// probe node for examining noise performance
output: test_noise start_sample=200e3
probe: noiseout

global_param: in_g1=92.31793713 delta_g1=5.0 step_time_g1=100e3*Ts
//global_param: in_g1=92.1 delta_g1=0.0 step_time_g1=100e3*Ts

simulink_prototype: [vin,out] = sd_synth_fast_simulink(sd_in,Ts)
```

```
--\-- test.par (Fundamental)--L14--All-----
```

- Within the CppSim Run Menu window, select **Simulink** as the **Sim Mode** option, and then click on the **Compile/Run** button as shown in the figure below. As also shown in the figure, the resulting messages indicate how to compile and run the S-function. Note that you need to have a C++ compiler installed on your computer to enable the compile operation.



## Using Python with CppSim

While using the CppSim Run Menu and CppSimView is a convenient interface for beginners, more advanced users may want to consider running their simulations and doing post-processing tasks directly in Python. To use CppSim within Python, you simply need to import the CppSim Data module (which comes with the standard CppSim installation) by including the following lines in a given Python script:

```
# import cppsimdata module
import os
import sys
if sys.platform == 'darwin':
    home_dir = os.getenv("HOME")
    sys.path.append(home_dir + '/CppSim/CppSimShared/Python')
else:
    cppsimsharedhome = os.getenv("CPPSIMSHAREDHOME")
    sys.path.append(cppsimsharedhome + '/Python')
from cppsimdata import *
```

The CppSim Data module provides a class called **CppSimData** to allow easy loading of simulation data into Python, a function called `cppsim()` to run CppSim simulations within Python, and a supporting function for plotting phase noise. While we will show some simple examples below, one should read the manual **CppSim and Ngspice Data Modules for Python** that is available in the Doc menu of Sue2 for further details.

For the Python examples below, it is highly recommended that you download and install the Express (i.e., free) version of the Enthought Canopy distribution of Python available at:

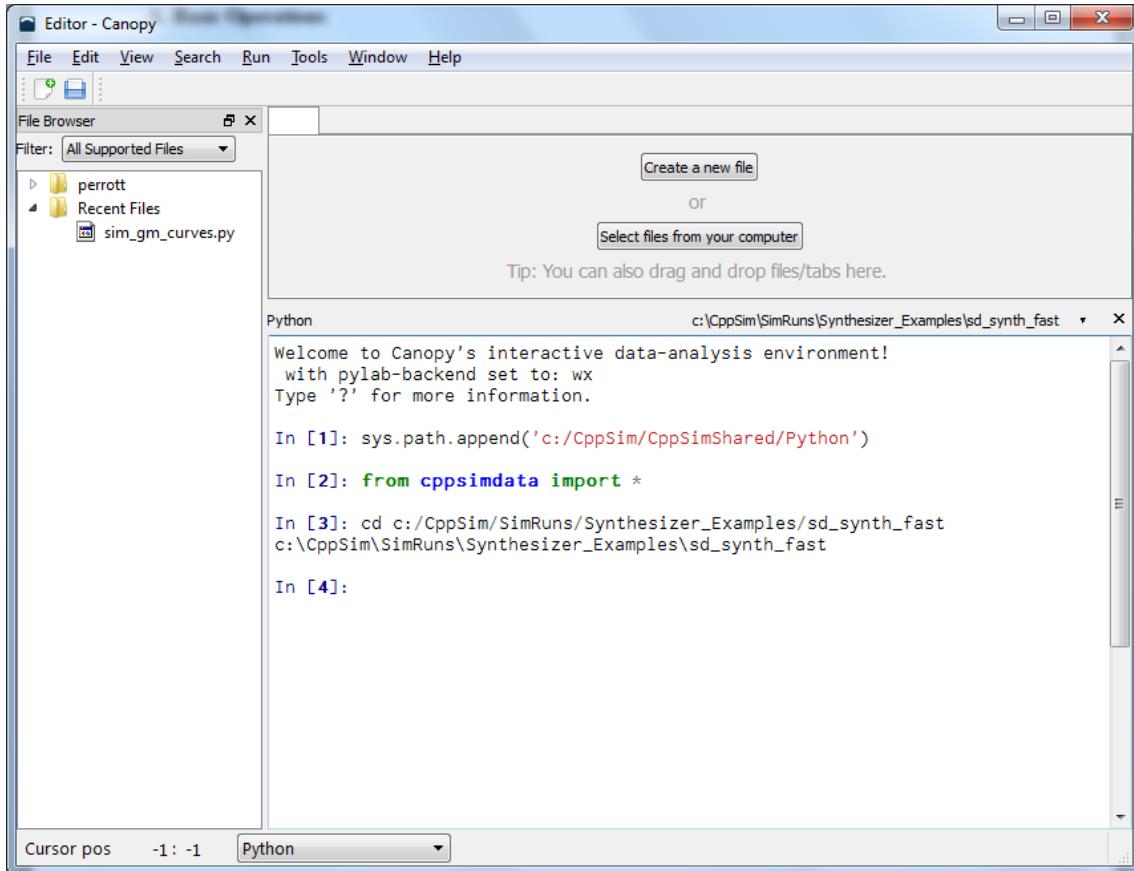
<https://www.enthought.com/products/epd/free/>

Note that for Windows platforms, you should download the 32-bit version of Canopy. For Mac platforms (assumed to be 64-bit), you should download the 64-bit version of Canopy. For Linux platforms, you should download the version that corresponds to your Linux operating system.

- As an example of running CppSim in Python, go to the simulation directory for the cell **sd\_synth\_fast** by typing the following in the Editor window in Canopy Python (which we will refer to as the Python prompt):

```
cd c:/CppSim/SimRuns/Synthesizer_Examples/sd_synth_fast
```

For the above command, you must substitute the proper path for CppSim in place of `c:/CppSim`. If you type `ls` at the Python prompt, you will see the many files produced by previously simulations. The simulation file, **test.par**, should and must be present in order for the steps that follow to work. Also, you must have imported the **cppsimdata** module as discussed above. The Canopy editor window below summarizes these operations:



- Once you are in the above directory, type

**cppsim()**

at the Python prompt – this will run CppSim by default on the **test.par** file located in the current directory. The **cppsim()** script will use the current directory information to determine the name of the cell and library (the current directory is the cell name (i.e., **sd\_synth\_fast**), and the next directory up is the library name (i.e., **Synthesizer\_Examples**)) and then use this information to automatically netlist the Sue2 cell and then run the simulation. The Canopy editor window displays the result of running **cppsim()** as shown below:

```

g++ -c test.cpp
g++ -o test test.o cppsim_classes.o -lm
export LD_LIBRARY_PATH=../../test
..... 0% completed .....
Warning in Vco.inp: divide_val is too small for the given sample rate!
in this case, divide_val = '2', and should be >= '18'
-> setting divide val to '18' whenever it is too small
..... 10% completed .....
..... 20% completed .....
..... 30% completed .....
..... 40% completed .....
..... 50% completed .....
..... 60% completed .....
..... 70% completed .....
..... 80% completed .....
..... 90% completed .....

Simulation Completed for Alter Run 0
*****

```

Note that if one desires to run CppSim on a different simulation file, such as **test2.par** for instance, then type the following command at the Python prompt instead of the above:

**cppsim('test2.par')**

- Once the run has completed, load the signals in file **test.tr0** into Python by typing

**data = CppSimData('test.tr0')**

You can then view the signal names contained within this file by typing

**data.lsSig()**

The Canopy editor window displays the result of running these commands as shown below:

The screenshot shows the Canopy Python editor interface. On the left, a 'File Browser' panel lists 'perrott' and 'Recent Files' with 'sim\_gm\_curves.py'. The main window has tabs for 'Python' and 'c:\CppSim\SimRuns\Synthesizer\_Examples\sd\_synth\_fast'. The Python tab displays a simulation log:

```

in this case, divide_val = '2', and should be >= '18'
-> setting divide val to '18' whenever it is too small
..... 10% completed .....
..... 20% completed .....
..... 30% completed .....
..... 40% completed .....
..... 50% completed .....
..... 60% completed .....
..... 70% completed .....
..... 80% completed .....
..... 90% completed .....

Simulation Completed for Alter Run 0
*****
In [5]: data = CppSimData('test.tr0')
In [6]: data.lssig()
Out[6]: ['TIME', 'div_val', 'out', 'pdfout', 'ref', 'sd_in', 'vin',
'xi12.xor_out']
In [7]: |

```

The status bar at the bottom shows 'Cursor pos -1: -1' and 'Python'.

- The signals **sd\_in** and **vin** are loaded into corresponding Python Numpy arrays as follows:

```

sd_in = data.evalsig('sd_in')
vin = data.evalsig('vin')

```

One can then perform post-processing or plotting of the above signals in Python as desired. As an example, one can plot the **vin** signal by using the following commands:

```

from pylab import *
plot(vin)

```

These commands are also shown in the Canopy editor window below:

**Editor - Canopy**

File Edit View Search Run Tools Window Help

File Browser Filter: All Supported Files

perrott Recent Files sim\_gm\_curves.py

Create a new file  
or  
Select files from your computer  
Tip: You can also drag and drop files/tabs here.

Python c:\CppSim\SimRuns\Synthesizer\_Examples\sd\_synth\_fast

```
..... 80% completed .....
..... 90% completed .....

Simulation Completed for Alter Run 0

***** CppSim run completed! *****

In [5]: data = CppSimData('test.tr0')

In [6]: data.lssig()
Out[6]: ['TIME', 'div_val', 'out', 'pf dout', 'ref', 'sd_in', 'vin',
'xi12.xor_out']

In [7]: sd_in = data.evalsig('sd_in')

In [8]: vin = data.evalsig('vin')

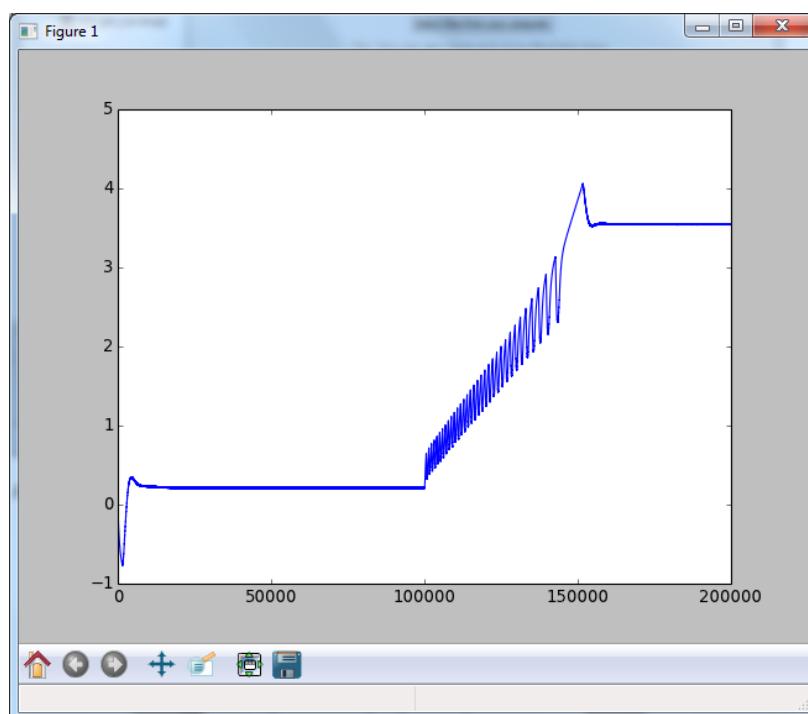
In [9]: from pylab import *

In [10]: plot(vin)
Out[10]: []

In [11]:
```

Cursor pos -1: -1 Python

Also, the resulting plot is shown below:



## **Using Matlab with CppSim**

To use CppSim within Matlab, users simply need to add the Hspice Toolbox commands (which come with the standard CppSim installation) to the Matlab path. This operation is performed by typing

```
addpath('c:/CppSim/CppSimShared/HspiceToolbox')
```

at the Matlab prompt, where c:/CppSim should be replaced by the actual path you chose for CppSim during the installation.

Three types of CppSim operations are supported within Matlab – running CppSim simulations, creating Matlab mex functions, and creating Simulink S-functions.

### **A. Running CppSim Simulations in Matlab**

- As an example of running CppSim in Matlab, go to the simulation directory for the cell **sd\_synth\_fast** by typing (in Matlab):

```
cd c:/CppSim/SimRuns/Synthesizer_Examples/sd_synth_fast
```

Again, you must substitute the proper path that you chose for CppSim in place of c:/CppSim. If you type **ls** at the Matlab prompt, you will see the many files produced by previously running simulations within CppSimView. The simulation file, **test.par**, should and must be present in order for the steps that follow to work.

Once you are in the above directory, type

```
cppsim
```

at the Matlab prompt – this will run CppSim by default on the **test.par** file located in the current directory. The **cppsim** script will use the current directory information to determine the name of the cell and library (the current directory is the cell name (i.e., **sd\_synth\_fast**), and the next directory up is the library name (i.e., **Synthesizer\_Examples**)) and then use this information to automatically netlist the Sue2 cell and then run the simulation. If one desires to run CppSim on a different simulation file, such as **test2.par** for instance, then type the following command at the Matlab prompt instead of the above:

```
cppsim test2.par
```

Once the run has completed, load the signals in file **test.tr0** into Matlab by typing

```
x = loadsig_cppsim('test.tr0');
```

You can then view the signals contained within this file by typing

```
lssig(x);
```

Finally, plot the signals **sd\_in** and **vin** by typing

```
plotsig(x,'sd_in;vin');
```

- A key advantage of using Matlab is the greatly increased flexibility it offers for doing post-processing. In particular, one can create Matlab scripts to load in CppSim output files (i.e., test.tr0, test.tr1, ...) and then perform sophisticated processing on the signals they contain. To do so, one needs to turn signals embedded within CppSim output files into Matlab signals. This is achieved for signal **vin** in the above example by typing

```
vin = evalsig(x,'vin');
```

in Matlab. The above operation allows one to now directly access the data values of **vin** in Matlab. For instance, to view the first ten samples of **vin**, simply type

```
vin(1:10)
```

in Matlab.

It is worthwhile to examine the Hspice Toolbox manual (which is provided as the PDF document: c:/CppSim/CppSimShared/HspiceToolbox/document.pdf) for more information on the Matlab commands it offers related to viewing and post-processing. Also, a few post-processing functions are available in c:/CppSim/CppSimShared/MatlabCode that perform similar operations to the **plot\_pll\_phasenoise(...)** and **plot\_pll\_jitter(...)** functions in CppSimView.

## B. Creating Matlab Mex Functions

Assuming you have created a **mex\_prototype:** statement within the **Sim File** (i.e., test.par) of a cell, you can directly create and compile the mex function in Matlab.

- As an example of creating a mex function corresponding to a CppSim simulation in Matlab, go to the simulation directory for the cell **sd\_synth\_fast** by typing (in Matlab):

```
cd c:/CppSim/SimRuns/Synthesizer_Examples/sd_synth_fast
```

Note that you must substitute the proper path that you chose for CppSim in place of c:/CppSim.

Once you are in the above directory, type

```
cppsim2mex
```

at the Matlab prompt – this will run CppSim by default on the **test.par** file located in the current directory. The **cppsim2mex** script will use the current directory information to determine the name of the cell and library (the current directory is the cell name (i.e., **sd\_synth\_fast**), and the next directory up is the library name (i.e., **Synthesizer\_Examples**)) and then use this information to automatically netlist the Sue2 cell, create the mex code for the

CppSim simulation, and then compile it (assuming you have installed a C++ compiler on your computer). If one desires to run the **cppsim2mex** script on a different simulation file, such as **test2.par** for instance, then type the following command at the Matlab prompt instead of the above:

```
cppsim2mex test2.par
```

### C. Creating Simulink S-Functions

Assuming you have created a **simulink\_prototype:** statement within the **Sim File** (i.e., **test.par**) of a cell, you can directly create and compile the Simulink S-function in Matlab.

- As an example of creating an S-function corresponding to a CppSim simulation in Matlab, go to the simulation directory for the cell **sd\_synth\_fast\_simulink** by typing (in Matlab):

```
cd c:/CppSim/SimRuns/Synthesizer_Examples/sd_synth_fast_simulink
```

Note that you must substitute the proper path that you chose for CppSim in place of **c:/CppSim**.

Once you are in the above directory, type

```
cppsim2simulink
```

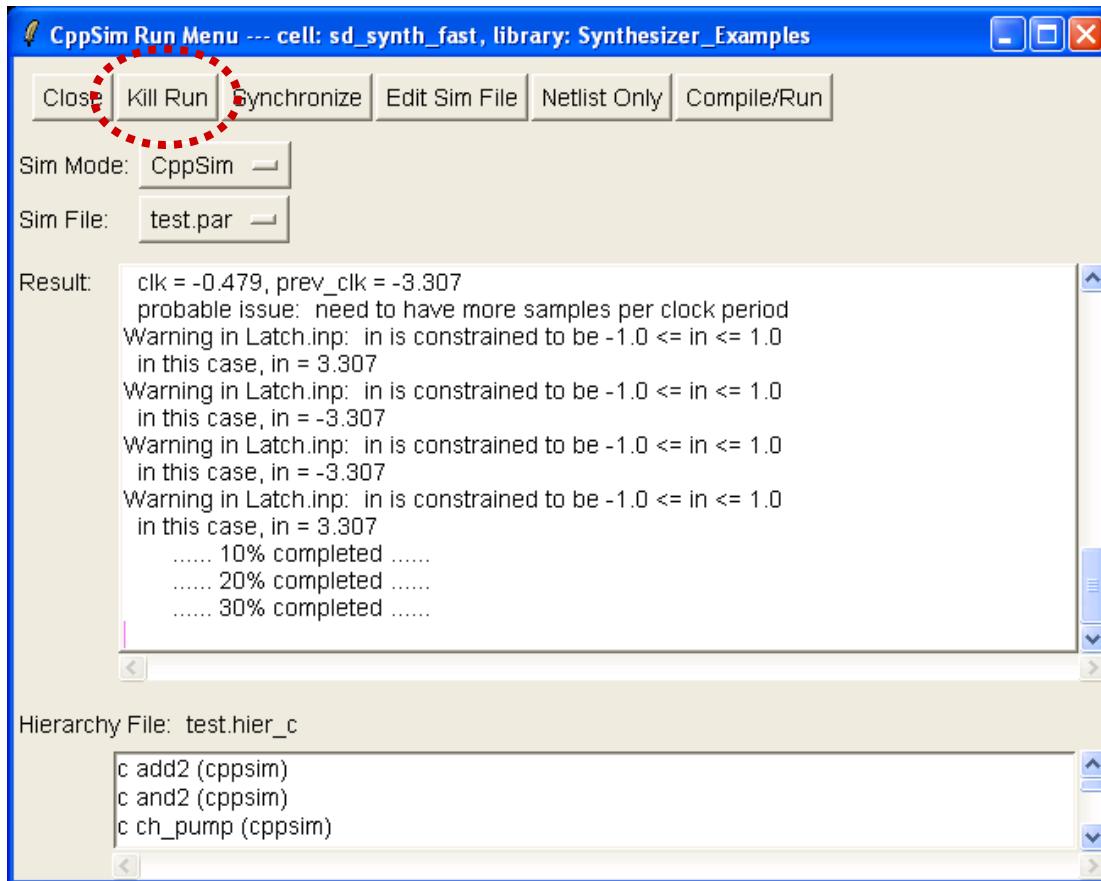
at the Matlab prompt – this will run CppSim by default on the **test.par** file located in the current directory. The **cppsim2simulink** script will use the current directory information to determine the name of the cell and library (the current directory is the cell name (i.e., **sd\_synth\_fast\_simulink**), and the next directory up is the library name (i.e., **Synthesizer\_Examples**)) and then use this information to automatically netlist the Sue2 cell, create the S-function code for the CppSim simulation, and then compile it (assuming you have installed a C++ compiler on your computer). If one desires to run the **cppsim2simulink** script on a different simulation file, such as **test2.par** for instance, then type the following command at the Matlab prompt instead of the above:

