

Custom Netlist Procedures in AMS Designer

**Chandrashekar L Chetput (shekar@cadence.com),
Devendra Deshpande(devendra@cadence.com),
Dori Singewald (dori@cadence.com),
Kat Nellayapan (kat@cadence.com),
Gene Lauritano (genel@cadence.com).**

Cadence Design Systems Inc.

**INTERNATIONAL CADENCE USERSGROUP CONFERENCE
September 13-15, 2004
Santa Clara, CA.**

Abstract

This paper will describe various techniques that can be used to generate customized Verilog-AMS netlists in AMS Designer. Suitable examples reflecting design methodology requirements where customized netlists are useful will also be presented. Note that custom netlist procedures is a new feature in AMS Designer, released in IC 5.0.33.

1.0 Introduction

Custom netlist procedures are used by designers to generate customized netlists. This capability is already available in Spectre/Spectre Direct. This paper discusses the generation of custom netlist procedures in AMSDesigner.

AMS Designer uses a Verilog-AMS netlister to generate standard Verilog-AMS[2] netlists from schematics via the DFII/Hierarchy Editor AMS environment. In several situations, it is desirable to control and customize the generation of netlists to address various design, flow and methodology requirements of designers. This paper discusses various techniques that can be used to generate customized Verilog-AMS netlists using AMS Designer depending on user requirements. Note that custom netlist procedures is a new feature in AMS Designer, released in IC 5.0.33.

Some example scenarios where customized netlists are useful are:

- Customize cellview parameter values by generating expressions/equations for simulating corners.
- Support pcells (i.e netlist parameterized ports and extra terminals via dynamic simInfo capabilities).
- Customize instance parameter overrides.
- Customize instance master(cell) names(e.g.: modelName) based on a parameter value.
- Customize the generation of instance names(useful for scalarized array of instances) - similar to 'hnlMapInstInName' in Artist.
- Customize the generation of instances using macro substitution(i.e replace instantiation statements with actual behavioral code).

Netlist customization is a powerful feature that can vary from simple customizations to complicated fully customizable netlist procedures. Given the different types of customizations possible, it is important to choose the appropriate method based on the design/methodology requirements. This paper addresses the various capabilities of netlist procedures from the perspective of a designer's requirements. It attempts to provide the scope of various netlist customization capabilities so that a designer can choose the most appropriate solution to meet his/her design/methodology requirements.

For specifics about the syntax of netlist procedures, refer to [1] "[Cadence AMS Environment User Guide for IC 5.0.33.](#)", on page 21

The paper starts with an overview of netlist procedures in SpectreDirect that provides a background for capabilities that existed before netlist procedures were supported in AMSDesigner. It then gives an overview of how netlist customization works in AMSDesigner and what is supported in IC 5.0.33. This is followed by discussions on various levels of customizations possible based on designer's requirements. Suitable examples are provided to illustrate the various levels of netlist customization.

2.0 Netlisting procedures for spectreDirect

The Direct integration for spectre is based on OSS (Open Simulation System). The whole netlisting process is divided into two classes: Netlister and Formatter. The Netlister is the traversal engine that goes through designs and gathers information about instances, nets, terminals and so on. The gathered information is passed on to the Formatter object for actual printing to the netlist file.

The Direct nlNetlister is not completely customizable, it has options to modify certain behavior though. For example, a couple of options deal with how the names are to be mapped and the kind of prefixes or suffixes that are to be used.

The Netlister object drives the Formatter object by calling appropriate methods of the Formatter. A Formatter class should derive from nlAnalogFormatter to generate Spice like formats. Users may write their own formatters for integrations with different simulators.

However, a large class of users do not need to write whole new formatters. Of interest is the capability to just be able to format some aspects of an instance in a different way. Netlist procedures is a way to achieve the same. The user simply puts the name of a procedure in the netlistProcedure field of the spectre section in the CDF simulation information of the particular device. The procedure should accept two arguments: the formatter and an instance of that master. After the netlist procedure has been established, it should print the instance completely.

The netlist procedure may use methods provided by the formatter class to format different parts of the instance statement. There are methods available to obtain the name of the instance, the name of the model, the terminal connections, the parameters and so on.

3.0 Netlisting Procedures for AMSDesigner

3.1 Overview

The netlisting process in AMSDesigner is represented by data objects in such a way as to provide finer granularity for controlling the format and content of the netlist. The netlister object is the top-level object which contains information about the formatter object. Currently, AMSDesigner supports Verilog-AMS netlisting only(from schematics). Therefore, the Verilog-AMS formatter is the only formatter object accessible from the netlister. Before looking at other data objects,

here is an overview of the various types of customizations possible in AMSDesigner.

Netlist customization is controlled using SKILL procedures which specify the format and content of the netlist and when it should be customized. Various levels of customizations are possible in AMSDesigner as described below:

1. Customize using environment variables specified in the 'ams.env' file. This provides basic support for controlling the format of certain parts of the netlist.
2. Customize the data in the netlist by modifying its value only. This provides more control over the data(content of the netlist) while retaining the default format of the netlist.
3. Customize the data and the format of the netlist. This provides complete control over the format and content of the netlists - like adding/removing data in the netlist and customizing the formatting of several parts of the netlist. This is the most powerful customization technique and can be the most complicated too.
4. Customize using the 'netlistProcedure' field in CDF->AMS simInfo for a given cell. This allows the user to customize the netlisting of all instances of a given cell (i.e on a cell-by-cell basis).
5. Levels of precedence are defined to manage customizations via 3. and 4. together. Netlist procedures specified at the cell level(i.e 4.) will take precedence over the global netlist procedures(i.e 3.).
6. Customize using dynamically evaluated SKILL procedures which are specified in any of the fields of the CDF->AMS simInfo for a given cell. These procedures are executed at the appropriate time during netlist generation; to compute a value for that field; which is specific to that instance of the cell. The resulting value is then used in the netlist generation. Note that this type of customization is unique to the AMSDesigner solution and is referred to as "dynamic simInfo".

With this overview about various levels of customization, the next section will provide a brief description of how the Verilog-AMS netlister partitions the netlist that enables simple, flexible and powerful techniques to control the content and format of the netlist.

3.2 Components of a Verilog-AMS Netlist in AMSDesigner

In AMSDesigner, the netlist is partitioned into several sections to provide maximum control and flexibility over the content of the netlist. [Figure 1: "Verilog-AMS Netlist Sections", on page 4](#) shows the various sections of the netlist. [Table 1: "Mnemonics representing the netlist sections", on page 5](#) shows the mnemonics representing these sections of the netlist. The mnemonic representation can be passed as an optional argument to the 'amsPrint()' helper function[1] ["Cadence AMS Environment User Guide for IC 5.0.33.", on page 21](#) to customize the content

and format of a specific section in the netlist.

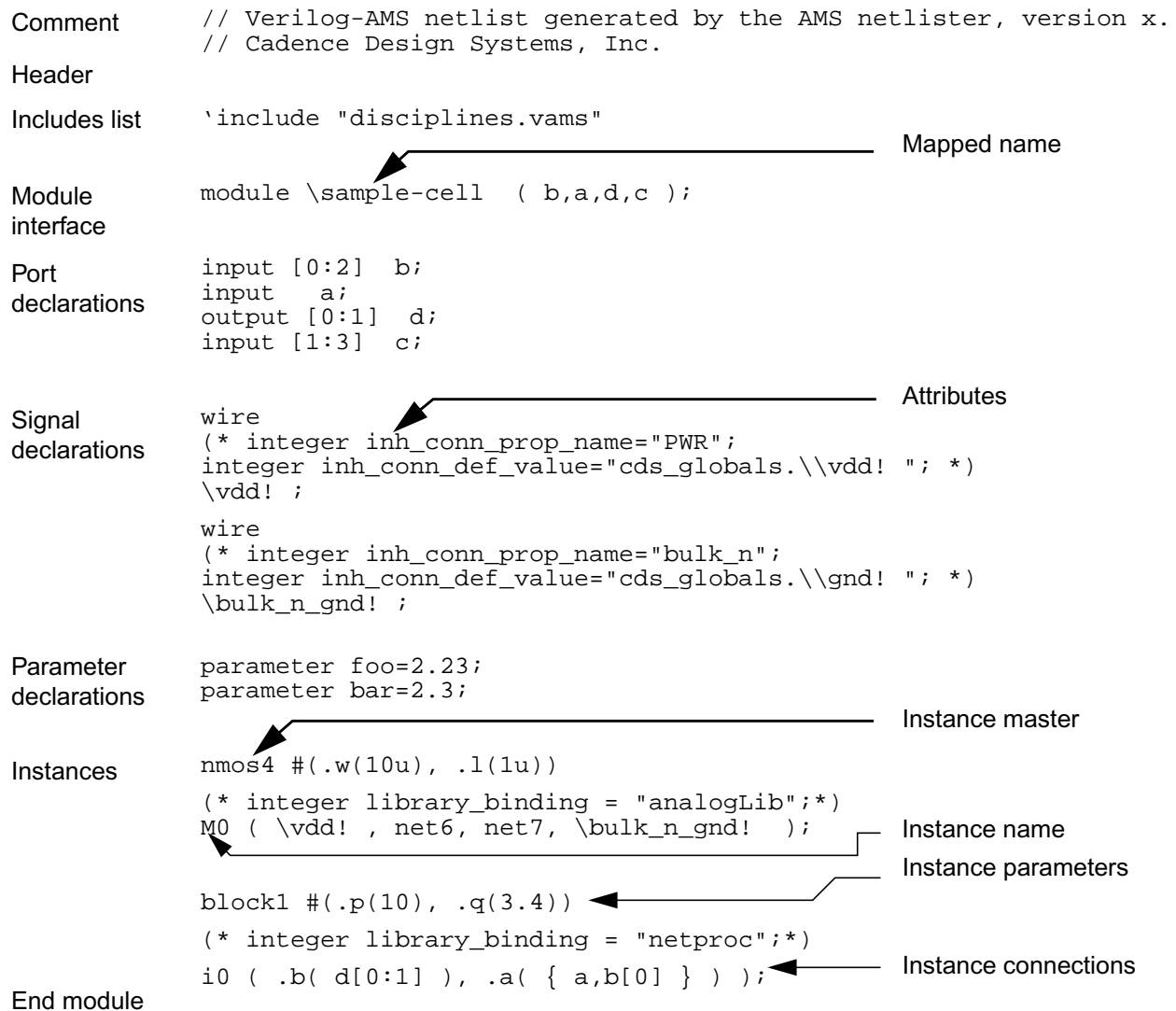


Figure 1: Verilog-AMS Netlist Sections

The section information is automatically maintained appropriately during the netlisting process. This mechanism allows the user to control the format of various sections of the netlist. For example, if the user wants to add an additional parameter to the cellview being netlisted(CBN) based on its availability on a particular instance, then the user can accomplish this using the ‘amsPrint’ helper function.

Netlist Section	Mnemonic to be used in 'amsPrint' helper function.
Comment, Header, Includes List	AMS_VLOG_INCLUDES_LIST
Module Interface	AMS_VLOG_MODULE_IF
Port Declarations	AMS_VLOG_PORT_DECL
Signal Declarations	AMS_VLOG_SIGNAL_DECL
Parameter Declarations	AMS_VLOG_PARAM_DECL
Instances(including cds_alias instances)	AMS_VLOG_INSTANCES
End Module	AMS_VLOG_END_MODULE

Table 1: Mnemonics representing the netlist sections

During netlisting, the netlister processes the schematic data to arrive at the content that finally gets generated into the netlist. While the netlist is partitioned into various sections, the data processed by the netlister is also classified into various objects. These object types reflect, to a certain extent, the processing associated with various sections in the netlist. The next section will introduce the various data objects used in the netlisting process.

3.3 Data Objects for Netlisting in AMSDesigner

The section "[3.1 Overview](#)", on [page 2](#) introduced the Netlister and Formatter objects. The various other data objects provided in AMSDesigner to support customized netlisting are:

- Cellview Object
- Parameter Object
- Instance Object
- Port Object
- IO Object
- Wire Object
- Alias Object and
- Attribute Object

Of these, in IC 5.0.33, only the cellview, parameter and instance objects are supported with custom netlist procedures. For more information on these objects, refer to [\[1\] "Cadence AMS Environment User Guide for IC 5.0.33.", on page 21](#)

The formatter object contains the fields that represent the default netlist procedures. These fields

can be overwritten with names of custom netlist procedures to generate customized netlists. For example, say, 'vlogId' is the formatter object. Then, the field 'vlogId->parametersProc' specifies the netlist procedure to be used to print the parameter declarations of the cellview being netlisted(CBN). By default, its value is 'amsPrintParameters'.

If the designer is interested in generating a customized netlist for CBN parameters using a custom netlist procedure, say 'MYPrintParameters', it can be accomplished by setting the appropriate formatter field as follows:

vlogId->parametersProc = 'MYPrintParameters'

Table 2: "Formatter fields representing the netlist procedures", on page 7 provides a list of various fields of the formatter that represent the netlist procedures along with information about the default netlist procedures and the default section Ids used by 'amsPrint' helper function. For more information, refer to [1] "Cadence AMS Environment User Guide for IC 5.0.33.", on page 21

Formatter Field	Can it be Overridden?	Default netlist procedure	Arguments	Default SectionId used for generating the netlist
commentsProc	yes	amsPrintComments	A_formatterId A_cellViewId	'INCLUDES_LIST
headersProc	yes	amsPrintHeaders	A_formatterId A_cellViewId	'INCLUDES_LIST
moduleProc	no	amsPrintModule	A_formatterId A_cellViewId	Not Applicable
footersProc	yes	amsPrintFooters	A_formatterId A_cellViewId	'END_MODULE
moduleName-Proc	no	amsPrintModuleName	A_formatterId A_cellViewId	'MODULE_INTERFACE
portsProc	yes	amsPrintPorts	A_formatterId A_cellViewId	'MODULE_INTERFACE
iosProc	yes	amsPrintIOs	A_formatterId A_cellViewId	'PORT_DECLARATION
parametersProc	yes	amsPrintParameters	A_formatterId A_cellViewId	'PARAMETER_DECLARATION
wiresProc	yes	amsPrintWires	A_formatterId A_cellViewId	'SIGNAL_DECLARATION