```
cppsim2simulink test2.par
```

### Killing Runaway CppSim Simulations

If you ever wish to kill an ongoing CppSim simulation, there are two ways to do it.

- If you have run the simulation from the CppSim Run Menu window, simply click on the **Kill Run** button to end it as shown in the figure below.



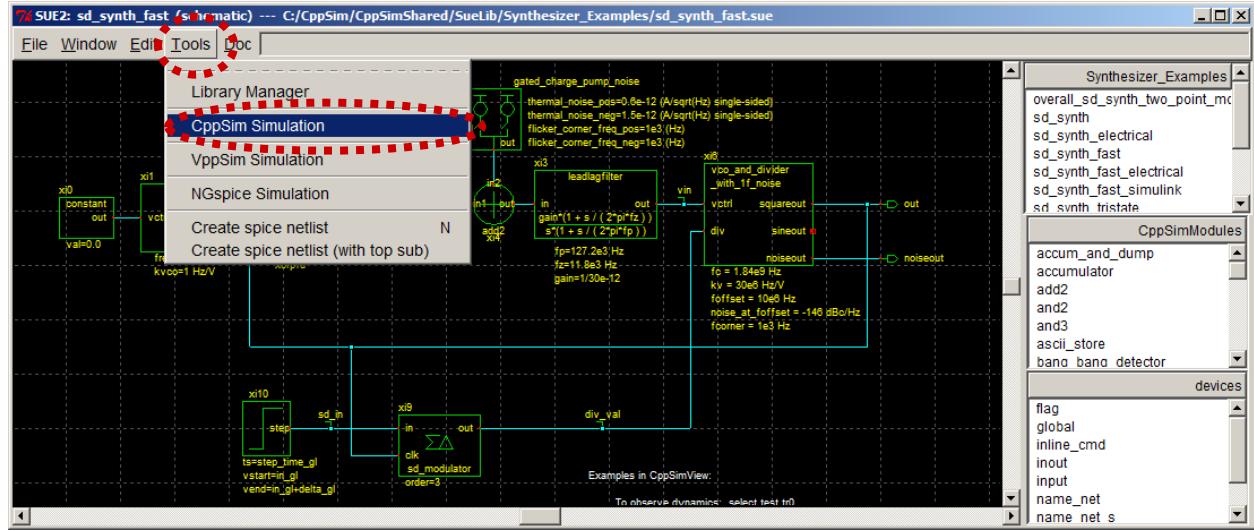
- An alternative method is to directly kill the process:
  - In **Windows**: push **Ctrl-Alt-Delete** in Windows, left-click on the **Processes** tab of the form that comes up, and then left-click on **test.exe** (or the appropriate name if the simulation file was not test.par – i.e., **test\_new.par** -> **test\_new.exe**). You can search among process by repeatedly hitting the first letter of the process you seek (i.e., press the **t** key several times in this case). Once you have selected **test.exe** (or the appropriate process), then left-click on **End Process** and then click **Yes** in the dialog box that pops up.
  - In **Mac OS X**: run the **Activity Monitor** from the **/Applications/Utilities** folder. Select the application with process name **test** (or the appropriate name if the simulation file was not test.par – i.e., **test\_new.par** -> **test\_new**), and then click on the **Quit Process** button.
  - In **Linux**: run the **top** command from the Linux shell prompt. Find the application with process name **test** (or the appropriate name if the simulation file was not test.par – i.e., **test\_new.par** -> **test\_new**), and then type **kill -9 test**.

## More Details on CppSimView

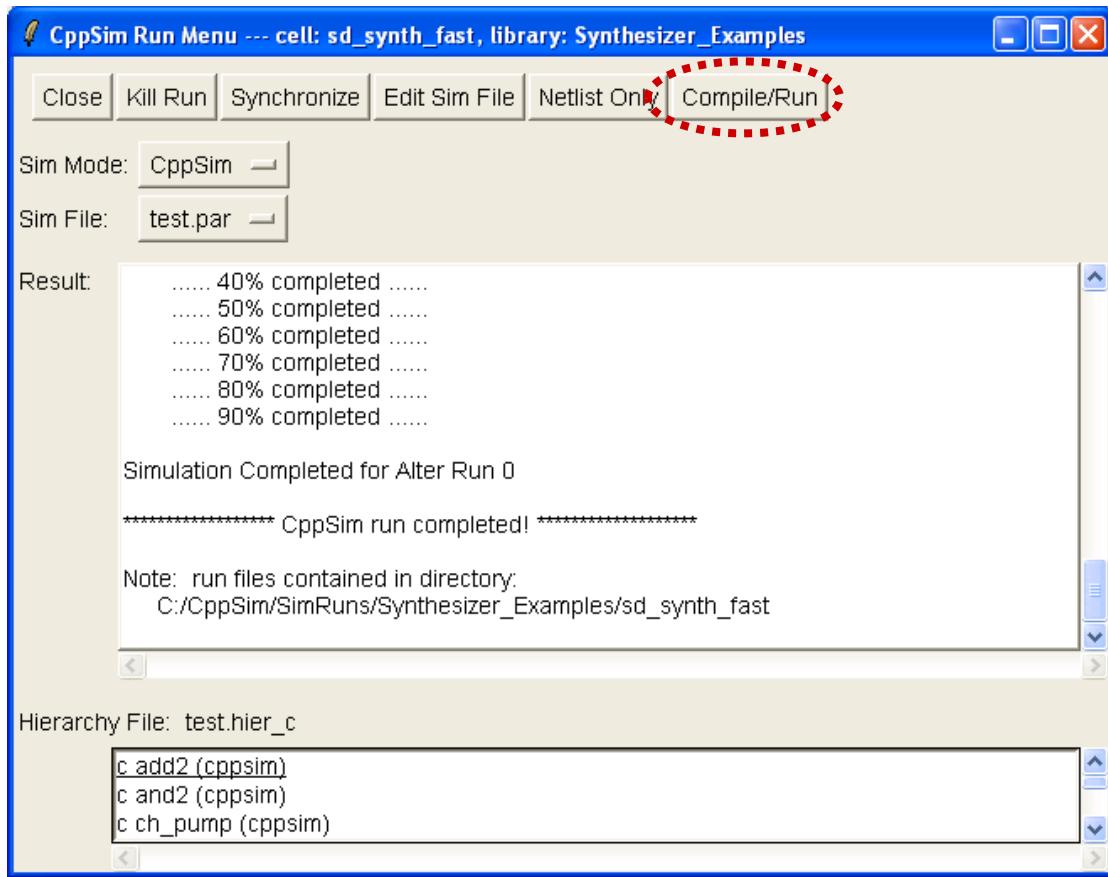
In this section, we will examine more details related to using CppSimView to view simulation results from CppSim. We will do so through example.

### A. Preliminary Setup

- Within Sue2, open up the **sd\_synth\_fast** schematic within the **Synthesizer\_Examples** library, and then select the **CppSim Simulation** item from the **Tools** menu item as shown in the figure below.

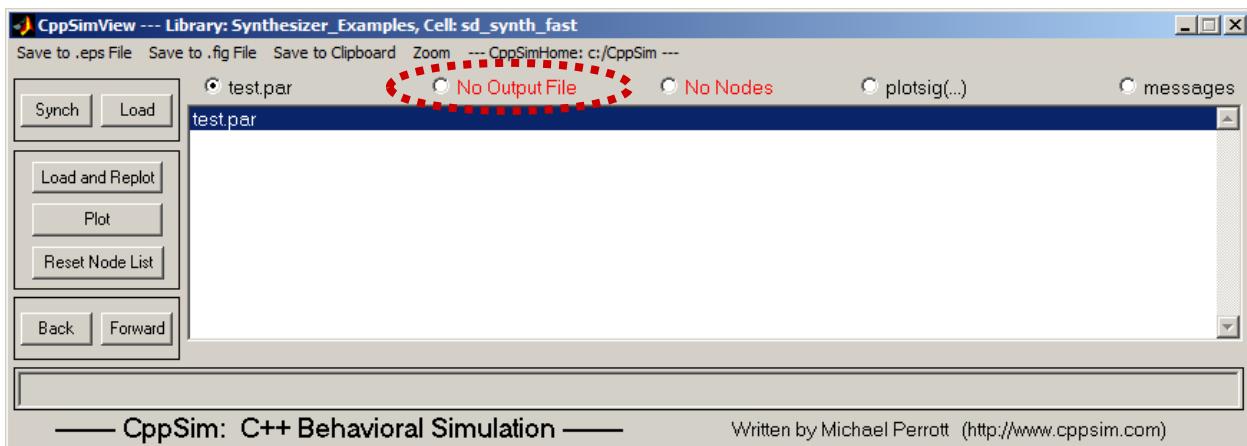


- In the **CppSim Run Menu** that comes up, click on the **Compile/Run** button to run the CppSim simulation, as shown in the figure below.



- Start **CppSimView** by performing the following action:
  - Windows:** double-click on the **CppSimView** icon on the Windows Desktop.
  - Mac OS X:** double-click on the **CppSimView** app in the Applications folder
  - Linux:** at the Linux prompt, run the command **cppsimeview**

You should see a window as shown below.



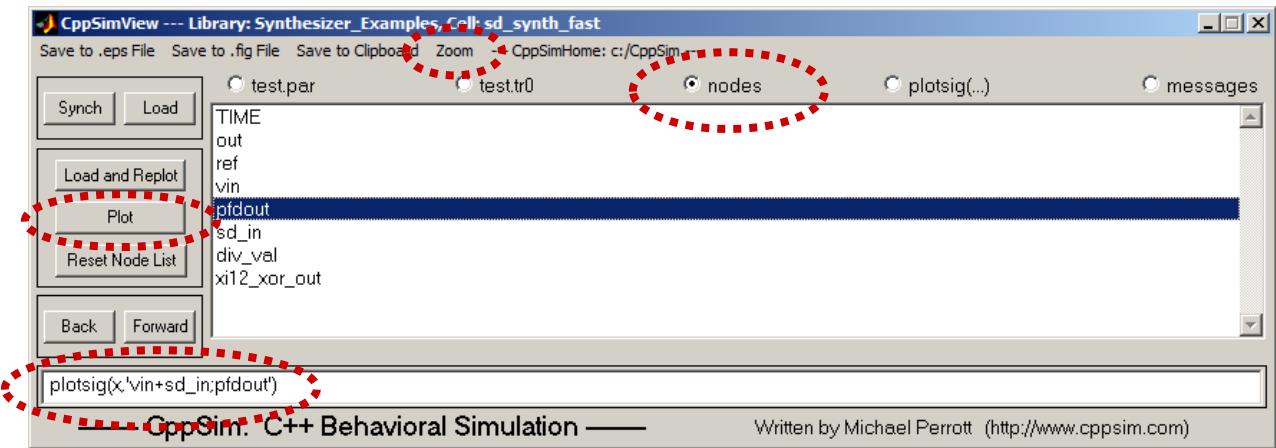
## B. Selecting an Output File

- Click on the **output file radio button** (see circled button above) to select an output file produced during the CppSim simulation. In this case, let us choose **test.tr0**. In practice,

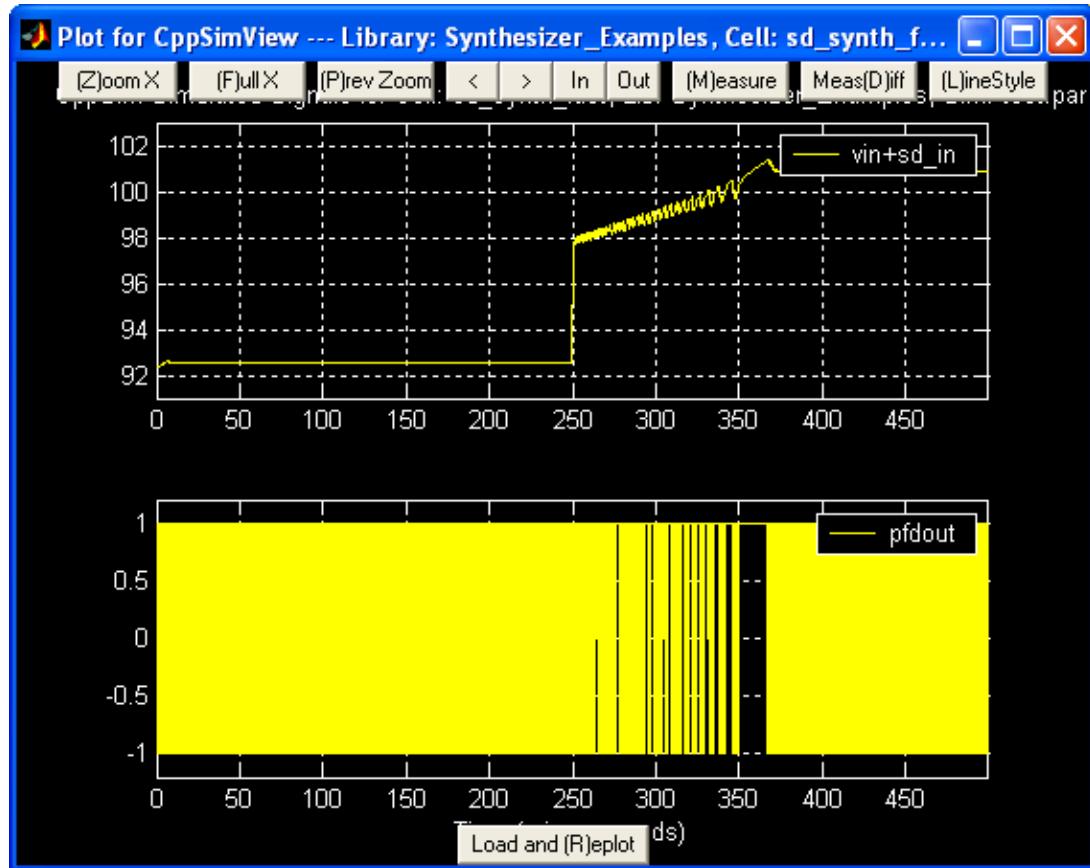
multiple output files are supported, which can be useful in case fancy triggering and decimation routines are desired. See the **test.par** file for this example to see how triggering was used to create **test\_noise.tr0**, and refer to the CppSim Reference Manual (i.e., **cppsimdoc.pdf**) for further details.

## C. Basic Plotting and Zooming Methods

- Click on the **nodes** radio button as circled below. Then enter an expression into the command line and then click on the **Plot** button to produce a plot of the specified signals. In the example below, we have specified that we would like to plot signals **vin** and **sd\_in** added together in one subplot and **pfdout** in the other subplot. Please refer to the Hspice Toolbox for Matlab/Octave manual for a description of the notation used here.



- Now click on the **Zoom** button (circled in the above figure) to bring up zoom controls on the plot as shown below. Each of the zoom keys has a respective hot key indicated by the parenthesis in each word. For instance, pressing **z** (upper or lowercase), allows one to zoom into a subportion of the x-axis of the current plot. Exceptions to this rule are the zoom **In** and **Out** buttons, whose hotkeys are the up and down arrow keys. Also, the left and right pan keys, < and >, are hot-keyed to the left and right arrow keys.
  - Note that no Y zoom functions are currently implemented. However, they are generally unnecessary since the Y-axis gets adjusted automatically during X-zoom operations.

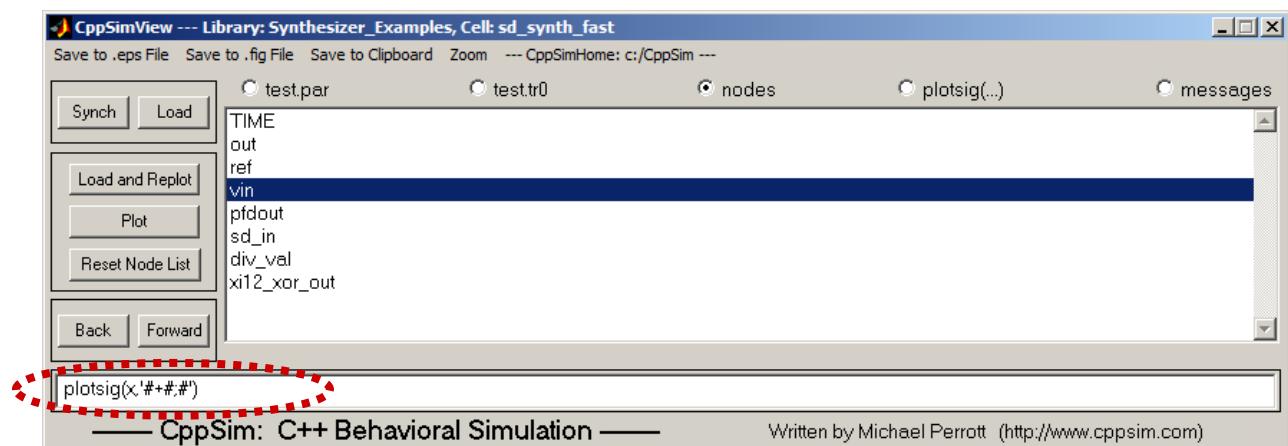


- Press the **m** key to begin measuring a signal. Press the left mouse button repeatedly until you are satisfied with the point selected. Then press the right mouse button to complete the measurement.
- Press the **d** key to begin a difference measurement. Press the left mouse button repeatedly until you are satisfied with the first point to be selected. Then press the right mouse button repeatedly until you are satisfied with the second point to be selected. Press the left mouse button to complete the measurement.
  - Note that you can combine the **Measure** and **MeasDiff** commands. First, perform a measurement command by pressing the **m** key as described above. Upon completion of this command, press the **d** key to begin a **MeasDiff** command. However, instead of pressing the left button, press the right one. The first point will remain that selected by the **Measure** command, and the second can now be set where desired. Press the left mouse button to complete the **MeasDiff** operation. The advantage offered by this option is that you can zoom into a particular part of the waveform and select an initial point using the **Measure** command. You can then zoom into a different portion of the waveform, and then left-click on **MeasDiff** to determine the difference from the last point to a new point in the current zoom location by using this technique.
- Press the **l** key (i.e. lowercase L) to display the actual sample values from the simulation (as indicated by circles). Press the **l** key again to return to solid lines for the plot.