Formatter Field	Can it be Overridden?	Default netlist procedure	Arguments	Default SectionId used for generating the netlist
aliasesProc	yes	amsPrintAliases	A_formatterId A_cellViewId	'VLOG_INSTANCES
instanceProc	yes	amsPrintInstance	A_formatterId A_cellViewId A_instanceId	'VLOG_INSTANCES
instanceMaster-NameProc	yes	amsPrintInstance-MasterName	A_formatterId A_cellViewId A_instanceId	'VLOG_INSTANCES
instanceParametersProc	yes	amsPrintInstance-Parameters	A_formatterId A_cellViewId A_instanceId	'VLOG_INSTANCES
instance-PortsProc	yes	amsPrintInstance-Ports	A_formatterId A_instanceId [x_iteration]	'VLOG_INSTANCES
attributesProc	yes	amsPrintAttributes	A_formatterId A_objectId	'VLOG_INSTANCES when objectId is an instanceId, 'SIGNAL_DECLARATION when objectId is wireId, 'PORT_DECLARATION when objectId is portId.

Table 2: Formatter fields representing the netlist procedures

This provided an overview of how the Verilog-AMS netlist is partitioned and when the various netlist procedures are available for customization in AMSDesigner. The next section will dwell into more details on the various levels of customization that a designer can use to control the generated netlist.

Note that only the following netlist procedures are supported in IC5.0.33 - in alignment with the

support for the parameter and instance objects:

Formatter field for which overrides are supported in IC5.0.33	Corresponding default netlist procedures
parametersProc	amsPrintParameters
instanceProc	amsPrintInstance
instanceMasterNameProc	amsPrintInstanceMasterName
instanceParametersProc	amsPrintInstanceParameters
instancePortsProc	amsPrintInstancePorts
attributesProc	amsPrintAttributes.

Table 3: Netlist Procedures and their overrides supported in IC 5.0.33

For more information, refer to [1] "[Cadence AMS Environment User Guide for IC 5.0.33.](#)", on page 21.

4.0 Levels of netlist customization based on requirements

The section "[3.1 Overview](#)", on page 2 provided an overview of the various levels of netlist customization available in AMSDesigner. They give the designer the flexibility to choose the type of customization that is best-suited for addressing the needs of a specific design/methodology requirements. The following sections will discuss the scenarios in which each of these customization capabilities is useful along with a suitable example.

4.1 Control the format of certain parts of the netlist

If the requirement is to enable simple customizations on the format of a netlist, then, the first place to look for performing this customization should be the `ams.env` variables. The section ‘Using `ams.env` Variables to Customize Netlists’ in [1] "[Cadence AMS Environment User Guide for IC 5.0.33.](#)", on page 21 provides a list of `ams.env` variables that can be used to format certain parts of the netlist.

For example, consider the situation where the designer needs to generate the name of an iterative instances in the following format:

```
‘<instance_base_name>_<index>_’
```

This can be accomplished by setting the value of ‘`iterInstExpFormat`’ `ams.env` variable as follows:

```
amsDirect.vlog      iterInstExpFormat      string      ““%b_%i_””
```

With this setting, an iterative instance of ‘nand2<0:3>’ will look like:

```
nand2
(* integer library_binding = "sample";
   integer elaboration_binding = "nand2[0:3]"; *)
nand2_0_ ( .Y( net43[0] ), .B( en ), .A( d[0] ) ),
nand2_1_ ( .Y( net43[1] ), .B( en ), .A( d[1] ) ),
nand2_2_ ( .Y( net43[2] ), .B( en ), .A( d[2] ) ),
nand2_3_ ( .Y( net43[3] ), .B( en ), .A( d[3] ) );
```

More powerful formatting capabilities will be addressed via custom netlist procedures in ["4.3 Control both the data and the format of the netlist, including powerful formatting capabilities"](#), on page 11

4.2 Control the data in the netlist but not its format

If the requirement is to modify the data in the netlist, but not its format, then, the designer can use netlist procedures to modify the data as needed and subsequently call the default netlist procedures to generate the netlist in the default format.

For example, consider the situation where a designer needs to model the narrow width effects of a mosfet device accurately by accounting for the number of fingers on the mosfet during simulation. As the same models are used for both pre- and post-layout, the scaling needed to adjust for the differing numbers of fingers cannot be done in the models; it must be done during netlisting. This can be achieved using a netlist procedure that will customize the model parameters(data) as required and print them in the default netlisting format.

The number of fingers for each instance can be specified as a user property in the Edit Object Properties form. [Figure 2: "Edit Object Properties Form for instance M0 of cell ‘nmos’"](#), on page 10 specifies the value of ‘fingers’ user property as ‘10’.

Here is a custom netlist procedure that uses the value of the ‘fingers’ property to modify the values of relevant parameters for every instance of ‘nmos’:

```
(defun myPrintInstanceParameters (formatterId cvId instanceId)
;; Modify the value of relevant parameters of “nmos”
  (if (instanceId->masterName == "nmos") then
    numfingers = instanceId->id->fingers
    (if (numfingers != nil) then
      (foreach param instanceId->parameters
        (if ((param->name == "w") || (param->name == "as") ||
            (param->name == "ad") || (param->name == "ps")) ||
```

```

(param->name == "pd" ) then
  param->value = strcat(param->value "/" numfingers)
); if
(if (param->name == "m") then
  param->value = strcat(param->value "*" numfingers)
); if
);foreach
);if
); if
;; Call the default instance netlist procedure
amsPrintInstanceParameters(formatterId cvId instanceId)
); defun

```

The screenshot shows a dialog box titled "Edit Object Properties" with several sections:

- Buttons:** OK, Cancel, Apply, Defaults, Previous, Next, Help.
- Apply To:** only current instance
- Show:** system user CDF
- Property Table:**

Property	Value	Display
Library Name	analogLib	off <input type="checkbox"/>
Cell Name	nmos	off <input type="checkbox"/>
View Name	ams	off <input type="checkbox"/>
Instance Name	M0	off <input type="checkbox"/>
- User Property Table:**

User Property	Master Value	Local Value	Display
fingers		10	both <input type="checkbox"/>

Figure 2: Edit Object Properties Form for instance M0 of cell 'nmos'

The custom netlist procedure modifies the value of parameters and calls the default netlisting procedure 'amsPrintInstanceParameters' to print them in the default format. The custom netlist procedure can be set up by overriding the appropriate formatter field as shown below:

```

;; =====
;; Set up area
;; =====
netlistId = amsGetNetlist()
formatterId = netlistId->vlog
;; Override the printing of instance parameters netlist procedure
formatterId->instanceParametersProc = 'myPrintInstanceParameters

```

With this setting, the following instantiation statement is generated for instance 'M0' of 'nmos':

```

nmos #(.ps(11.6u/10), .as(1.32E-12/10), .l(130.0n), .pd(9.8u/
10), .ad(1.14E-12/10), .w(6u/10), .m(2*10)) (* integer
library_binding = "analogLib"; integer passed_mfactor = "m";
*) M0 ( net20, net9, cds_globals.gnd! , bulk_n_gnd! );

```

The above example illustrates how the data in the netlist can be customized without any worries about the format of the netlist.

4.3 Control both the data and the format of the netlist, including powerful formatting capabilities

In some cases, the requirement is to modify not only the data but also the format of the netlist. In general, the 'ams.env' variables provide limited formatting capability and may not meet the desired formatting requirements in all situations. Custom netlist procedures can be used in these cases to take advantage of its powerful formatting capabilities.

For example, consider a situation where the designer wants to monitor certain signals that are connected to an instance, at every clock transition, by specifying those signals in the schematic. The designer can create a user property, say 'CLOCK_MONITOR' on the instance whose value will be a list of signals that need to be monitored. The user can then write a custom netlist procedure that generates Verilog-AMS behavioral code to monitor these signals.

Here is an Edit Object Properties form for instance 'R0' of a resistor component that has a user

property 'CLOCK_MONITOR' with a value of 'in_d'.

The screenshot shows the 'Edit Object Properties' dialog box. At the top, there are buttons for 'OK', 'Cancel', 'Apply', 'Defaults', 'Previous', 'Next', and 'Help'. Below these are 'Apply To' options: 'only current' and 'instance'. There are also 'Show' checkboxes for 'system', 'user', and 'CDF'. A 'Browse' button and a 'Reset Instance Labels Display' button are also present. The main area contains a table with columns 'Property', 'Value', and 'Display'. The rows are: 'Library Name' with value 'analogLib' and 'Display' 'off'; 'Cell Name' with value 'res' and 'Display' 'off'; 'View Name' with value 'symbol' and 'Display' 'off'; 'Instance Name' with value 'R0' and 'Display' 'off'. Below this table are buttons for 'Add', 'Delete', and 'Modify'. At the bottom, there is a table for 'User Property' with columns 'Master Value', 'Local Value', and 'Display'. The row for 'CLOCK_MONI..' has a redacted Master Value, a Local Value of 'in_d', and a Display of 'both'.

Figure 3: Edit Object Properties Form for instance 'R0' of cell 'resistor'

Here is an example custom netlist procedure that generates Verilog-AMS behavioral code to display the value of signals specified in the 'CLOCK_MONITOR' user property.

```
(defun myInstanceSignalMonitor (formatterId cellviewId instanceId)
  ;; Call the default instance procedure
  amsPrintInstance(formatterId cellviewId instanceId)
  (let (lcv)
    ;; Check for property CLOCK_MONITOR on the instance
    (when instanceId->id->CLOCK_MONITOR
      (setq signal_list (parseString instanceId->id->CLOCK_MONITOR))
      monitor_signal = car(signal_list)
      print_debug_signals = 0
      (if (monitor_signal != nil) then
        amsPrint(formatterId "\n// Debug signals at every clock transition")
        amsPrint(formatterId "always @(posedge(clock) or negedge(clock))\n")
        amsPrint(formatterId "begin")
        amsPrint(formatterId " $display($stime, 'Signal values are:');")
        print_debug_signals = 1
      ); if
    ;; Display the listed signals
```

```

    (while (monitor_signal != nil)
      display_signal = strcat(" $display(" monitor_signal ": %b', " monitor_signal "); ")
      amsPrint(formatterId display_signal)
      signal_list = cdr(signal_list)
      monitor_signal = car(signal_list)
    )
    (if (print_debug_signals == 1) then
      amsPrint(formatterId "\nend \n")
    ); if
  ); when
); let
); defun

```

The netlist procedure first prints the instantiation statement using the default netlist procedure ‘amsPrintInstance’. It then queries the user property ‘CLOCK_MONITOR’ and generates Verilog-AMS behavioral code to display the value of specified signals at every clock transition. The custom netlist procedure can be set up by overriding the appropriate formatter field as shown below:

```

;; =====
;; Set up area
;; =====
netlistId = amsGetNetlistId()
formatterId = netlistId->vlog
;; Override the printing of instance netlist procedure
formatterId->instanceProc = 'myInstanceSignalMonitor

```

With this setting, the following statements are generated for instance ‘R0’ of ‘resistor’:

```

resistor #(.r(1K)) (* integer library_binding = "analogLib"; *)
              R0( in_d, net20 );
// Debug signals at every clock transition
always @(posedge(clock) or negedge(clock))
begin
  $display($stime, "Signal values are:");
  $display("in_d: %b", in_d);
end

```

The above example illustrated how the netlist format can be customized using ‘amsPrint’ helper functions. Helper functions are also available to generate information(amsInfo), warning(amsWarning) and error(amsError) messages during netlisting. For more information, refer to [1] "[Cadence AMS Environment User Guide for IC 5.0.33.](#)", on page 21.

4.4 Control the data/format for instances of specific cells in the netlist.

The previous two sections had examples where custom netlist procedures were set up using global overrides via the formatter field. In general, a given cellview(CBN) may contain instances of several components. If the requirement is to control the customization of instances of a particular instance master, then, using global instance netlist procedure overrides may not be suitable or intuitive.

In some cases, it is desirable to keep the custom netlist procedure always with the cell for re-use with other designs. A global instance override for a particular design may not bode well with a different global instance override for a different design. In these cases, the 'netlistProcedure' field in the CDF->simInfo form for 'ams' simulator can be used to specify the instance netlist procedures for a given cell. This ensures that the information associated with the cell(including the CDF 'nelistProcedure' field setting) is maintained across designs. For information about syntax used in the netlistProcedure field, refer to [1] "[Cadence AMS Environment User Guide for IC 5.0.33.](#)", on page 21.

Typically, a symbol in the schematic cellview results in an instantiation statement in the CBN netlist. Consider the situation where the designer wants to use a symbol in the schematic cellview to represent Verilog debugging code that is specified in a file, say 'verilog.v' at a known location. The example in [Figure 4: "Edit Object Properties Form for instance 'I1' of cell 'verinc'"](#), on page 15 specifies a 'VERILOG_INCLUDE' user property whose value is a 'library/cell/view' where such a 'verilog.v' file is located.