- Press the **p** key to return to the previous zoom value (i.e., the last achieved through use of the **Zoom X** button). Note that if you just used the **Zoom X** function without doing any other zoom or pan operations, you will see no change in the plot.

- Press the **Zoom** button again on the CppSimView main window (as circled in the figure above) to remove the zoom buttons from the plot window.

## D. Advanced Plotting Methods

- There are actually five ways to perform plotting with CppSimView.
  - The first is to left-click on the **Plot** button once an expression is entered into the bottom command line (as demonstrated above).
  - The second is to double-click on a node in the listbox (such as **vin**, as shown in the figure below). The plot expression currently selected on the plot radio button (i.e., **plotsig(...)** in the figure below) will then be filled with the selected node name (i.e., **vin** in this case, so that we obtain **plotsig(x,'vin')** in the command line) and the expression is plotted. If you continue to double-click on nodes, additional subplots are created with the new signals. To reset the number of subplots to one, press the **Reset Node List** button.
  - The third is to enter a plot expression directly into the command line and then press the **Enter** key to produce the corresponding plot. One can also modify an existing expression created, for instance, by the second method. The latter method often proves convenient – simply double-click on the desired signals to produce various subplots, and then modify the resulting command line expression to implement functions on the various signals or to position them on the same subplot (using a comma separator rather than a semicolon).
  - The fourth is to enter a plot expression in the command line, but insert # characters into the expression where you would like to have signal names. Once you have completed the expression, double-click on node names and observe that the # characters are substituted from left to right with the signal names. Once the last # character has been filled in, a plot of the expression will be produced.

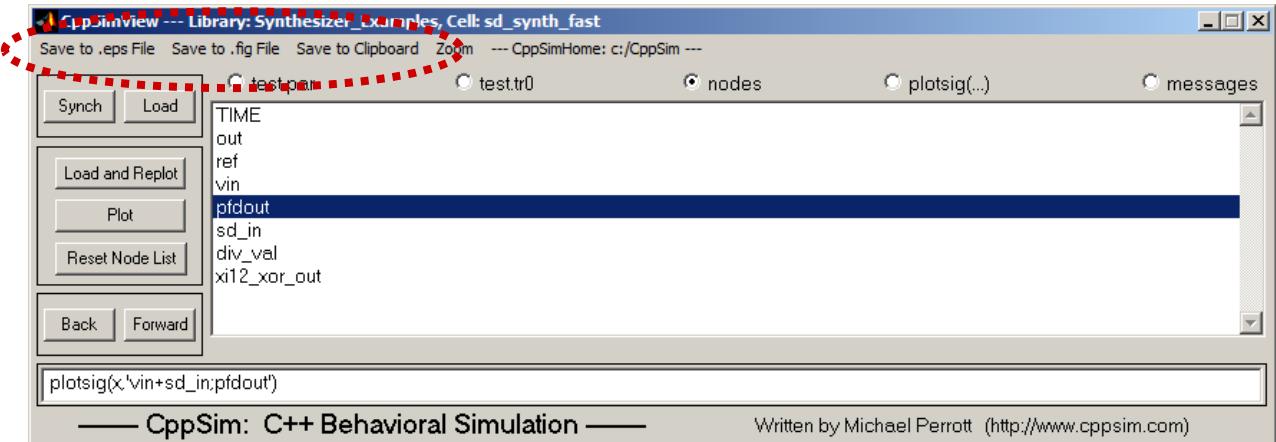


- The fifth method is to use the **Back** and **Forward** buttons to scroll through a history of previous plotting expressions. Once a desired plotting expression is encountered, left-click on the **Plot** button to replot it or perform alterations of the expression in the command line as desired and then press the **Enter** key. Note that the history commands are specific to the selected simulation file and output file (i.e., **test.par** and **test.tr0**, for example, in the figure below). The history keeps track of the last 400

commands used on a given cellview (i.e., for **sd\_synth\_fast**, as an example), and it is shared among the various simulation and output files for that cellview.

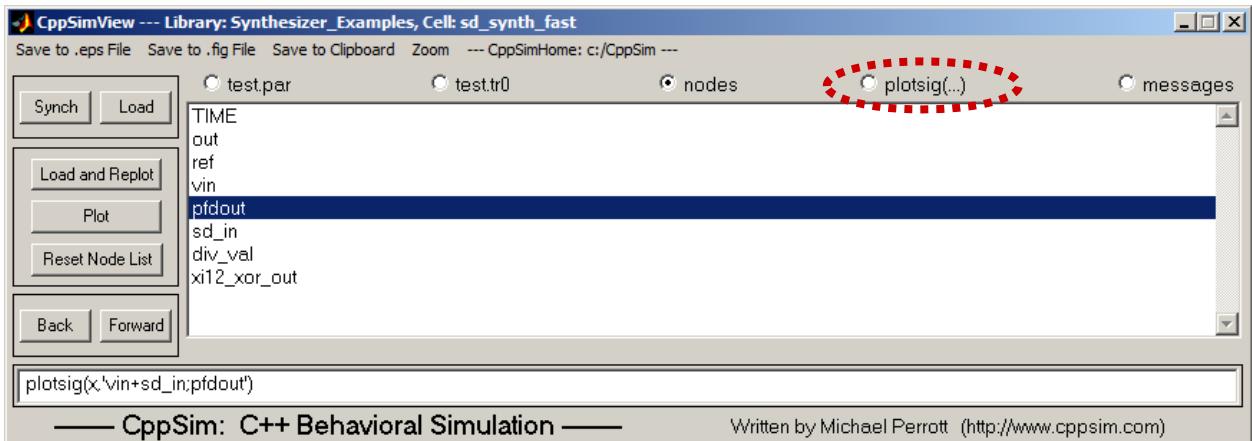
## E. Saving Plots to EPS files, FIG files, or the Windows Clipboard

- To save plots to an eps file, fig file, or to the clipboard, press either **Save to .eps File**, **Save to .fig File**, or **Save to Clipboard**, respectively, in the CppSimView main window. When saving to the clipboard, the plots can then be pasted into other Windows applications such as Word or PowerPoint.



## F. Choosing Different Plotting Functions

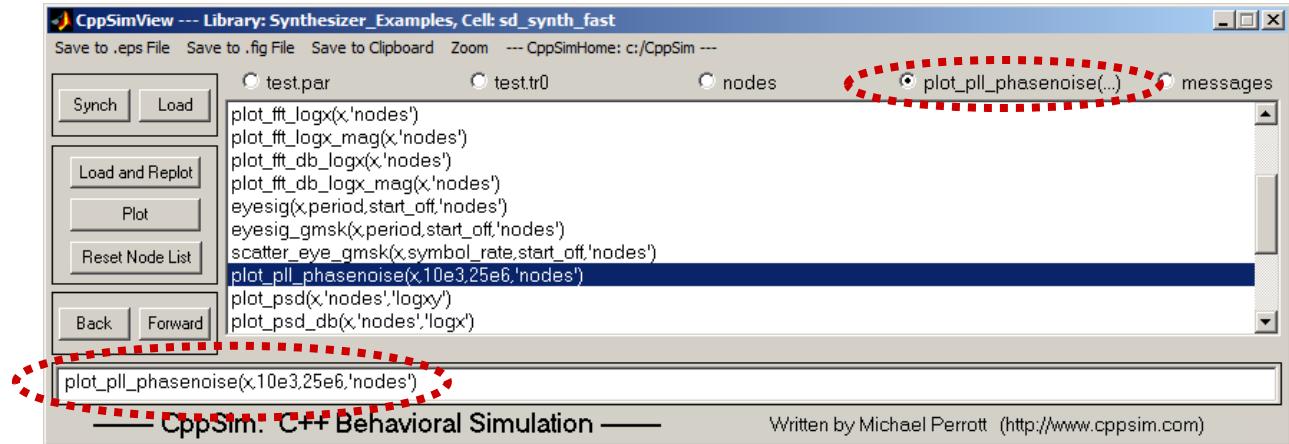
- As mentioned above, double clicking on signal names allows plotting to occur by filling in the current plot expression. To choose a different plot expression, press the plot function radio button (i.e., **plotsig(..)** in the figure below).



As you select different expressions in the listbox, their name will appear in the radio button label, with the exception of non-plotting functions such as **plot\_title('name')**, etc.

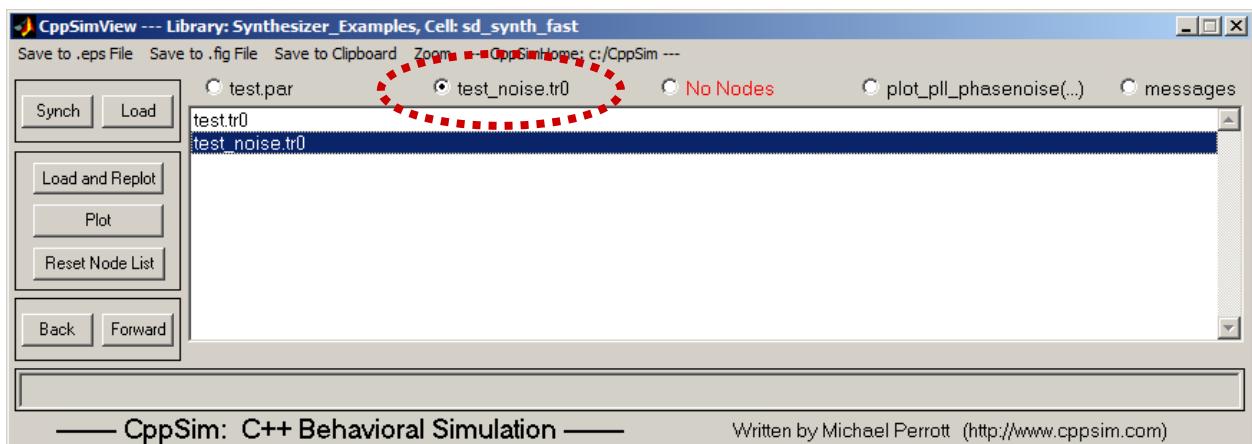
- The non-plotting functions can be used to label current plots. Simply select one of these non-plotting functions, then click on the **nodes** radio button, and then hit **Enter** to execute the function.

- Some of the plotting functions, such as `plot_pll_phasenoise(...)`, require additional parameters beyond just signal names (such as `f_low` and `f_high` in this case). Alter the plot expression in the command line so that these parameters are replaced with numbers (i.e, substitute, as an example, `10e3` for `f_low` and `25e6` for `f_high`). Upon pressing **Enter**, the expression in the listbox will contain the updated information. If one then clicks on the **nodes** radio button and begins double-clicking on signal names, the new expression will be appropriately executed.



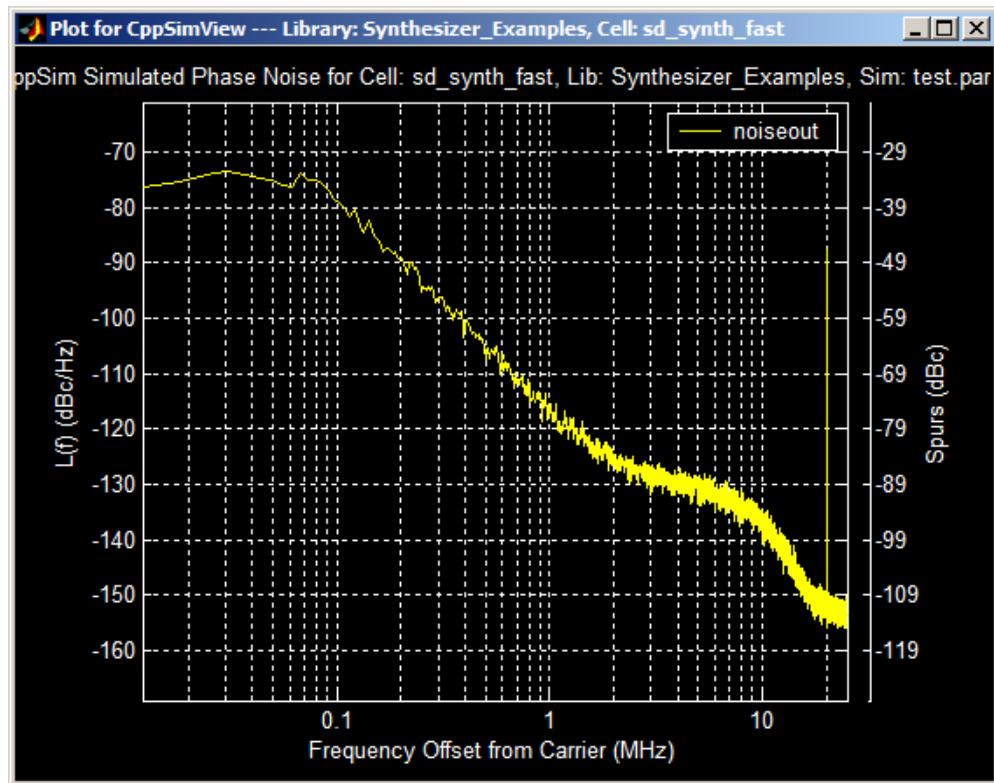
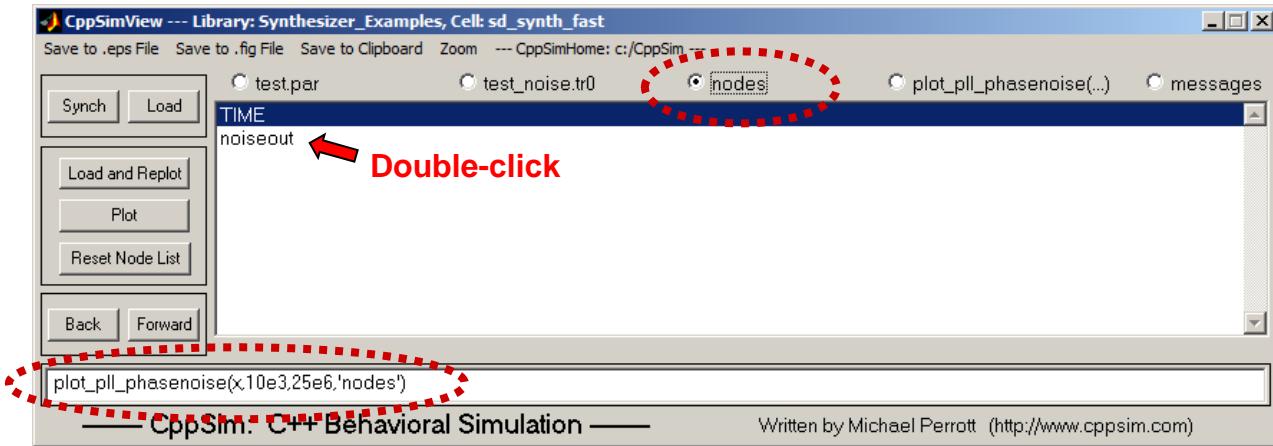
## G. Using the `plot_pll_phasenoise(...)` Plotting Function

- Continuing with the above example, click on the **test.tr0** radio button, and then choose **test\_noise.tr0** as shown below. If you examine the **test.par** file by pressing the **Edit Sim File** button, you'll notice that `test_noise.tr0` stores data from time sample `200e3` to the end of the simulation run – avoiding the first `200e3` samples allows transient effects to be removed from the noise analysis to follow.



- Now click on the **No Nodes** radio button to load in the signals from the **test\_noise.tr0** file. As shown below, only two signals were probed in this file: **TIME** and **noiseout**. Notice that the `plot_pll_phasenoise(x,10e3,25e6,'nodes')` function now appears in the command line. Double-clicking on **noiseout** causes the '**nodes**' entry in the function to be replaced with **noiseout** and for the resulting expression to be plotted. The

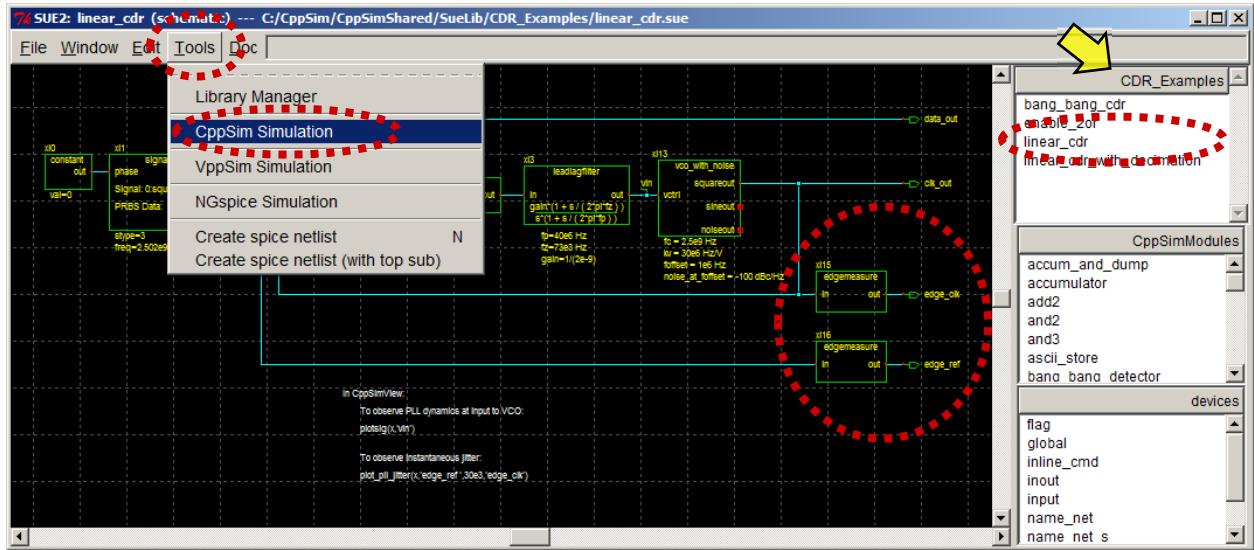
resulting phase noise plot for the synthesizer is shown below – the left axis indicates the value of the phase noise in dBc/Hz, and the right axis indicates spur levels in dBc. The left and right axis display different scales due to the fact that the resolution bandwidth is not 1 Hz.



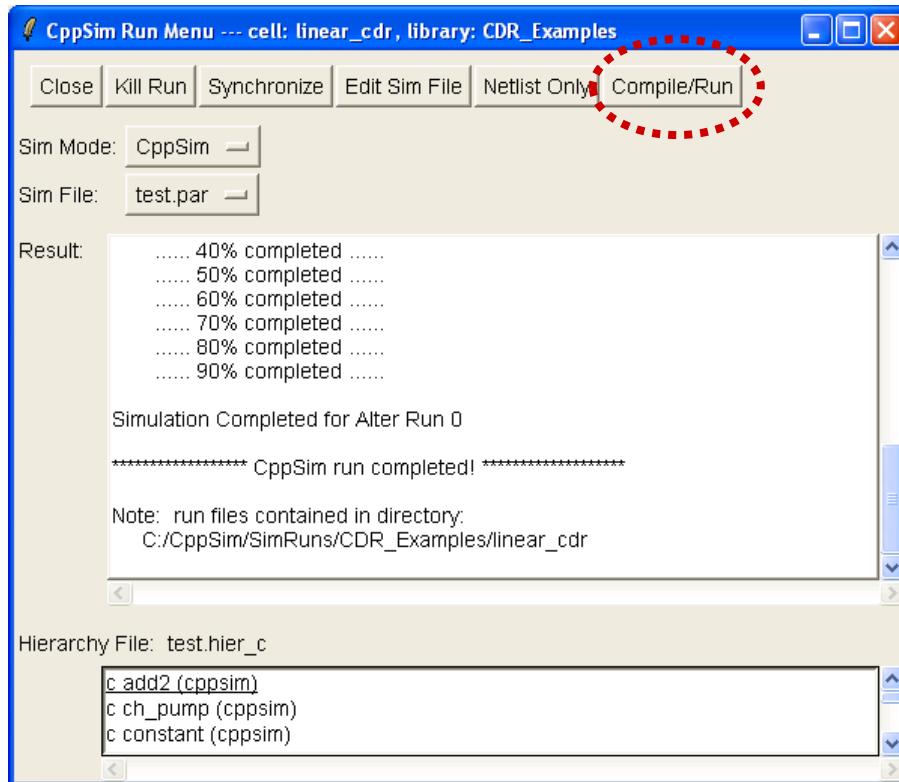
## H. Using the `plot_pll_jitter(...)` Plotting Function

- Now choose the **CDR\_Examples** library in the Sue2 schematics listbox. Click on `linear_cdr`, which brings up the schematic of a clock and data recovery circuit that uses a Hogge detector for phase detection as shown below. As shown circled in the figure, the source clock and output clock are passed through `edgemeasure` blocks whose outputs are `edge_ref` and

`edge_clk`, respectively. The `edgemeasure` blocks produce signals that are compatible with the `plot_pll_jitter(...)` command now described.

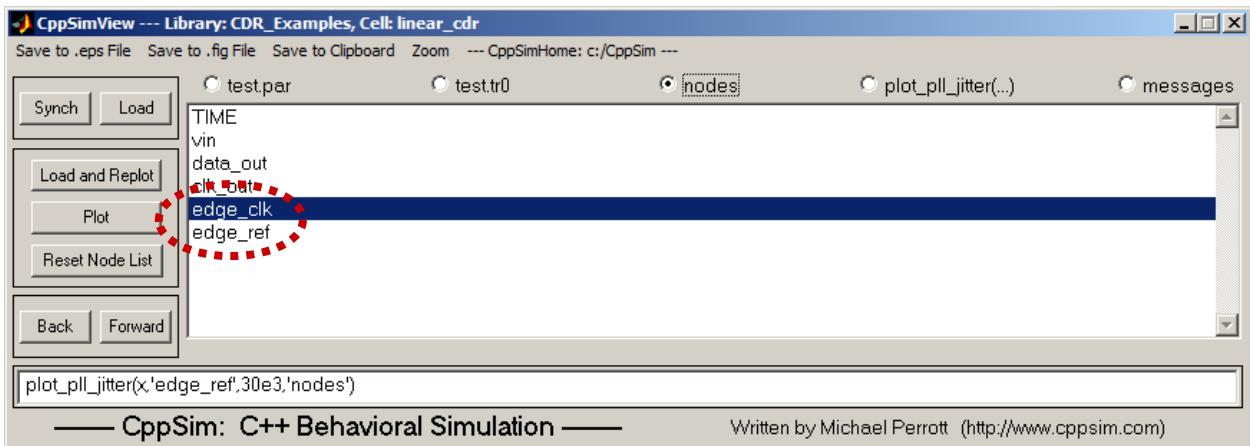


- Now click on the **CppSim Simulation** menu item as shown above, such that the **CppSim Run Menu** window appears as shown below. Click on the **Compile/Run** button as circled below.

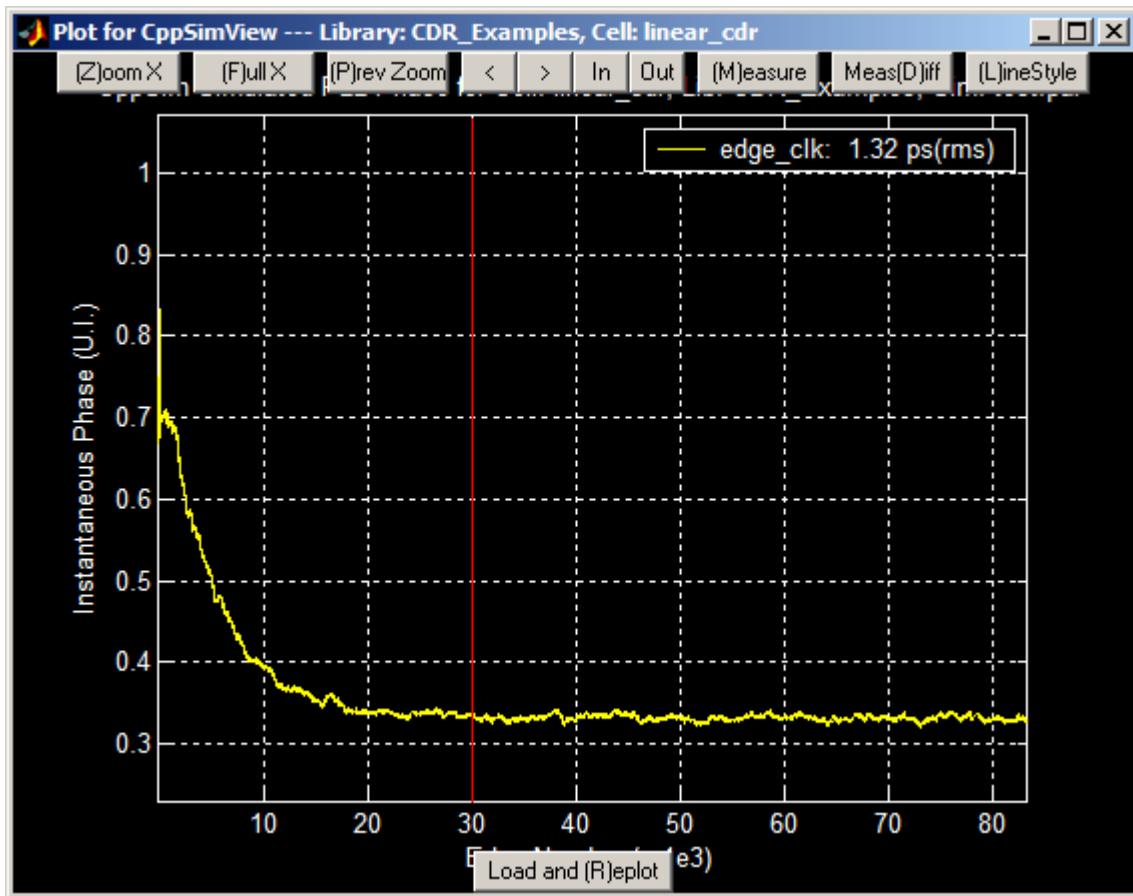


- Now click on **Synch** button in the CppSimView window. The top of the CppSimView window should indicate that the current cell has changed to `linear_cdr`. Then click on the **No Output File** radio button to select `test.tr0`, followed by the **No Nodes** radio button to load in the nodes of `test.tr0`. Now click on the **plotsig(...)** radio button and

choose the **plot\_pll\_jitter(...)** function from the list box. This function will appear in the command line – replace ‘**ref\_timing\_node**’ with ‘**edge\_ref**’, and replace **start\_edge** with **30e3**. Hit the **Enter** key to update the **plot\_pll\_jitter(...)** function in the listbox. Now click on the **nodes** radio button – the CppSimView window should appear as shown below. Double-click on **edge\_clk** in the listbox to fill in ‘nodes’ within the **plot\_pll\_jitter(...)** function and create the plot shown below.



- As shown below, the **plot\_pll\_jitter(...)** plot displays the instantaneous phase difference between the reference and output clock edges in unit intervals (UI) – note that one UI corresponds to  $2\pi$  radians in phase. The red line corresponds to the chosen value of **start\_edge**, and marks the region over which the steady-state jitter calculation occurs. In the example below, the jitter between the reference and output clock is calculated to be 1.3 ps (rms) for all edges to the right of the red line.



## More Details on Sue2

Sue2 provides a convenient graphical interface for creating and modifying CppSim systems, and is designed to have a similar look and feel as Cadence Composer so that IC designers can easily alternate between these tools as they iteratively perform system and circuit level design. A more complete manual is available for Sue2 as the PDF document: c:/CppSim/CppSimShared/Doc/sue\_manual.pdf, but we will cover enough of its operation here for users to get a good feel of this package.

Before we begin, there are two important things to keep in mind when you use Sue2:

- Always pay attention to the Help Message Window, which is to the right of the menu at the top of the main canvas, during command operations – it provides information for bindkeys activated while a given command is in effect
- To break out of any given command mode, hit the **Esc** key. This is very important to remember – if Sue2 ever seems to lock up, hit the **Esc** key! (The other reason Sue2 may appear to lock up is if an entry form was opened but not completed – in such case, be sure to find the entry form among the Windows applications and close it to continue with Sue2).

## A. Using Navigation and Edit Commands

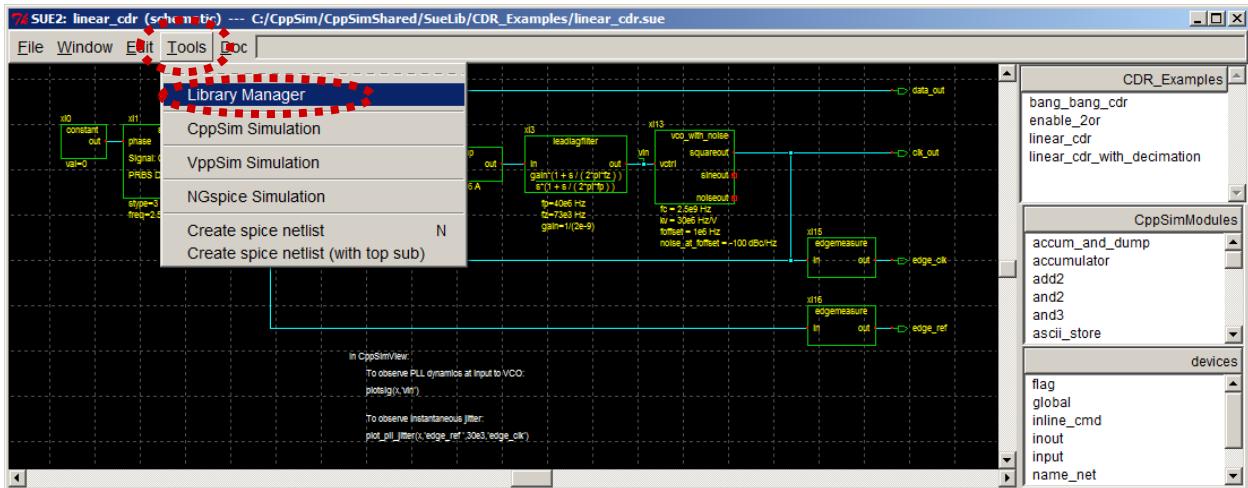
Sue2 allows its bind-keys to be changed according to user preference by editing of the file c:/CppSim/Sue2.suerc. That being said, the default values of common navigation and edit bind-keys are listed here.

- Sue2 navigation commands:
  - **f** – fit view to the window size
  - **z** – zoom in
  - **Z** – zoom out
  - Zooming can also be accomplished by pressing the right mouse button and dragging the mouse over the region to be zoomed into
  - Panning is done by either hitting the arrow keys or by holding the **Ctrl** key and then dragging the mouse while the left mouse button is held down.
  - **e** - descend into hierarchy of selected cell.
  - **Ctrl+e** – Return to higher level of hierarchy.
- Sue2 editing commands:
  - Modify the parameters of a cell within a schematic by double-clicking on the cell. A listbox will appear that displays the cell parameters and allows their modification.
  - Move cells by pressing and holding the left mouse button on the desired cell and then dragging the mouse.
  - Select multiple items by holding the left mouse button and dragging the mouse over the items to be selected. Additional items can be added to the current selection by holding the **shift** key and then progressively clicking the left mouse button on the items of interest.

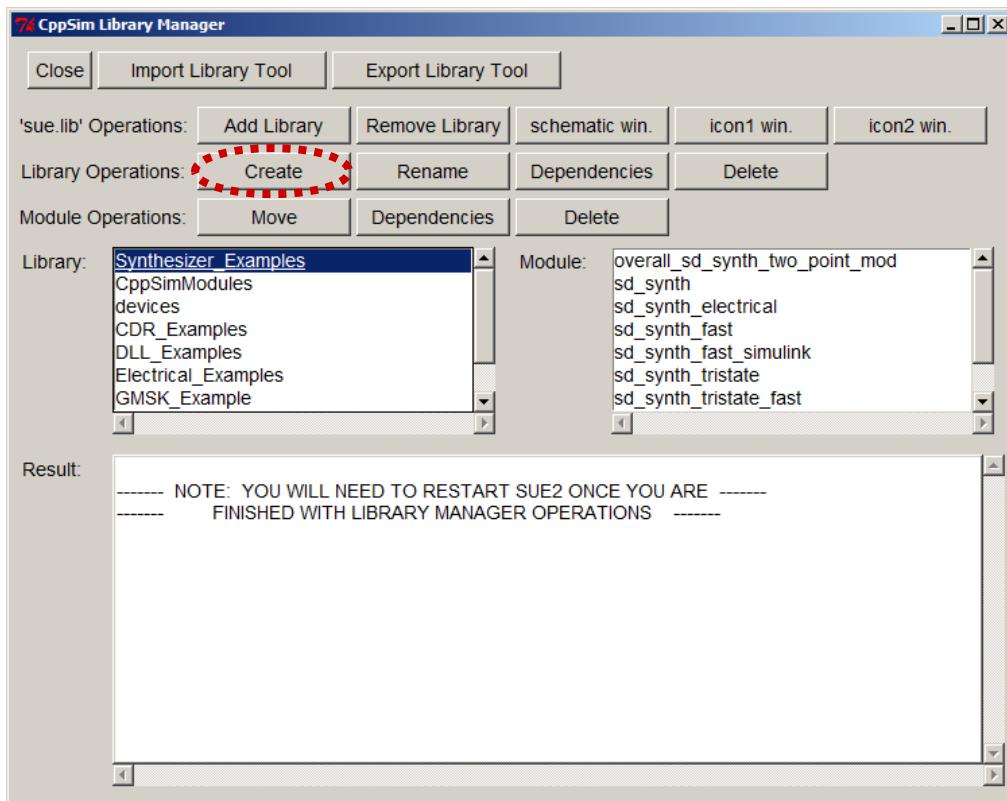
## B. Creating a New Schematic

Let us now walk through an example to see how to create a new Sue2 schematic. In this case, we will create a pseudo-random bit stream (PRBS), pass it into a lowpass filter, and then view the results both as time domain signals and in the form of an eye diagram.

- We will first create a new library called **PRBS\_Examples**
  - In Sue2, click on the **Library Manager** menu item under the **Tools** menu bar item as shown in the figure below.



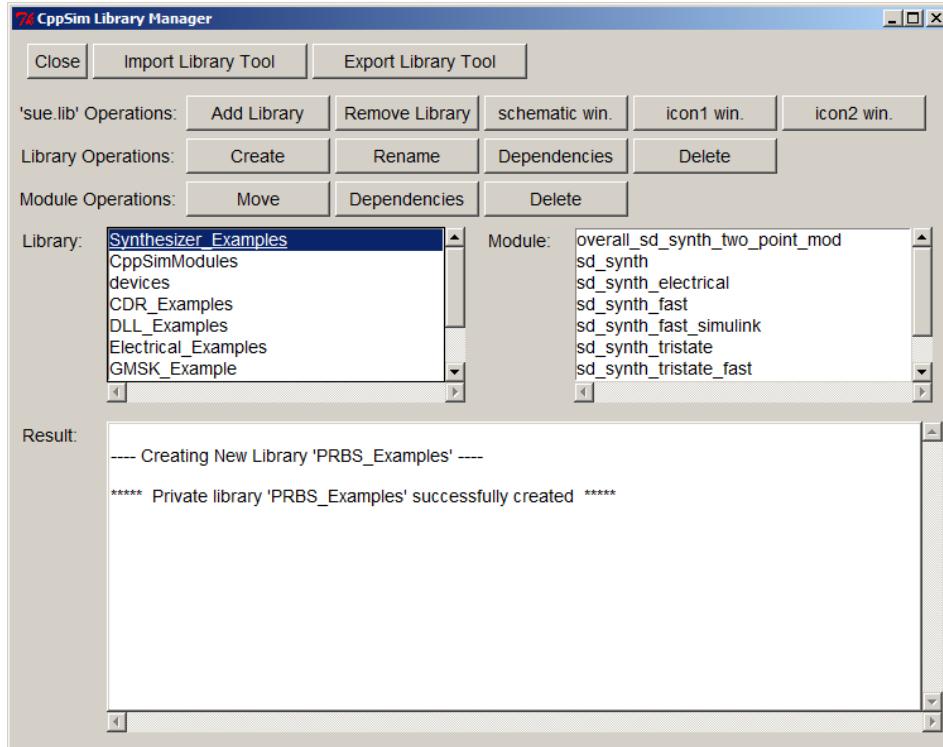
- In the **Library Manager** window that appears, click on **Create Library** as shown below.



- Choose the new library name as **PRBS\_Examples** and then press **OK**.

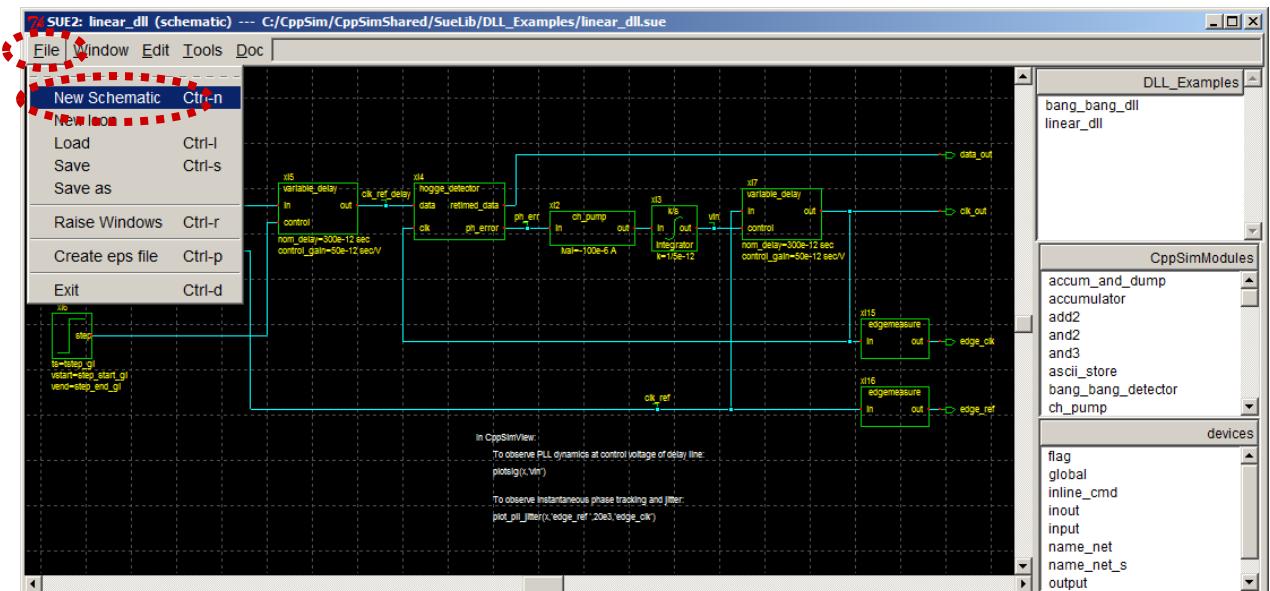


- You should see a confirmation window in the **CppSim Library Manager** window as shown below. You should then **Close** this window.

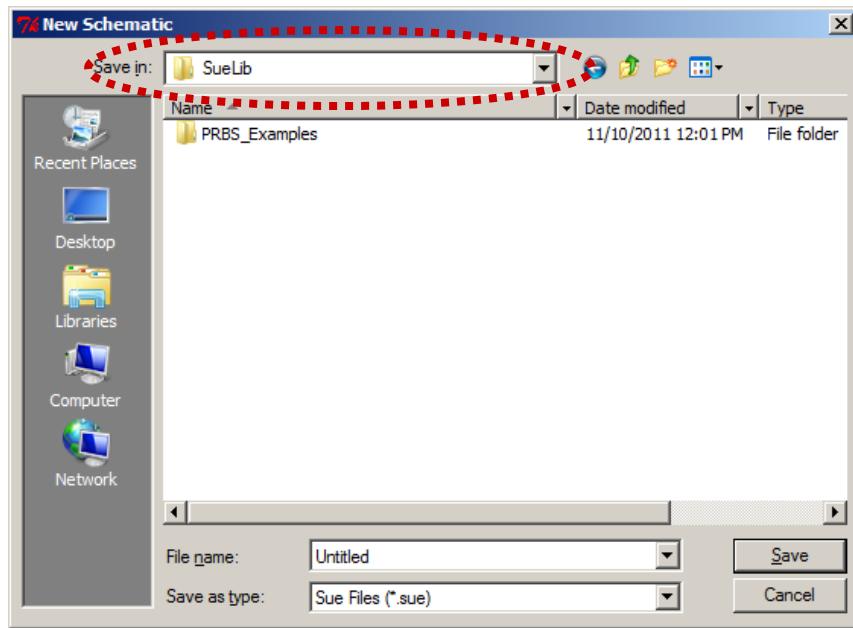


- In Sue2, create a new schematic cell as follows:

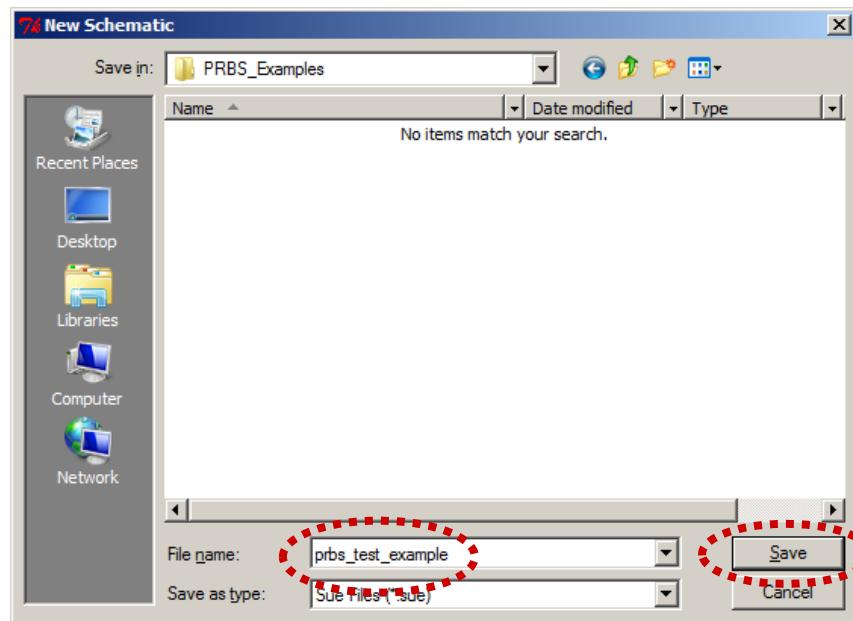