Here is an example custom netlist procedure that queries the 'VERILOG_INCLUDE' property on a given instance, uses its value to locate the 'verilog.v' file and prints the contents of the file into the Verilog-AMS netlist.

```
(defun myInstanceVerilogInclude (formatter cellview inst)
  (let (lcv file)
    ;; Check for property VERILOG_INCLUDE on the instance
    ;;
    (when inst->id->VERILOG_INCLUDE
      (setq lcv (parseString inst->id->VERILOG_INCLUDE))
      ;; Get the the default file name verilog.v
      (setq file (ddGetObj (car lcv) (cadr lcv) (caddr lcv) "verilog.v"))
      ;; Read in the file and print the contents
      (if (null file)
          (progn
            sprintf(errmsg "Expected a verilog.v in %s:%s.%s\n"
                      (car lcv) (cadr lcv) (caddr lcv) )
            (amsError formatter errmsg)
          )
        )
    )
  )
```

```

(prog (filePort lineBuffer)
  ;; Open the file, and start printing contents of the file
  (setq filePort (infile file->readPath))
  (while (gets lineBuffer filePort)
    (amsPrint formatter lineBuffer)
  ); while
  (close filePort)
); prog
); if
); when
); let
); defun

```

Property	Value	Display
Library Name	NetlistLib	off
Cell Name	verinc	off
View Name	symbol	off
Instance Name	I1	off

User Property	Master Value	Local Value	Display
VERILOG_INC..		NetlistLib verir	both

Figure 4: Edit Object Properties Form for instance 'I1' of cell 'verinc'

As the netlist customization is applicable only for instances of cell 'verinc', the custom netlist procedure can be set up using the 'netlistProcedure' field in the CDF->simInfo form for cell 'verinc' as show in [Figure 5: "Edit Simulation Information Form for cell 'verinc'", on page 16.](#)

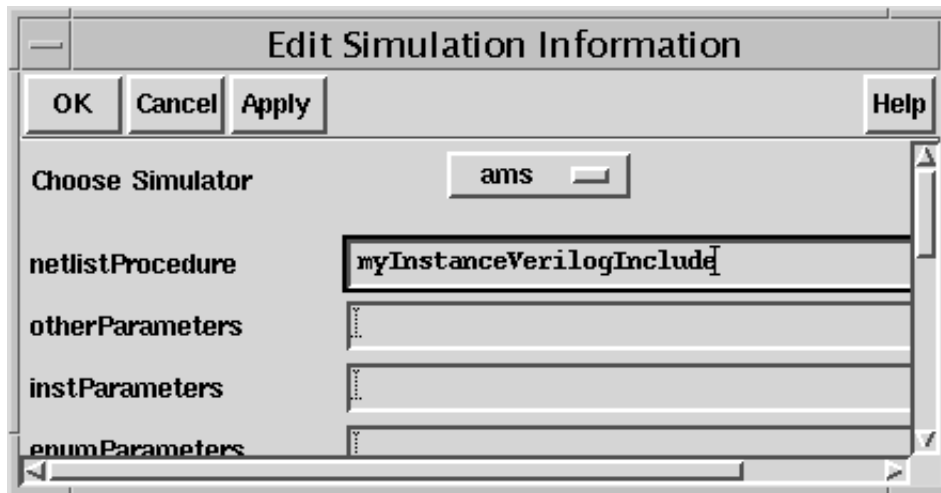


Figure 5: Edit Simulation Information Form for cell ‘verinc’

With this setting, the following statements are generated in the netlist for instance ‘I1’ of cell ‘verinc’:

```
// --- begin included file ---// Design debugger/monitor
parameter TCOff=0;`ifdef CHECK_INPUT_TRANSITIONS
always @(posedge(TCOff||in_d) or negedge(TCOff||in_d))
    if (eval==1'b1)
        $display($stime," WARNING: %m in_d transition (evaluate is
active)");
always @(posedge(TCOff||out_y) or negedge(TCOff||out_y))
    if (eval==1'b1)
        $display($stime," INFO: %m out_y transition (evaluate is
active)");
`endif// ---- end of included file ----
```

The above example illustrated the mechanism to specify custom netlist procedures and control them on a cell by cell basis. This can be particularly useful in situations where the custom netlist procedure needs to exist along with the cell for reuse between designs.

4.5 Levels of precedence

The examples in the above two sections ["4.3 Control both the data and the format of the netlist, including powerful formatting capabilities"](#), on page 11 and ["4.4 Control the data/format for instances of specific cells in the netlist."](#), on page 14 show that there are two ways to customize the netlisting of an instantiation statement:

1. Globally set the netlisting of various parts of the instance using one or more of the instance

netlisting overrides.

2. Use the 'netlistProcedure' field in 'CDF->simInfo->ams' to customize the netlisting of various parts of the instance.

Note that while both the customizations can co-exist in the design, only one of them will be used for customizing the instance in the netlist. When a custom netlist procedure is specified in the 'netlistProcedure' field of simInfo for a given instance master(cell), it will precede any global override for that instance.

Consider the examples in the previous two sections. Lets assume that the instances 'R0' in section ["4.3 Control both the data and the format of the netlist, including powerful formatting capabilities", on page 11](#) and 'I1' in section ["4.4 Control the data/format for instances of specific cells in the netlist.", on page 14](#) exist in the same cellview(CBN). Assume that the global instance override is set via:

```
;; Override the printing of instance netlist procedure  
formatterId->instanceProc = 'myInstanceSignalMonitor
```

and the 'netlistProcedure' field of CDF->simInfo for cell 'verinc' is set to 'myInstanceVerilogInclude' as shown in [Figure 5: "Edit Simulation Information Form for cell 'verinc'", on page 16](#).

In this case, during netlisting, when the instance 'R0' is encountered, the global netlist procedure override 'MyInstanceSignalMonitor' will be used to netlist that instance. This is because the cell 'resistor' does not have a netlist procedure specified in its CDF->simInfo 'netlistProcedure' field.

When the instance 'I1' is encountered, the cell-specific netlist procedure 'MyInstanceVerilogInclude' will be used to netlist that instance. The global netlist procedure override will not affect this instance. In the absence of both the override mechanisms, the instance will be netlisted using the default instance netlist procedure i.e 'amsPrintInstance'. Precedence levels are thus established to provide finer control over the customization of various instances.

4.6 Dynamic simInfo

The CDF->simInfo form for a given cell has several fields that specify various simulator-specific information for that cell. In some cases, it is desirable to customize the values of these fields(which are on a cell level) for different instances of the same cell. This can be achieved by setting the value of the field to a custom netlist procedure. The custom netlist procedure can then process instance-specific data to determine the final value of the field, which will be used in the same manner as the value of a normal simInfo field. This technique where the value of the simInfo field is determined dynamically, based on instance-specific information, is called 'dynamic simInfo'. It is unique to the AMSDesigner solution.

Consider an example, where the designer would like to establish ‘dynamic’ inherited connections on programmable nodes of pcells i.e the names and values of the inherited connections are calculated from object properties and can vary for each instance. This capability allows each of the instances within a single schematic to have different inherited connections.

The following netlist procedure can be used to process instance properties and determine the inherited connections unique to that instance:

```
(defun AMSnmos_dnw_inhExtraTerminals (inst "g")
  (let
    ;; Default values
    ( (term1 '(nil name "B" direction "inputOutput" netExpr)
      (netExpr1 "[@vbulk_n:%:vssa!]" )
      (term2 '(nil name "DNW" direction "inputOutput" netExpr)
        (netExpr2 "[@vdnw:%:not_set!]" )
        (term3 '(nil name "SUB" direction "inputOutput" netExpr)
          (netExpr3 "[@vsub:%:not_set!]" )
        )
      )
    ;; Override values, if any.
    (if (inst != nil) then
      (if (inst->bulkNode != "") then
        netExpr1 = (strcat "[@" inst->bulkNode ":%:" inst->bulkNode "]")
      )
      (if (inst->dnw_node != "") then
        netExpr2 = (strcat "[@" inst->dnw_node ":%:" inst->dnw_node "]")
      )
      (if (inst->sub_node != "") then
        netExpr3 = (strcat "[@" inst->sub_node ":%:" inst->sub_node "]")
      )
    )
    ;; Generate the dynamic "extraTerminals" list.
    extraTerminals = (list (append1 term1 netExpr1)
      (append1 term2 netExpr2)
      (append1 term3 netExpr3) )
  );let
);defun
```

The ‘extraTerminals’ field in CDF->simInfo->ams is used to specify information about inherited connections of a cell for the AMS simulator. This field will now contain the name of the netlist procedure as shown in [Figure 6: "Edit Simulation Information for cell ‘nmos_dynamic_inh’"](#), on [page 19](#).

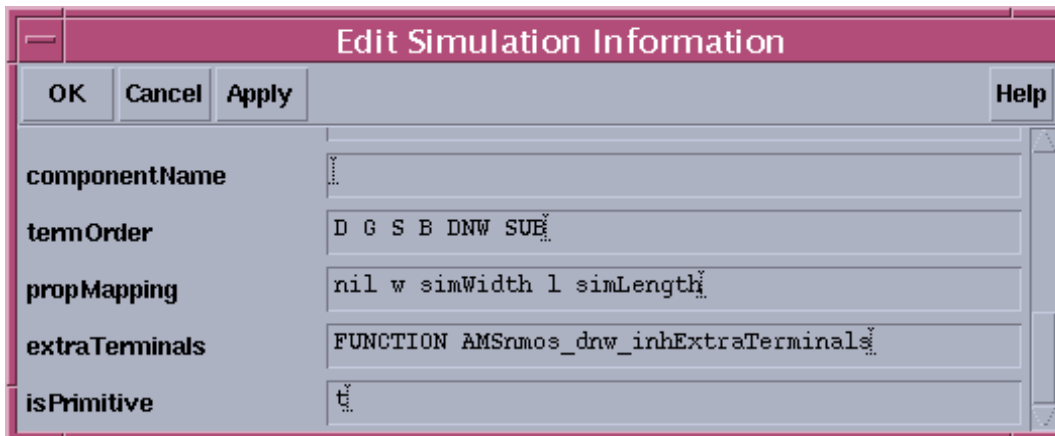


Figure 6: Edit Simulation Information for cell ‘nmos_dynamic_inh’

Consider an instance ‘M3’ of cell ‘nmos_dynamic_inh’ with the object properties set up as shown in [Figure 7: "Object properties for instance ‘D0’ of cell ‘nmos_dynamic_inh’", on page 19.](#)



Figure 7: Object properties for instance ‘D0’ of cell ‘nmos_dynamic_inh’

With this setting, during netlisting, the netlist procedure specified in the ‘extraTerminals’ field is executed for instance ‘M3’ to determine its final value of the ‘extraTerminals’ field. The final value for this setting would look like:

```
((nil name "B" direction "inputOutput" netExpr "[@VPOS!:%:VPOS!]")
(nil name "DNW" direction "inputOutput" netExpr "[@dnw!:%:dnw!]")
(nil name "SUB" direction "inputOutput" netExpr "[@sub!:%:sub!]"))
```

The netlister then uses this final value of ‘extraTerminals’ field to generate inherited connections for instance ‘M3’ in the netlist. The corresponding inherited connection declarations are also generated in the netlist. Here are the relevant statements from the netlist:

```

wire    (* integer inh_conn_prop_name="\sub! ";
         integer inh_conn_def_value="cds_globals.\sub! "; *)
        \sub!_sub! ;

wire    (* integer inh_conn_prop_name="\dnw! ";
         integer inh_conn_def_value="cds_globals.\dnw! "; *)
        dnw!_dnw! ;

wire    (* integer inh_conn_prop_name="\VPOS! ";
         integer inh_conn_def_value="cds_globals.\VPOS! "; *)
        \VPOS!_VPOS! ;

nmos_dynamic_inh #(.as(4.64E-12), .ps(1.664E-05), .m(5),
                  .ad(4.64E-12), .pd(1.664E-05), .l(3.2E-07), .w(1E-05))
  (* integer library_binding = "testLib";
   integer passed_mfactor = "m"; *)
  M3 ( D3, G, S, \VPOS!_VPOS! , \dnw!_dnw! , \sub!_sub! );

```

The above example illustrates an application of custom netlist procedures to dynamically compute the value of 'extraTerminals' field of simInfo based on instance specific information. Similar approaches can be used to dynamically compute the value any simInfo field. Note that this type of customization is unique to AMSDesigner.

["8.0 Appendix", on page 21](#) contains the complete customized netlist for the various examples discussed in this paper.

5.0 Future Work

The remaining data objects specified in ["3.3 Data Objects for Netlisting in AMSDesigner", on page 5](#), netlist procedures specified in [Table 2: "Formatter fields representing the netlist procedures", on page 7](#) and related helper functions will be supported in a future release.

6.0 Conclusion

This paper described the support for custom netlist procedures in AMSDesigner along with various example scenarios illustrating the various levels of customization. These capabilities give the designer the flexibility to choose the level of customization truly based on the requirements of the design/methodology, without adding any additional complexity. Helper functions are available to help the user with writing custom netlist procedures when the requirements are for generating a completely customized netlist. The various examples in the paper illustrate the power of netlisting as well as the scope of various customization capabilities so that the designer can choose the most appropriate method for meeting his/her design methodology requirements.

7.0 References

- [1] Cadence AMS Environment User Guide for IC 5.0.33.
- [2] "Verilog-AMS Language Reference Manual V2.0", Open Verilog International, Jan 2000.
- [3] SKILL Language User Guide, Product Version 06.10, September 2002.