- Select **File -> New Schematic** as shown below.



- A **New Schematic** window opens as shown below. Within the **Save in:** section, select the current path to be **c:/CppSim/SueLib**. You should see the **PRBS\_Examples** directory as shown in the figure below.

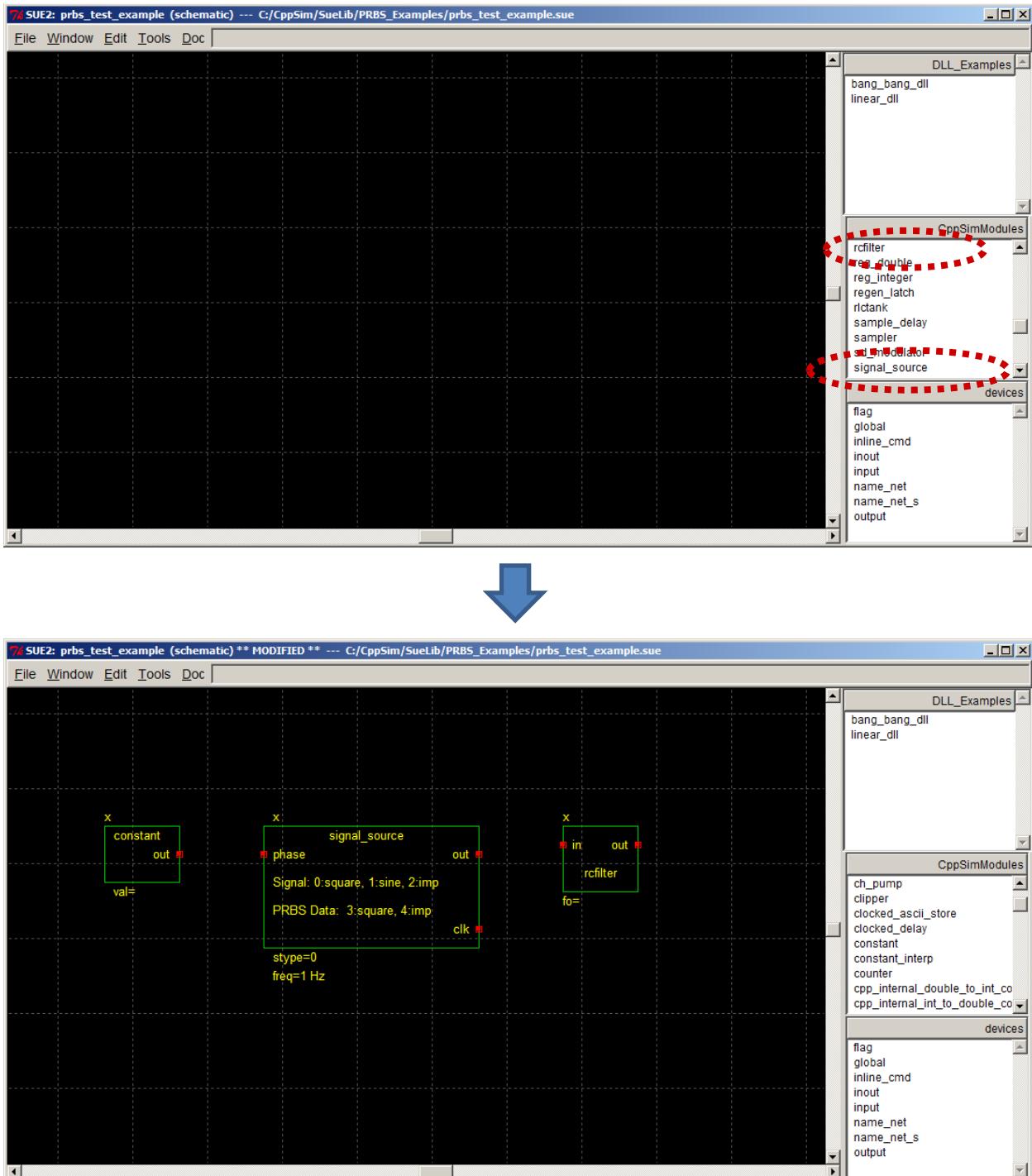


- Click on the **PRBS\_Examples** folder icon, and then specify the **File name** as **prbs\_test\_example** as circled below. Left-click on the **Save** button, as also circled below, to complete the creation of the new schematic. You should now see the top banner of the schematic window state that the new schematic is **C:/CppSim/SueLib/PRBS\_Examples/prbs\_test\_example.sue**. In case this point is not clear, please view the schematic window shown as a figure on the next page in this document to see how this information is displayed.

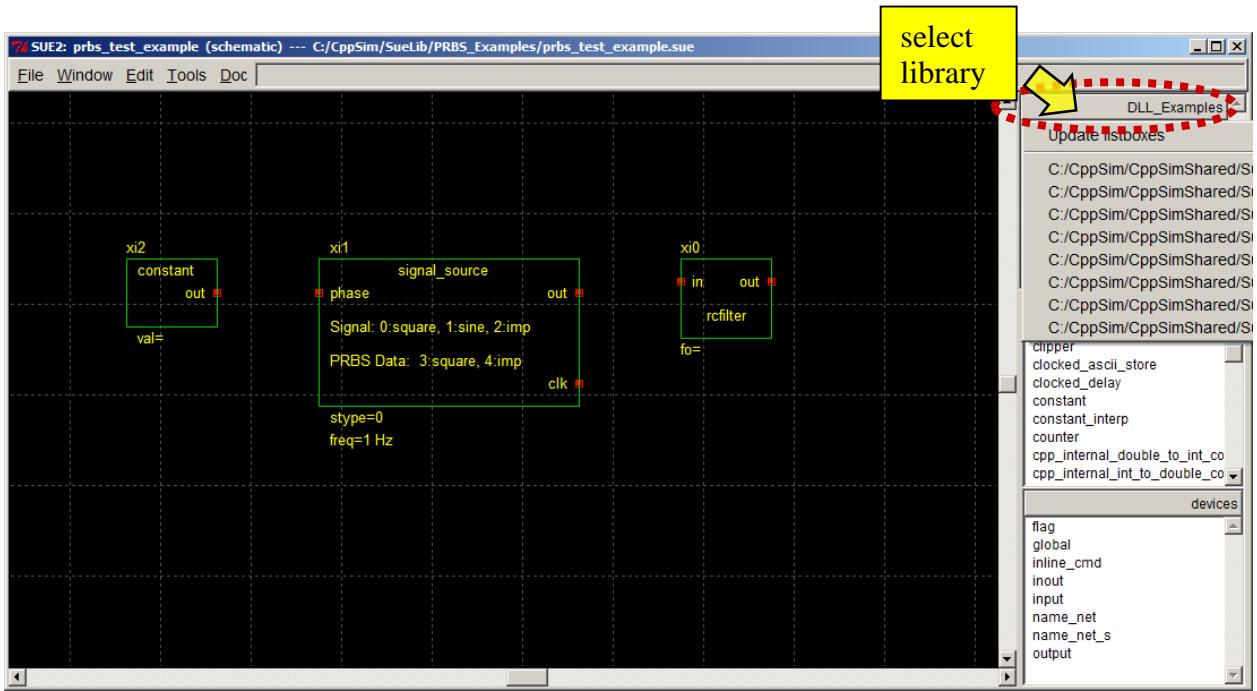


- In the Sue2 **icons1** listbox, as shown below, select the **signal\_source** icon and then move the cursor into the main Sue2 schematic window. Click on the mouse to place the icon at an appropriate place. Then select the **rcfilter** icon, as circled below, and again move the mouse into the main Sue2 schematic window to place this icon to the right of **signal\_source** cell.

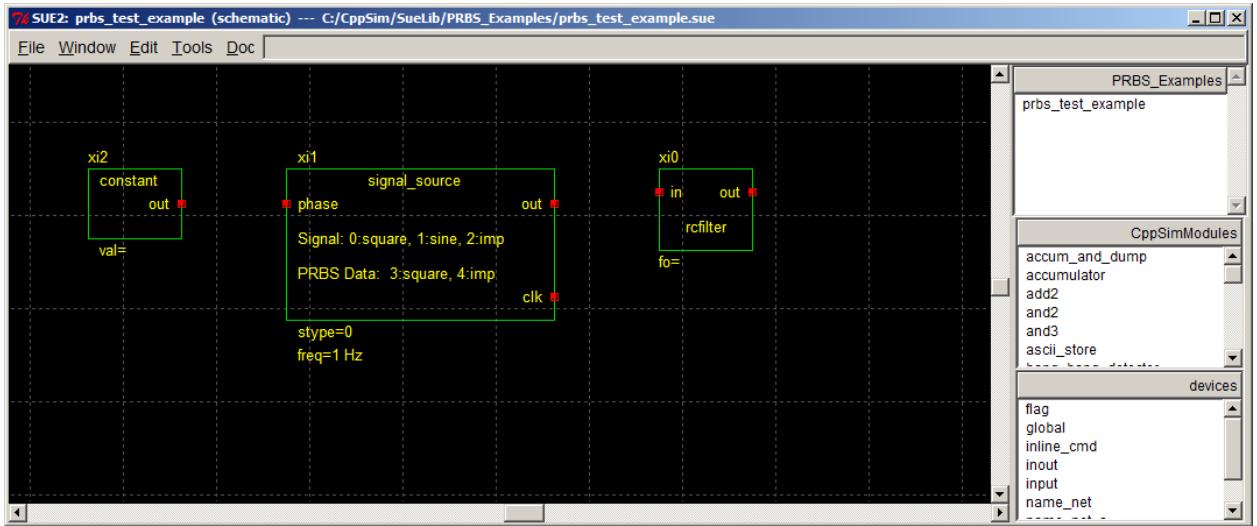
Finally, select the **constant** icon from the **icons1** listbox (you must use the scroll button to see this icon name) and then place it to the left of the **signal\_source** cell. The main Sue2 schematic window should now appear similar to the figure displayed below.



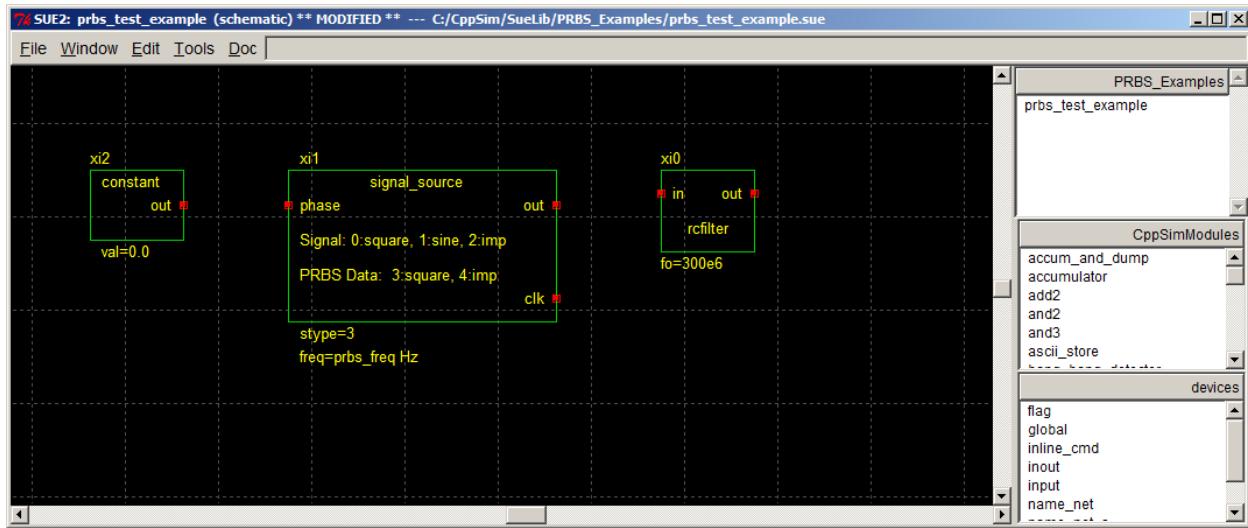
- Save the schematic view by clicking on **File->Save** (or hold the **Ctrl** key and then press the **s** key). If you now click on the top portion of the **schematics** listbox, as circled below, you'll notice that the library **PRBS\_Examples** does not show up.



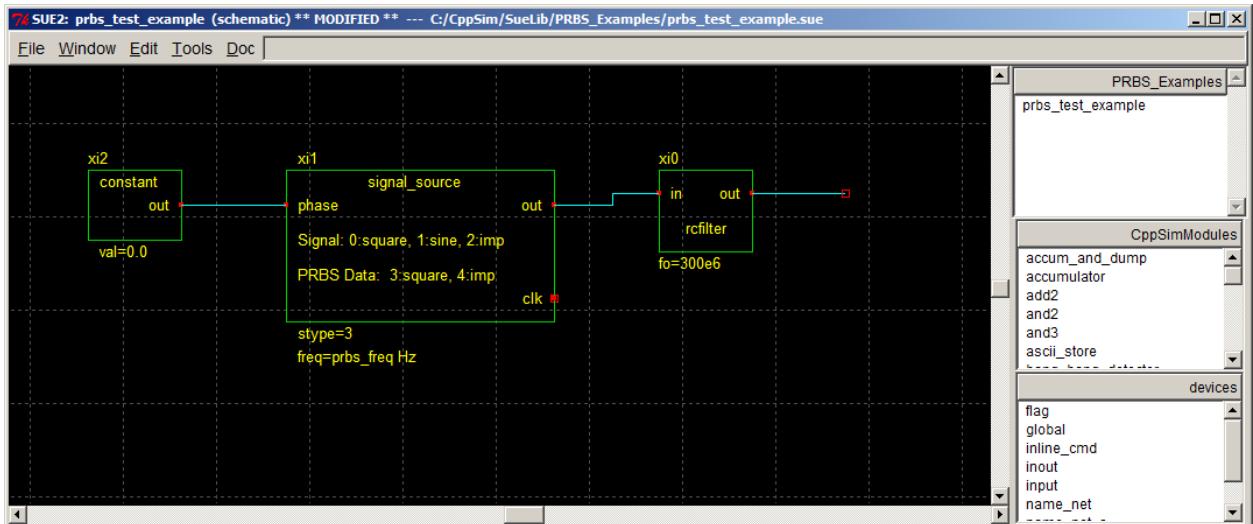
- The issue of **PRBS\_Examples** not showing up as a library in Sue2 occurred since there were previously no cells in this library. Now that you have created a cell for this library, you can correct this issue by exiting Sue2 and then restarting it again. After doing so, you should then click on the top portion of the **schematics listbox**. Now the set of libraries will include **PRBS\_Examples**, which should be selected. You should then choose schematic **prbs\_test\_examples** to re-obtain the same schematic shown above.



- Select parameter values for each of the cells in the above schematic by double-clicking on each of them and setting them as follows:
  - constant cell:** consval = 0.0
  - signal\_source cell:** stype = 3, freq = prbs\_freq
  - rcfilter cell:** fo = 300e6

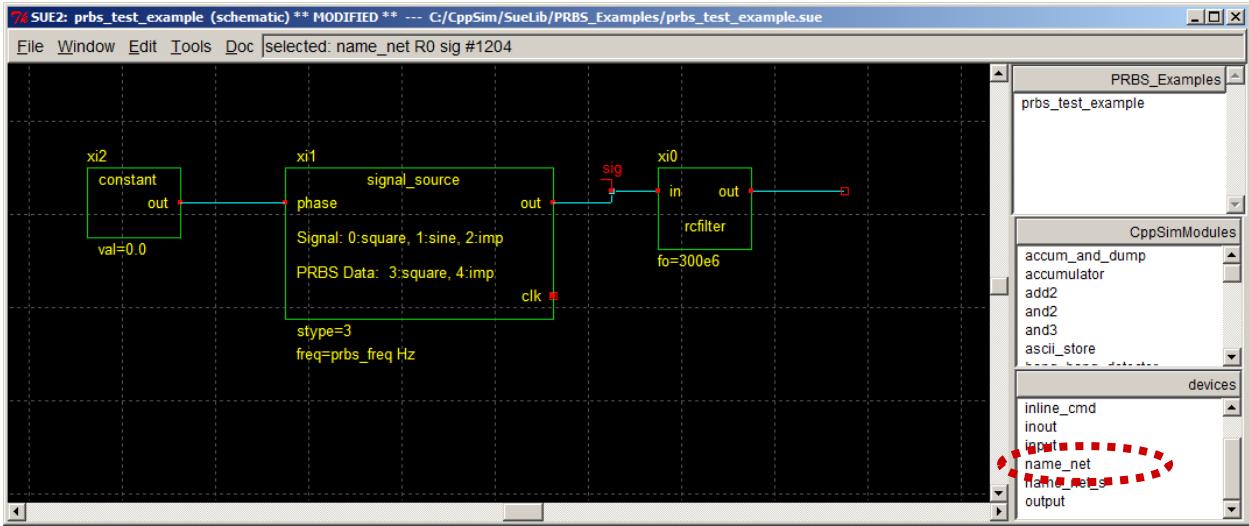


- To connect the cells, we need to add wires. You enter wire-create mode by typing **w** in the main Sue2 schematic window. To start a wire, left-click at the desired starting point (usually at the terminal of a cell). Place the cursor at the end of the desired wire segment, and then left-click to create a new segment. A wire is completed when it is connected to a cell or pin terminal, though double-clicking the left mouse button (or single-clicking the right mouse button) will force the end of a wire at any point in the schematic. Note that you must push the **Esc** key to end wire mode. Given this information, complete wiring for the schematic as shown below.

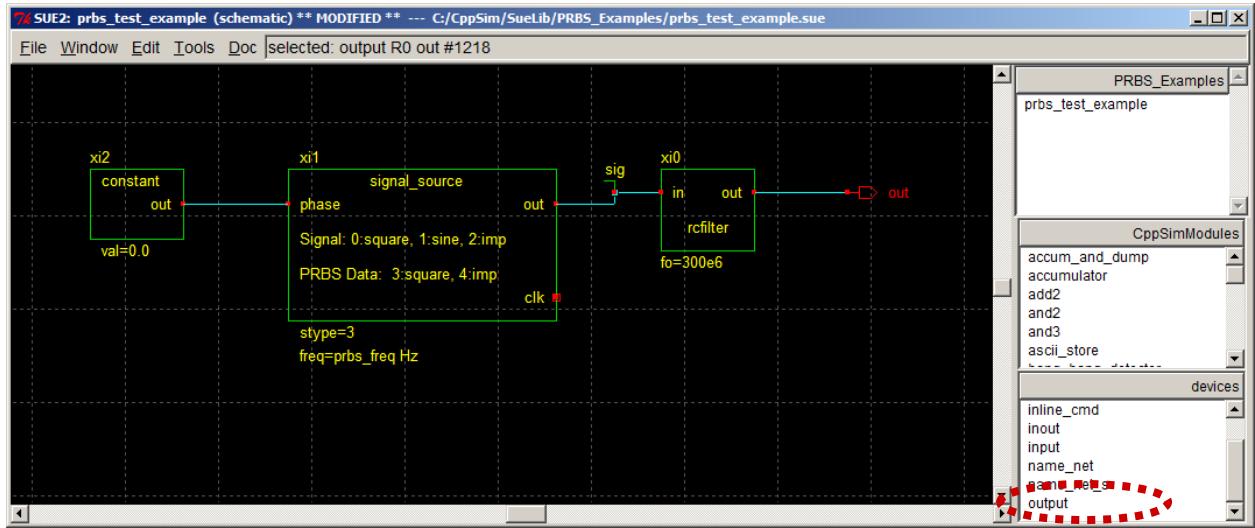


- To probe signals produced in the CppSim simulation of the schematic, we need to label all signals of interest. We also should add pins to any nodes that we might want to bring up to the next level of hierarchy.
  - For this example, let us label the output node of the **signal\_source** cell as **sig**. To do so, click on **name\_net** of the **icons2** listbox (as circled below), move the mouse cursor into the main schematic window, and then place the **name\_net** icon on the wire connected to terminal **out** of **signal\_source**. Double-click on the **name\_net** icon once it is placed, and set its name to **sig**. The schematic figure below illustrates how the

**name\_net** icon should look within the schematic once these operations are completed. Note that you can also use the **name\_net\_s** icon instead of **name\_net** to name nodes – the only difference between them is their appearance.

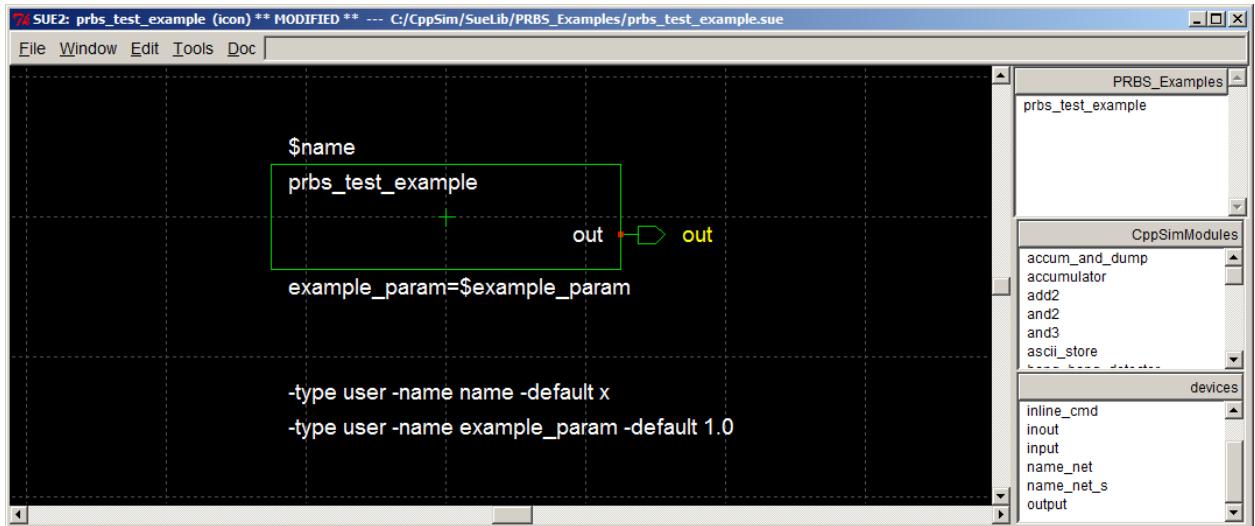


- Add an output pin to the schematic by clicking on **output** in the **icons2** listbox (as circled above), moving the mouse cursor into the main schematic window, and then placing the pin at the output of the rightmost wire in the schematic as shown below. Once the output pin has been placed, double-click on it to change its name to **out**. Be sure to save the schematic at the completion of these operations.



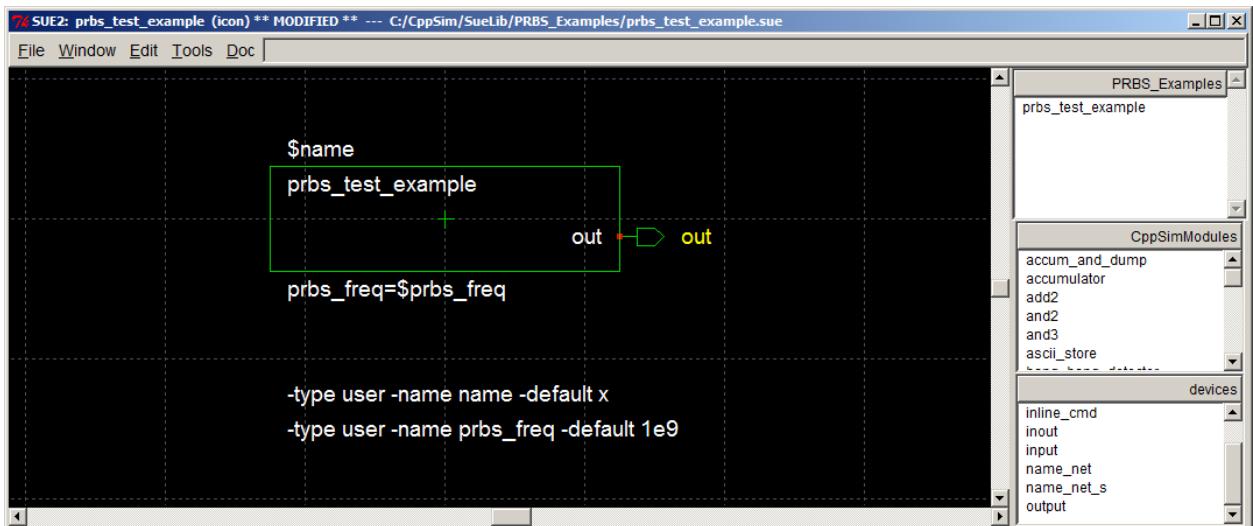
### C. Creating an Icon View (And Associated Parameters) For A Given Schematic

- Assuming you are currently in the schematic shown above, creation of an associated icon is straightforward. Simply click on **Window->make icon**, or press its associated bindkey, **K**. The resulting icon view should appear as shown below. Be sure to click on **File->Save** to save this new icon view.



- The newly created icon view is intended to be a template for the actual icon view desired. We will now change its default parameter, **example\_param**, and explain how to alter its rectangle box.
  - The two statements involving **example\_param** are intended as a template for creating parameters, and should either be removed or modified to reflect a parameter name of interest. The top statement specifies how the parameter and its value will be displayed when the icon is instantiated within a schematic. The bottom statement declares the parameter and provides its default value.