8.0 Appendix

Example1:

Refer to [Figure 8: "Schematic for example #1", on page 23](#). This figure shows a schematic cellview for 'top_ver_inc'. Parts of this example have been used to describe various examples in the following sections:

- ["4.2 Control the data in the netlist but not its format", on page 9](#)
- ["4.3 Control both the data and the format of the netlist, including powerful formatting capabilities", on page 11](#)
- ["4.4 Control the data/format for instances of specific cells in the netlist.", on page 14](#)
- ["4.5 Levels of precedence", on page 16](#)

Here is the complete netlist generated for the cellview 'top_ver_inc':

```
// Verilog-AMS netlist generated by the AMS netlister, version xx.  
// Cadence Design Systems, Inc.
```

```
`include "disciplines.vams"  
  
module top_ver_inc ( clock,eval,input_signal );  
  
input  clock;  
input  eval;  
input  input_signal;  
wire  
    (* integer inh_conn_prop_name="bulk_n";  
      integer inh_conn_def_value="cds_globals.\\gnd! "; *)  
    \bulk_n_gnd! ;  
  
and4 (* integer library_binding = "NetlistLib"; *) I9  
    (.A( eval ), .D( in_d ), .C( clock ), .Y( out_y ), .B(  
input_signal ) );  
// Debug signals at every clock transition  
always @(posedge(clock) or negedge(clock))  
begin
```

```

    $display($stime, "Signal values are:");
    $display("input_signal: %b", input_signal);
    $display("clock: %b", clock);
end

vsource #(.type("sine")) (* integer library_binding =
"analogLib"; *) V1 ( net9, cds_globals.nd! );

resistor #(.r(1K)) (* integer library_binding = "analogLib"; *)
R0 ( in_d, net20 );
// Debug signals at every clock transition
always @(posedge(clock) or negedge(clock))
begin
    $display($stime, "Signal values are:");
    $display("in_d: %b", in_d);
end

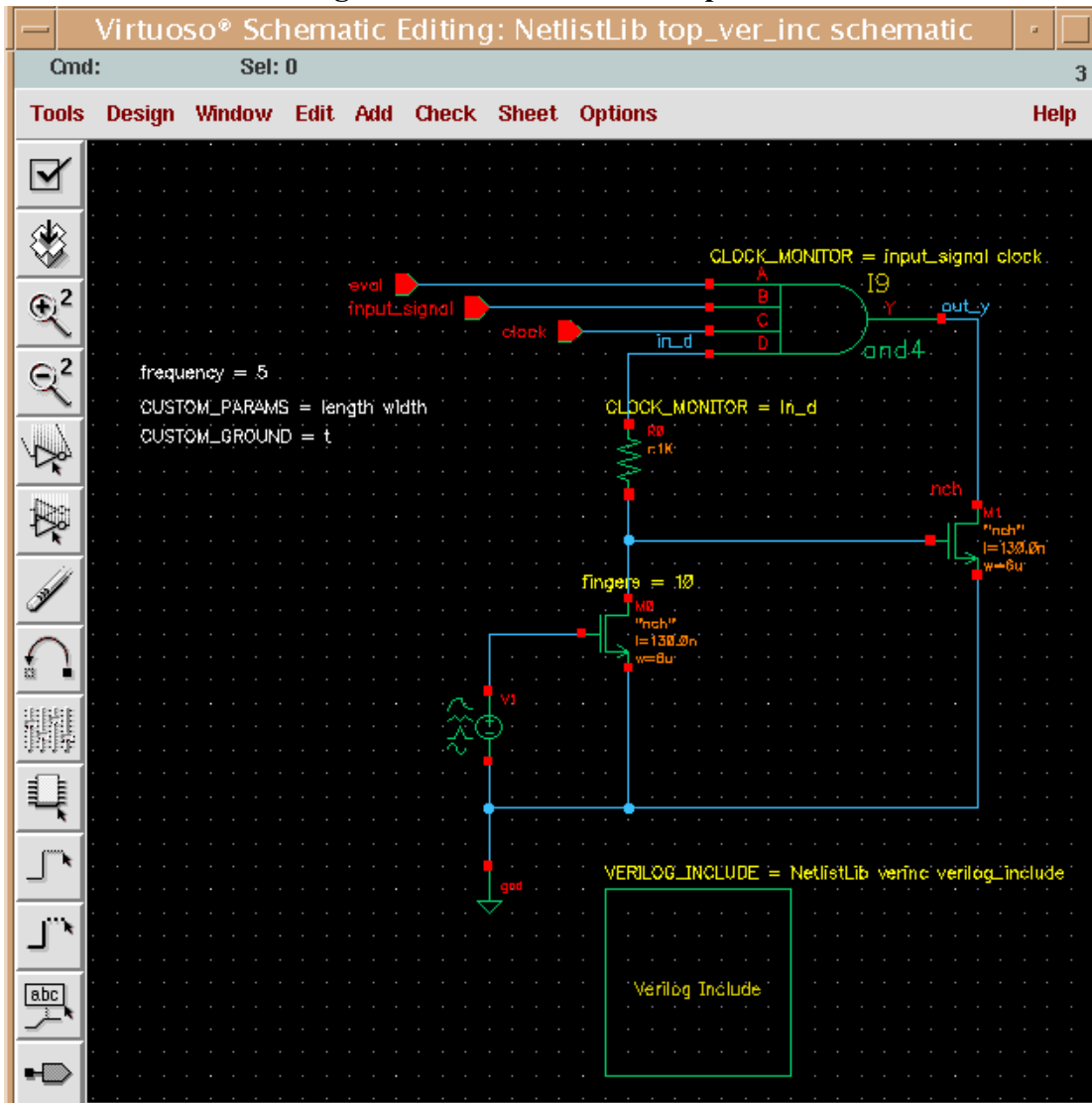
nch #(.as(2.04E-12), .ps(1.268u), .m(2), .ad(2.04E-12),
.pd(12.68u), .w(6u), .l(130.0n))
(* integer library_binding = "analogLib";
integer passed_mfactor = "m"; *)
M1 ( out_y, net20, cds_globals.gnd! , \bulk_n_gnd! );

nch #(.as(1.32E-12/10), .ps(11.6u/10), .m(2*10), .ad(1.14E-12/
10), .pd(9.8u/10), .w(6u/10), .l(130.0n))
(* integer library_binding = "analogLib";
integer passed_mfactor = "m"; *)
M0 ( net20, net9, cds_globals.gnd! , \bulk_n_gnd! );

// --- begin of included file ---
// Design debugger/monitor
parameter TCOff=0;
`ifdef CHECK_INPUT_TRANSITIONS
always @(posedge(TCOff||in_d) or negedge(TCOff||in_d))
    if (eval==1'b1)
        $display($stime," WARNING: %m in_d transition while evaluate
is active");
always @(posedge(TCOff||out_y) or negedge(TCOff||out_y))
    if (eval==1'b1)
        $display($stime," INFO: %m out_y transition while evaluate is
active");
`endif
// ---- end of included file ----