In this case, our schematic has one parameter, **prbs\_freq**, that we would like to implement. To do so, double click on the two statements involved **example\_param** (one at a time), and replace **example\_param** with **prbs\_freq**. Select the default value of **prbs\_freq** to be **1e9**, and add units of **Hz** to the top statement. Hit the **Enter** key each time you complete the changes for a given statement. The figure below indicates how the icon view should look upon the completion of these changes.



- To add more parameters, you would simply add more statements in similar fashion to the two you just modified. Statements can be added by either copying a current statement (click-left on a statement of interest to select it, press **c**, left-click again, then left-click once more to place the copy) and then modifying the copy, or by clicking on **Edit->add text** (bindkey is **t**) and directly entering a new text statement.
- To change the size of the icon rectangle (i.e., the green rectangle shown in the above icon view), double-click on the rectangle and solid boxes will appear at its corners. Left-click on one of the corner boxes and then move the mouse – the associated corner of the rectangle will change in accordance with the mouse movements. Release the left mouse key to retain the current position of the given rectangle corner.
- You have several options for creating shapes for icons in Sue2:
  - Create a line by pushing the **I** (as in **line**) key followed by the left mouse button, and then double-clicking on the left mouse button (or single-clicking the right mouse button) at a different point on the canvas. Multi-segment lines are created by single rather than double-clicking on the left mouse button at each desired breakpoint of the line, with a double-click of the left mouse button (or single-click of the right mouse button) to end the line. Press the **shift** key to limit the drawing of line segments to either the vertical or the horizontal plane. Once a line is created, its various line segments can be modified by first double-clicking on the line, and then pressing and holding the left mouse button over the given breakpoint followed by dragging of the mouse to the new desired location.
  - Create an arc by pressing the **a** button followed by pressing (not holding) the left mouse button, moving the mouse until the appropriate size and shape for the arc is achieved, and then pressing the left mouse button.
  - As mentioned above, create text by pushing the **t** key followed by the left mouse button at the desired location for the text. Modify text by double-clicking on it with the left mouse button and then performing edits. Only three sizes of text are available – the size of the given text segment may be varied while in text mode by holding the **Shift** key and then pressing either the left, middle or right mouse button to select the desired size. Also, the text can be changed to either left, middle or right justified by holding the **Ctrl** key then pressing either the left, middle or right mouse button.
- Save the icon view after you have completed the desired changes. The icon is ready to add to other schematics, and can be accessed in one of the **icons listboxes** by selecting **PRBS\_Examples** as the library for a given **icons listbox**.

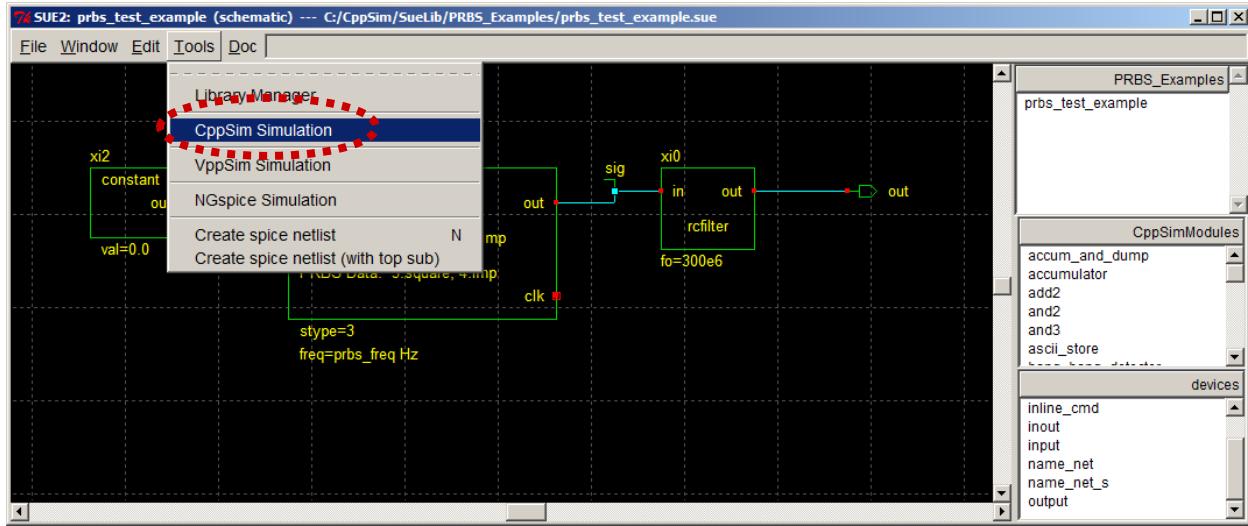
## **Creating and Running New CppSim Simulations**

We now walk through an example of setting up a new simulation file for a schematic, using the **eyesig(...)** plot function, and making use of the **alter:** statement.

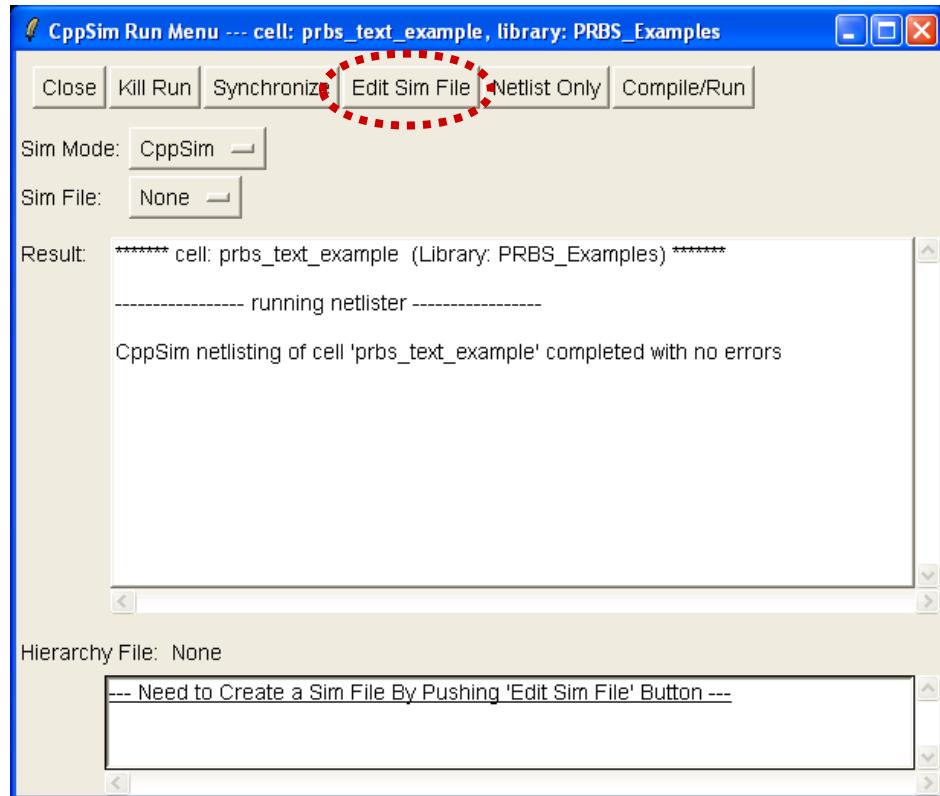
### **A. Creating a New Simulation File for a Newly Created Schematic**

Let us now simulate the newly created schematic **prbs\_test\_example**. If you are currently in the icon view of that cell, simply press **k** to revert back to its schematic view. Alternatively, select

**prbs\_test\_example** in the **schematic listbox** by first selecting the **PRBS\_Examples** library in the listbox and then clicking on **prbs\_test\_example**. The main schematic listbox should now display the schematic shown below. Click on the **CppSim Simulation** menu item as shown below.



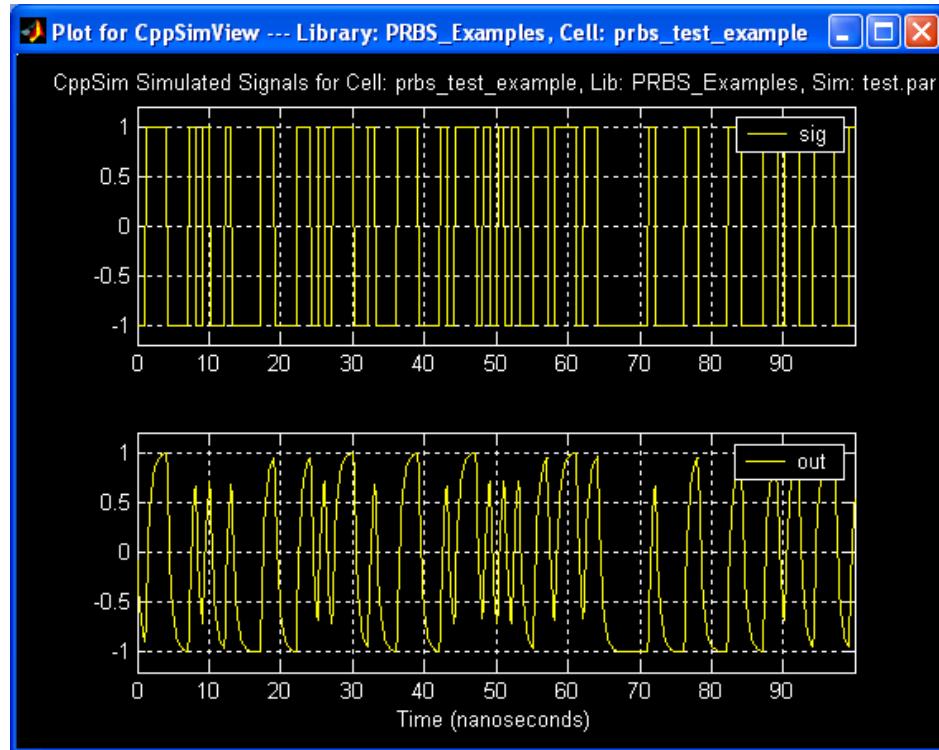
- Click on the **Edit Sim File** button in the **CppSim Run Menu** window as shown below. An Emacs text editor window will appear that contains a template **test.par** file.



- Modify the **test.par** file in Emacs as follows:

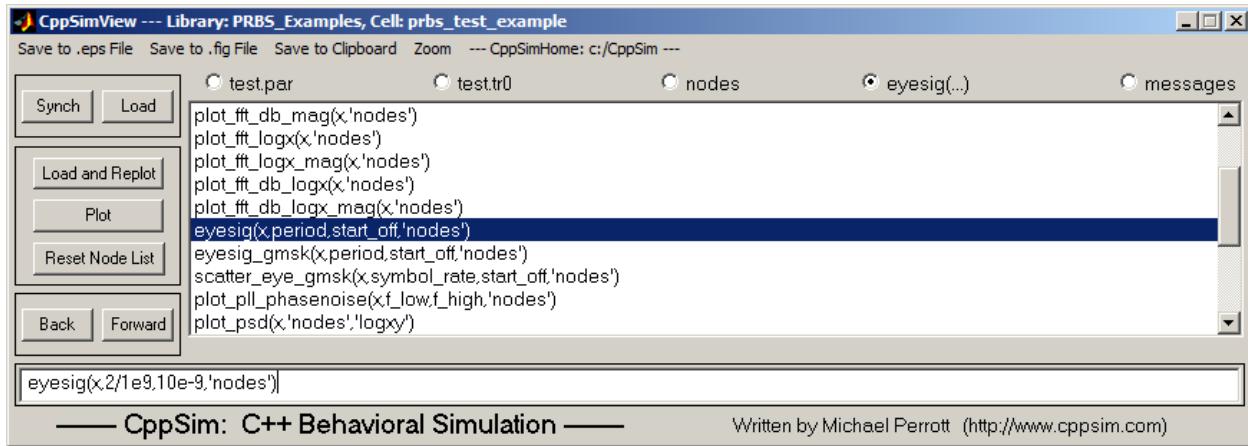
- o num\_sim\_steps: 1e3

- Ts: 1/10e9
  - output: test
  - probe: sig out
  - global\_param: prbs\_freq=1e9
  - alter: (i.e., keep this empty for now)
- Save the **test.par** file, and then press the **Compile/Run** button in the **CppSim Run Menu** to run the CppSim simulation.
  - Now start CppSimView by clicking on its icon. Use the appropriate commands in CppSimView to display signals **sig** and **out**. You should see signals as shown below if all steps were completed correctly.

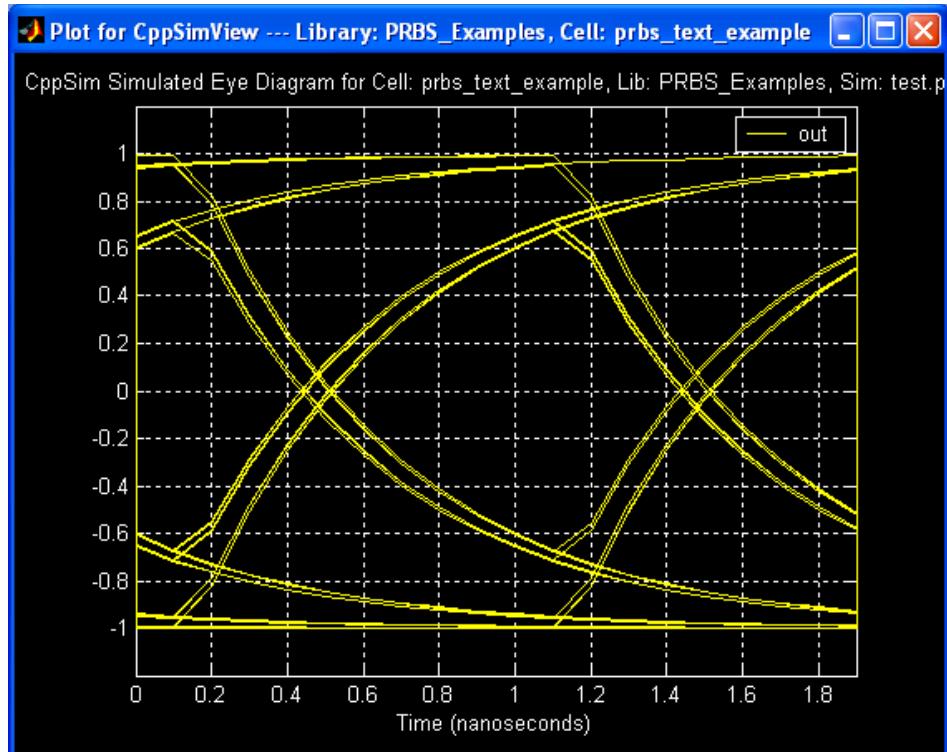


## B. Using the eyesig(...) Plotting Function

- In CppSimView, click on the **plotsig(...)** radio button and then select the **eyesig(...)** function in the listbox that appears. Since the frequency of the PRBS signal source is set at 1e9, choose a value of two periods for the eye diagram to be plotted (i.e., 2/1e9). For the starting point of the eye diagram, choose 10 nanoseconds (i.e., 10e-9) – this choice is rather arbitrary in this case, but the starting point should generally be chosen to remove transients from being considered in the eye diagram generation. After making the above changes, CppSimView should appear as below. Press **Enter** to record the changes in **eyesig(...)** within its listbox entry.



- Now click on the nodes radio button in CppSimView, and then double-click on node **out**. The eye diagram shown below should appear.



## C. Using the alter: Statement

- In the **CppClass Run Menu** window, press the **Edit Sim File** button if the Emacs session of **test.par** is not already open. Change the **alter:** statement to read:
  - alter: prbs\_freq = 1e9 .1e9 2e9**
  - Note: look at the CppSim Reference Manual (i.e., ccppsimdoc.pdf) to get more information on **alter:** statements. It is worthwhile to do so since CppSim allows reasonably sophisticated methods of varying parameters to be accommodated (i.e.,

combinations of parameters can be generated, or parameters can be changed in step with each other).

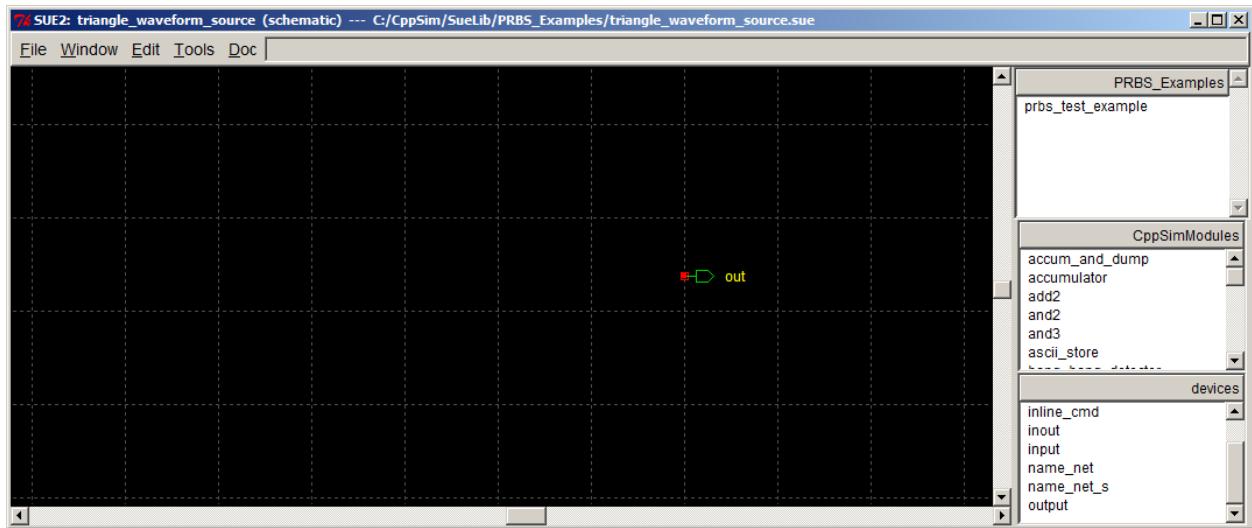
- Save the **test.par** file in Emacs, and then press the **Compile/Run** button in the **CppSim Run Menu** window.
- Within CppSimView, click on the **test.tr0** radio button and you should see that three output files were produced: **test.tr0**, **test.tr1**, and **test.tr2**. You may want to try looking at the eye diagrams for each of these different output files (be sure to change the symbol period in each case as appropriate). Note that a much more efficient way of examining the different output files would be to write a Matlab or Octave script to do the associated post-processing and graphical display such that automatic loading of all the output files was performed. The writing of such a script is beyond the scope of this document, but the user should be aware that the limitation of CppSimView in doing such large-scale post-processing is not a limitation of the CppSim simulation environment itself.

## Creating New CppSim Primitives

CppSim primitives are defined to be cells that do not contain other cells or primitives, and therefore must be represented as code. These cells are simply pins in their schematic view, and a corresponding icon that may contain parameters. In this section we walk through the creation of a new CppSim primitive in Sue2 and its corresponding code in the **modules.par** file.

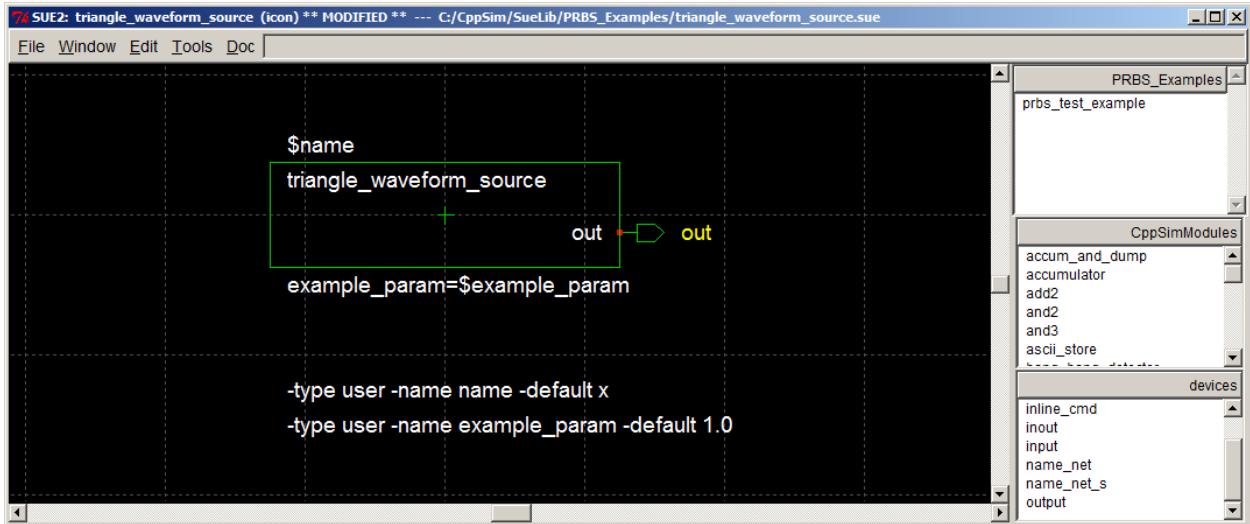
### A. Creating a Schematic View for the Primitive

- Schematic views for CppSim primitives are simply a set of input and/or output pins. For this example, create a new schematic by clicking on **File->New Schematic**. Name the new schematic **triangle\_waveform\_source** and place it into the **PRBS\_Examples** directory. In the schematic view, add an output pin and name it **out**. The schematic should now look as shown below.

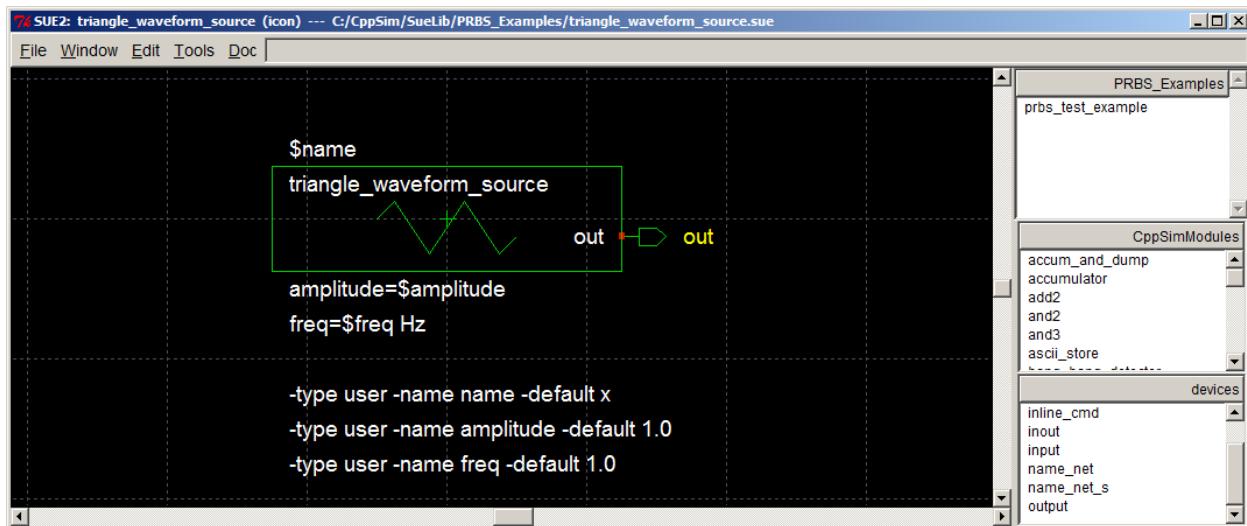


## B. Creating an Icon View for the Primitive

- Hit the **K** key (i.e., **Window->make icon**) to create an icon view for the above schematic. The automatically generated icon view should look as shown below (after it has been saved).

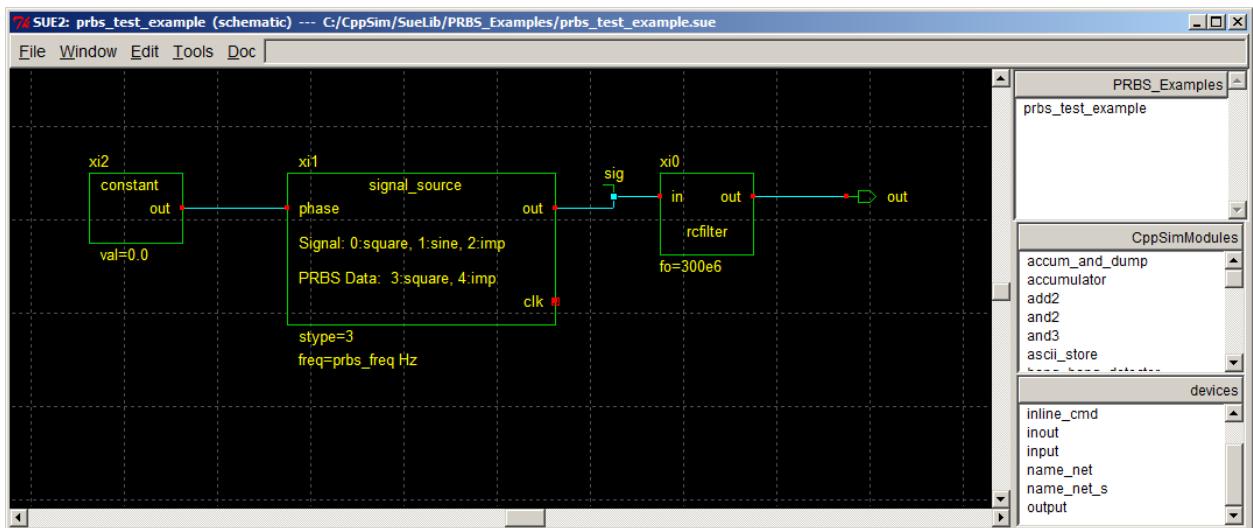
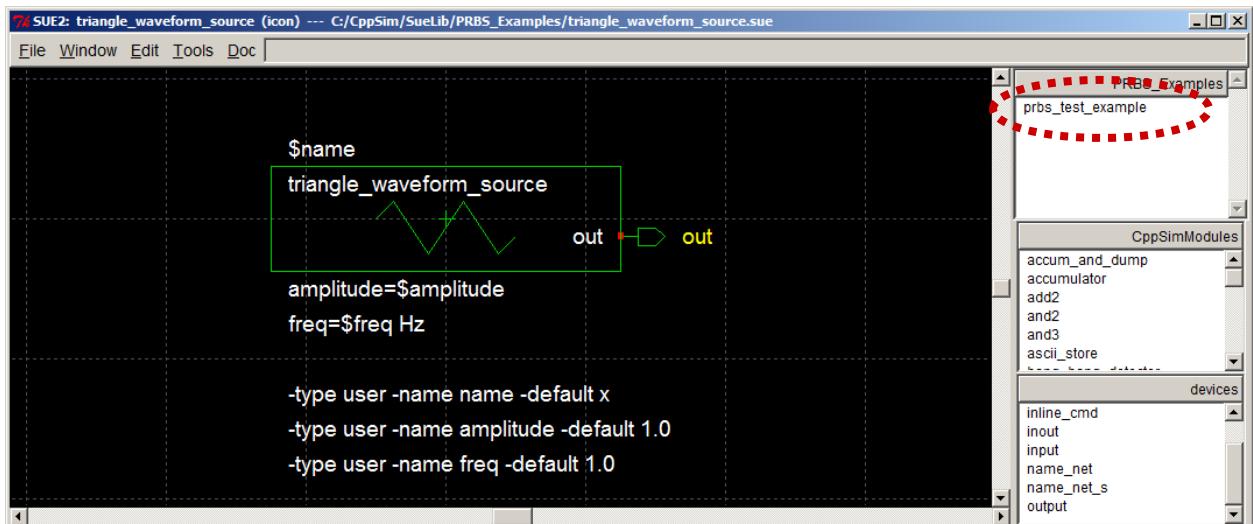


- Modify the icon view so that it supports two parameters, **amplitude** and **freq**, as shown below. Use the line command (**I** key, or **Edit->add line**) to create the triangle waveform placed within the icon rectangle as shown in the figure. Once you have saved this icon view, it is now ready to be placed within other schematics.

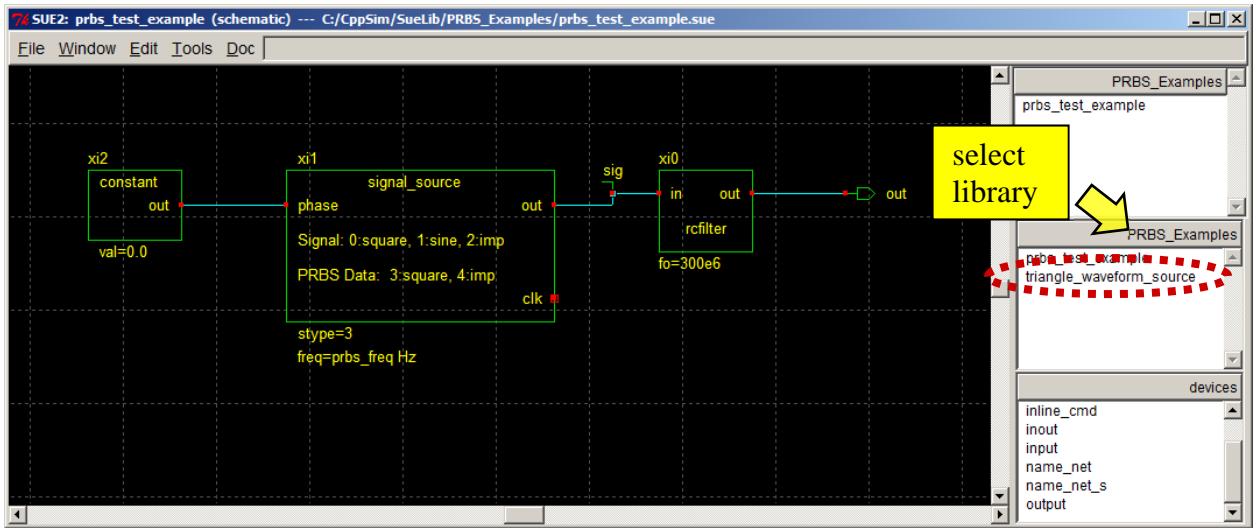


## C. Instantiating the Primitive Within a Different Schematic

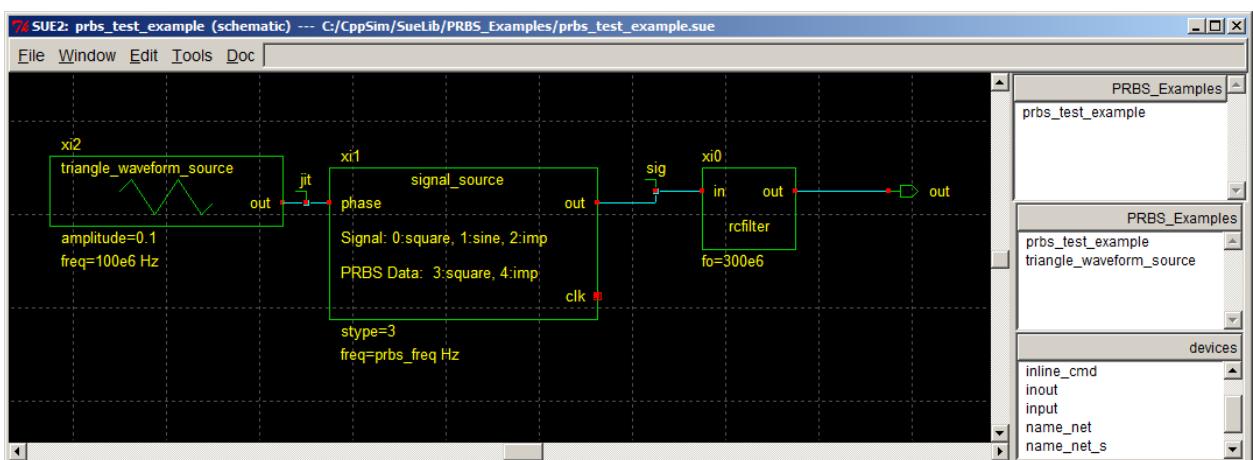
- Use the **schematics listbox** to bring up the **prbs\_test\_example** schematic that was created in an earlier section of this manual. You should then see the schematic as shown below.



- Replace the **constant** block in the above schematic with the **triangle\_waveform\_source** icon that you just created. To do so, the first step is to select the **PRBS\_Examples** library in the **icon1** listbox as shown below.

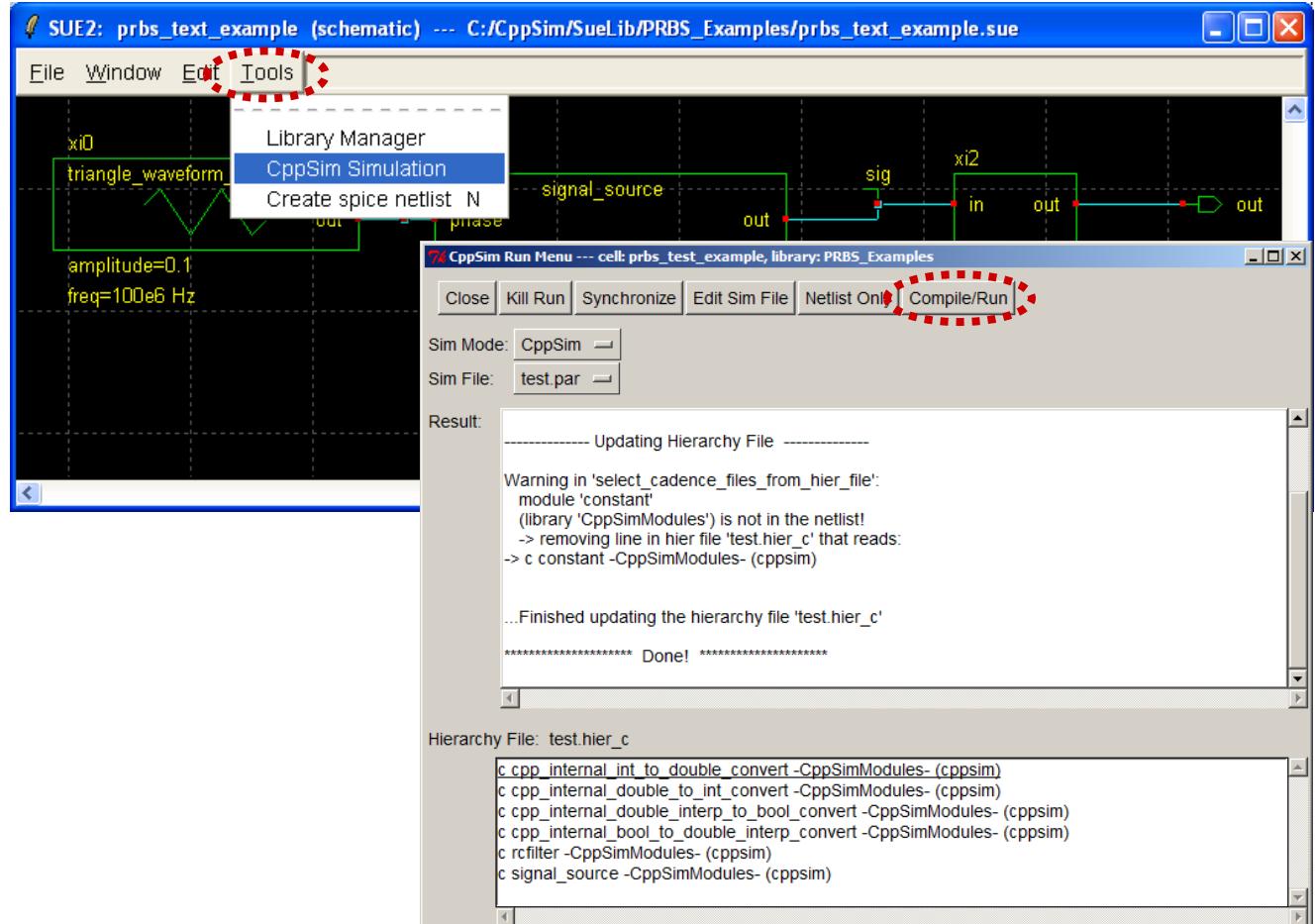


- Now delete the **constant** block by left-clicking on it and then pressing the **delete** key. Select the **triangle\_waveform\_source** entry from the **icon1 listbox** as circled above, move the mouse cursor into the schematic, and then left-click the mouse to place the **triangle\_waveform\_source** cell. Be sure not to try selecting the **triangle\_waveform\_source** from the **schematics listbox** – you will not be able to add the icon to the schematic above, but rather will be sent to the schematic view of **triangle\_waveform\_source**.
- Set the parameters of the **triangle\_waveform\_source** to be
  - amplitude = 0.1**
  - freq = 100e6**
- Finally, label the wire connected to the **out** terminal of **triangle\_waveform\_source** as **jit** using the **name\_net** symbol, and then save the schematic.
- Upon completion of the above operations, the schematic should look as shown below.

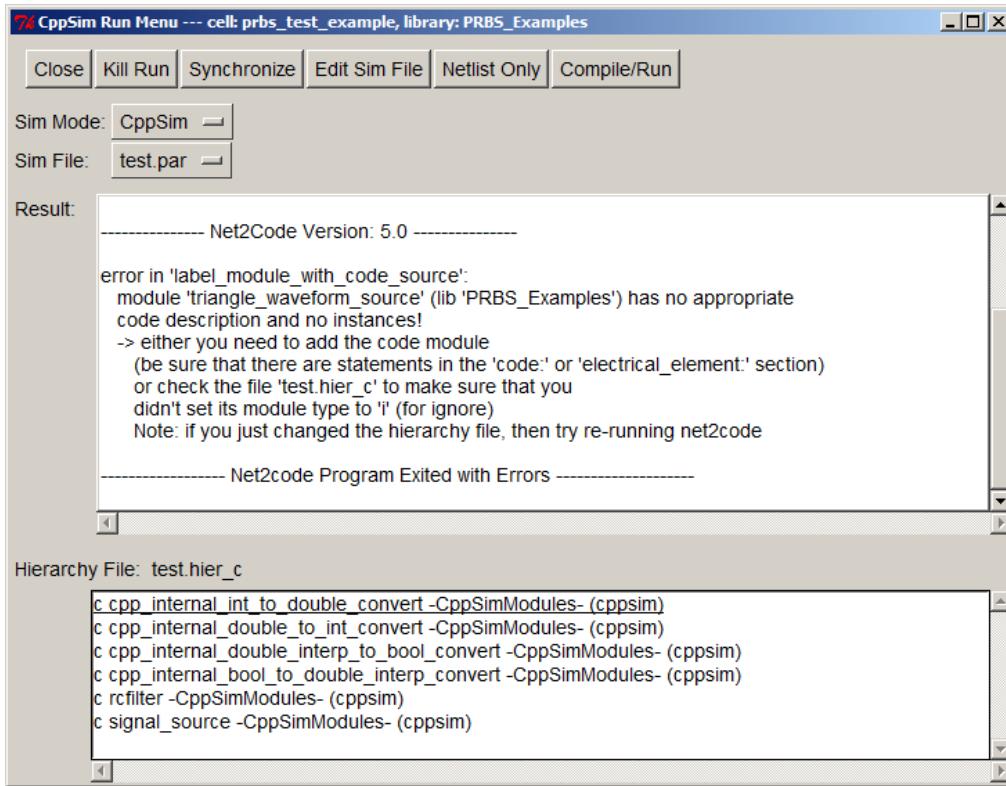


## D. Running CppSim with the Primitive

- Now open the **CppSim Run Menu** and click on the **Compile/Run** button, as shown in the figure below.

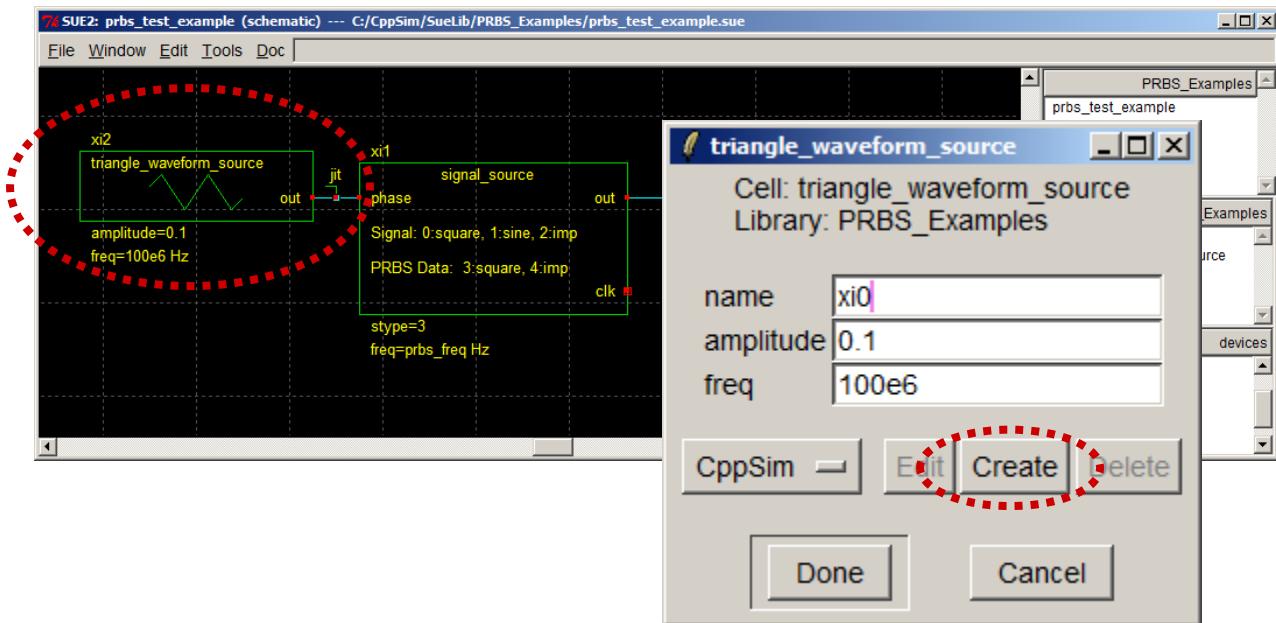


- Assuming that you already created a **test.par** file for **prbs\_text\_example** in the previous section of this manual, you should see the error message shown in the figure below. The error message indicates that we need to supply code for the **triangle\_waveform\_source** module since it is a primitive cell (i.e., it is not composed of other cells or primitives). In the following subsection, we will provide you with basic code to implement this module without providing a lot of comment on its details – please refer to the CppSim reference manual to gain a more solid understanding of creating CppSim modules.



## E. Creating Code for the Primitive

- To begin creation of module code for the **triangle\_waveform\_source** block, double-click on its icon in the Sue2 window as circled in the figure below. In the pop-up window that appears, click on the **Create CppSim** code button. An Emacs window will appear with a template of the CppSim module code description for the **triangle\_waveform\_source** module.



- In the Emacs window that appears from the previous operation, enter text such that the module code appears as shown below. Be sure to save the file once you are done.

---