```

Figure 8: Schematic for example #1



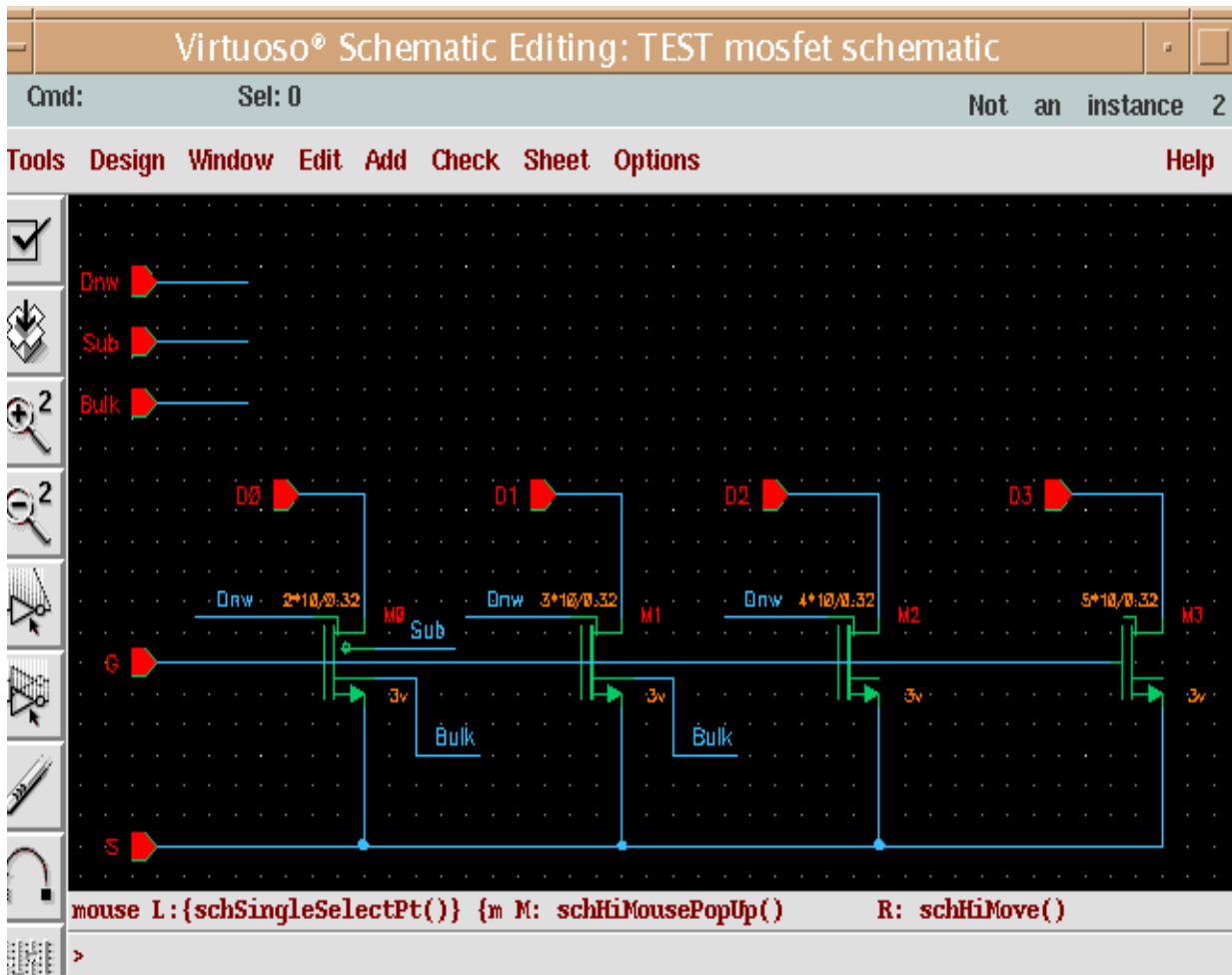


Figure 9: Schematic for example #2

Example 2:

Refer to [Figure 9: "Schematic for example #2", on page 24](#). This figure shows the schematic for the cell 'mosfet' which has 4 instances of a nmos device, each with programmable nodes which can be used to generate instance-specific inherited connections. Section ["4.6 Dynamic simInfo", on page 17](#) describes the properties on instance M3 from this schematic. The various values for programmable nodes in their respective instance object properties are summarized in [Table 4: "Values of programmable nodes in Edit Object Properties Form for instances M0, M1, M2 and M3.", on page 25](#).

Programmable node	M0	M1	M2	M3
Bulk node connection			bulk!	VPOS!
Substrate connection		sub!	sub!	sub!

Programmable node	M0	M1	M2	M3
Deep NWell connection				dnw!

Table 4: Values of programmable nodes in Edit Object Properties Form for instances M0, M1, M2 and M3.

Here is the complete generated netlist for the cellview 'mosfet':

```
// Verilog-AMS netlist generated by the AMS netlister, version
none.
```

```
// Cadence Design Systems, Inc.
```

```
`include "disciplines.vams"
```

```
module mosfet ( G,Bulk,D3,S,D1,Sub,Dnw,D0,D2 );
```

```
input  G;
input  Bulk;
input  D3;
input  S;
input  D1;
input  Sub;
input  Dnw;
input  D0;
input  D2;
```

```
wire
```

```
  (* integer inh_conn_prop_name="\sub! ";
     integer inh_conn_def_value="cds_globals.\sub! "; *)
  \sub!_sub! ;
```

```
wire
```

```
  (* integer inh_conn_prop_name="\dnw! ";
     integer inh_conn_def_value="cds_globals.\dnw! "; *)
  \dnw!_dnw! ;
```

```
wire
```

```
  (* integer inh_conn_prop_name="\VPOS! ";
     integer inh_conn_def_value="cds_globals.\VPOS! "; *)
  \VPOS!_VPOS! ;
```

```
wire
```

```
  (* integer inh_conn_prop_name="\bulk! ";
     integer inh_conn_def_value="cds_globals.\bulk! "; *)
  \bulk!_bulk! ;
```

```

nmos_dynamic_inh #(.as(4.64E-12), .ps(1.664E-05), .m(5),
.ad(4.64E-12), .pd(1.664E-05), .l(3.2E-07), .w(1E-05))
(* integer library_binding = "testLib";
integer passed_mfactor = "m"; *)
M3 ( D3, G, S, \VPOS!_VPOS! , \dnw!_dnw! , \sub!_sub! );

nmos_dynamic_inh #(.as(4.64E-12), .ps(1.664E-05), .m(2),
.ad(4.64E-12), .l(3.2E-07), .pd(1.664E-05), .w(1E-05))
(* integer library_binding = "testLib";
integer passed_mfactor = "m"; *)
M0 ( D0, G, S, Bulk, Dnw, Sub );

nmos_dynamic_inh #(.as(4.64E-12), .ps(1.664E-05), .m(4),
.ad(4.64E-12), .pd(1.664E-05),.l(3.2E-07), .w(1E-05))
(* integer library_binding = "testLib";
integer passed_mfactor = "m"; *)
M2 ( D2, G, S, \bulk!_bulk! , Dnw, \sub!_sub! );

nmos_dynamic_inh #(.as(4.64E-12), .ps(1.664E-05), .m(3),
.ad(4.64E-12), .pd(1.664E-05), .l(3.2E-07), .w(1E-05))
(* integer library_binding = "testLib";
integer passed_mfactor = "m"; *)
M1 ( D1, G, S, Bulk, Dnw, \sub!_sub! );

endmodule

```