```

module: triangle_waveform_source
description: produces a triangle waveform with amplitude of
    'amplitude' and frequency 'freq' (note that the peak-to-peak
    amplitude of the output will be twice that of 'amplitude')
parameters: double amplitude, double freq
inputs:
outputs: double out
classes:
static_variables: double phase_step, double phase
init:
phase_step = freq*Ts;
out = 0.0;
phase = 0.0;

end:
code:
phase += phase_step;

if (phase >= 0.5)
    phase -= 1.0;

out = 4.0*amplitude*fabs(phase) - amplitude;

electrical_element:
functions:
custom_classes_definition:
custom_classes_code:

```

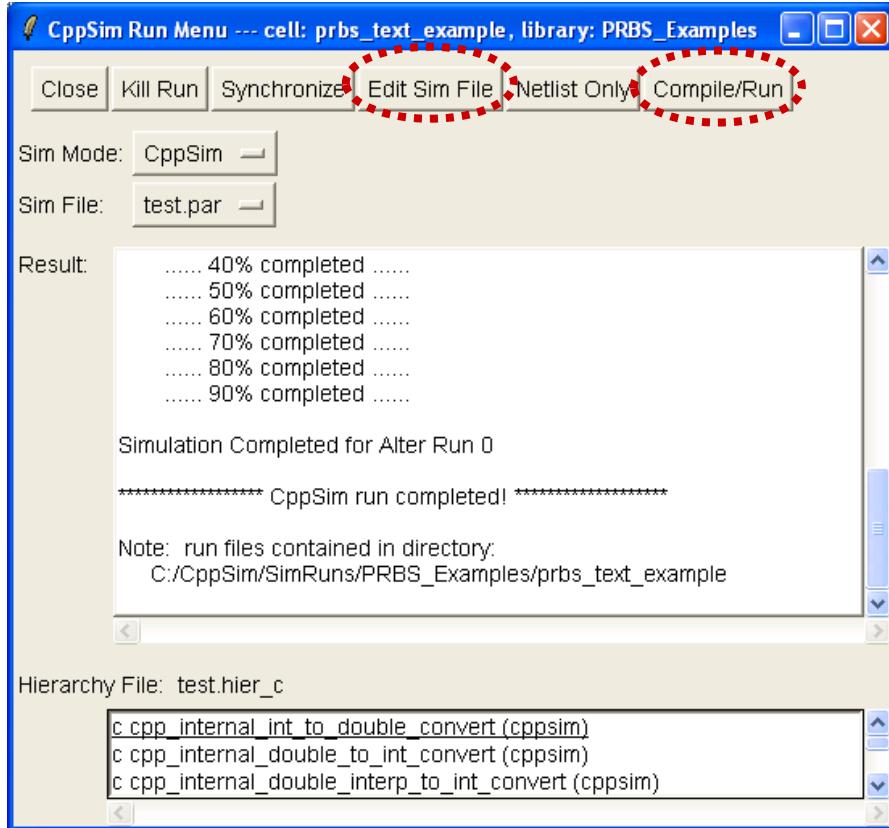
---

The above code creates a triangle wave with peak-to-peak amplitude equal to **2\*amplitude**, and with frequency equal to **freq** Hz. This is one of many ways to create this block – the above approach was primarily chosen for its simplicity. Note that the code within the **code:** section computes the next time sample of the module outputs given the current inputs and parameter values. Code within the **init:** section is run once at the beginning of each alter run, and is used to run one-time computations and to initialize variables and outputs. The code within the **end:** section is run at the very end of the alter run, and is left blank in this case since no specific “end” operations need be done in this case. The other items at the end such as **electrical\_element:** and **functions:** are left blank since they are not used at this time (you may also delete them from the code since they are unused).

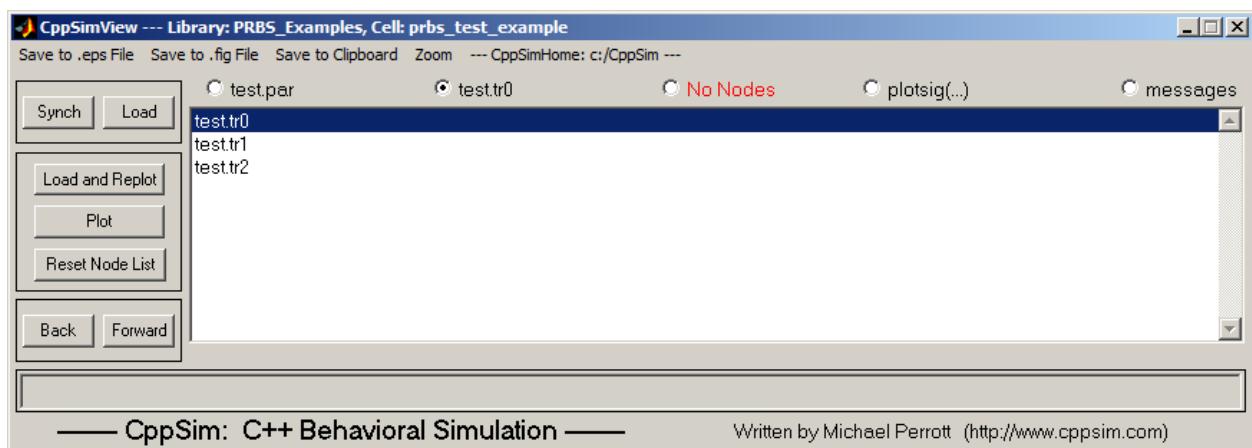
CppSim modules can be much more complex than the above example, and can take advantage of the CppSim classes described in the CppSim Reference Manual. The reader is encouraged to read the CppSim Reference Manual for more information on this topic, and to examine the code files of various modules to see examples of module code.

## F. Running CppSim with the Primitive (Part II)

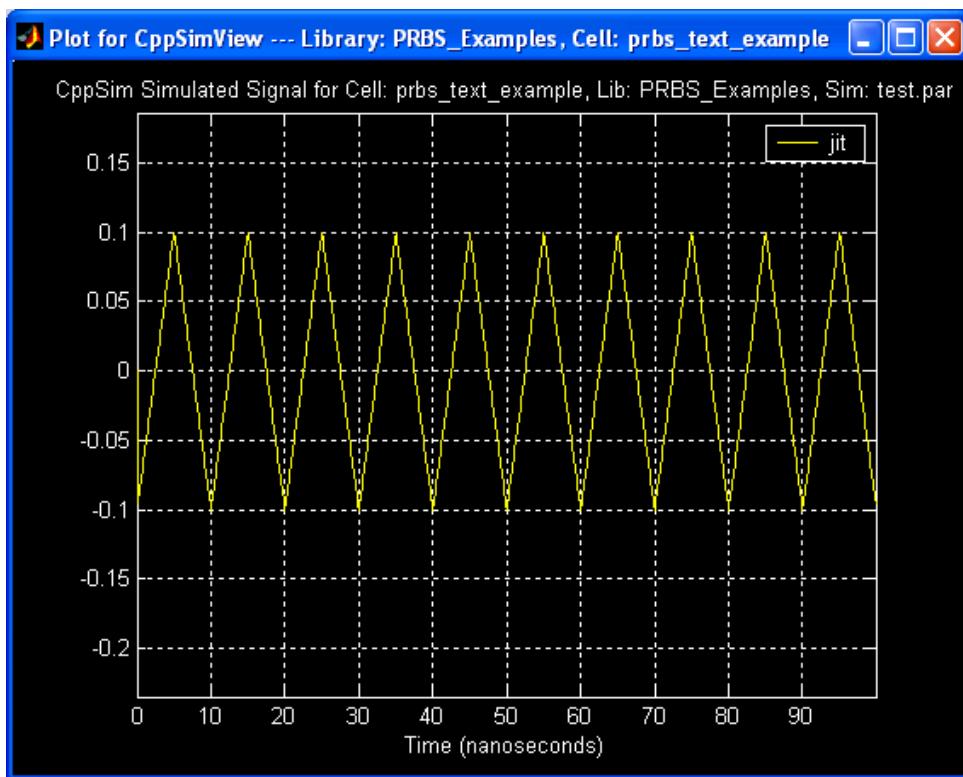
- Before running CppSim again, first left-click on **Edit Sim File** in the **CppSim Run Menu** window (as shown in the figure below). Make the following changes to the **test.par** file that comes up in Emacs:
  - Add **jit** to the **probe:** statement: `probe: sig out jit`
  - Comment out the **alter:** statement: `// alter: prbs_freq = 1e9 .1e9 2e9`
- Now run CppSim by clicking on **Compile/Run** in the **CppSim Run Menu** window as shown below. Notice that only one alter run is simulated since the **alter:** statement was commented out.



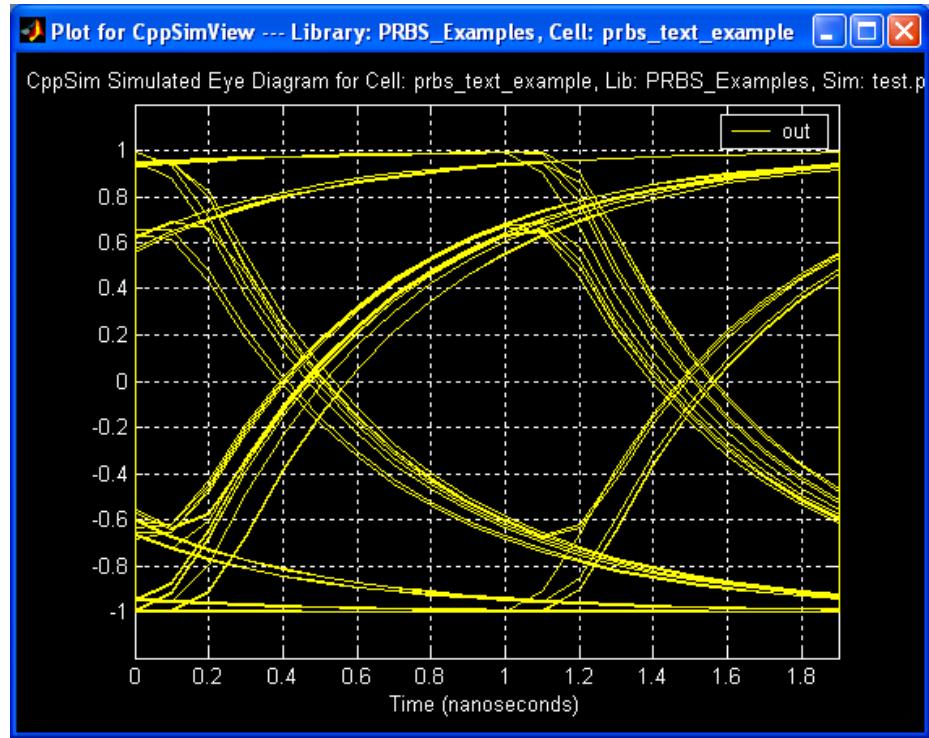
- Now start CppSimView by clicking on its icon. You should see the window shown below.



- Click on the **test.tr0** radio button. Notice that, although only one alter run was simulated, there are three output files (**test.tr0**, **test.tr1**, and **test.tr2**). In fact, only **test.tr0** is valid – the other two output files are left over from previous simulations which used the **alter:** command. To remove **test.tr1** and **test.tr2**, you must use standard Windows programs, such as Windows Explorer, to go into the simulation directory (which in this case is **c:/CppSim/SimRuns/PRBS\_Examples/prbs\_test\_example**) and then delete the files. Note that if you were to uncomment the **alter:** statement in the **test.par** file, then 3 alter runs would be simulated and all three of the output files would be valid.
- Click on the **No Nodes** radio button in CppSimView, and plot the **jit** signal using the **plotsig(...)** function. You should see the waveform shown below.



- Now use the **eyesig(...)** function to plot the eye diagram of the **out** signal. You should see the waveform shown below. Notice that the effect of the phase variation of the source is to cause jitter on the overall output of the system (no surprise!).



## Conclusion

This primer document covered basic operation of the CppSim simulator in the context of using the Sue2 schematic editor, the CppSimView waveform viewer, and Matlab. The reader is encouraged to also read the CppSim Reference Manual (<c:/CppSim/CppSimShared/Doc/cppsimdoc.pdf>), the Sue2 Manual ([c:/CppSim/CppSimShared/Doc/sue2\\_manual.pdf](c:/CppSim/CppSimShared/Doc/sue2_manual.pdf)), the Hspice Toolbox Manual (<c:/CppSim/CppSimShared/HspiceToolbox/document.pdf>), and a paper explaining PLL simulation techniques that are leveraged in CppSim for fast and accurate simulation of these systems (<c:/CppSim/CppSimShared/Doc/paper.pdf>). Each of these documents is also accessible from the **Doc** menu button of Sue2. The reader is also encouraged to look at the various Sue2 examples provided in the CppSim package and their module code – these examples include frequency synthesizers, CDR circuits, and DLL circuits.