paramset: A Verilog-A/MS Implementation of Spice .model Statements

Ken Kundert
Designer's Guide Consulting, Inc.

Version 5, December 2004

The promise that comes from supporting compact modeling in Verilog-A/MS is the release of the designers from the tyranny of proprietary models. However, that promise is only half satisfied if we only support the model equations in Verilog-A/MS. While having a industry standard language for expressing model equations is critically important, it does not address the problem of proprietary .model files. To completely fulfill the promise, we must provide non-proprietary industry standard equivalents to all of the capabilities currently used in SPICE .model files. That includes support for the .model statements themselves, plus support for process corners, Monte Carlo analysis, etc.

This proposal is geared at providing the capabilities in Verilog-A/MS needed to replace the SPICE .model files, but in a way that is substantially more flexible.

This document contains the original paramset proposal that was used as the starting point for the new paramset capability in Verilog-AMS vers. 2.2 (available from www.verilog-ams.com). This proposal differs somewhat from what was eventually implemented in the standard and so should only be used for guidance on what was originally intended.

Last updated on May 16, 2006. You can find the most recent version at www.designers-guide.org. Contact the author via e-mail at ken@designers-guide.com.

1.0 SPICE .model statements

SPICE provides the venerable .model statement to allow users to specify parameters that would be common to many components once in one place. Generally it is used for semiconductor components, where there are a large number of process parameters that are shared with all components of a certain type. On the individual instances of these components, the user would generally only give geometrical parameters (such as width and length). For efficiency, SPICE allows a .model statement to be associated with only one type of component and hard-codes the parameters that it may take. That way the component models may be compiled to expect parameters to be stored in a common place, decreasing evaluation time and increasing storage efficiency.

Increasingly situations arise that the .model statement is not capable of handling efficiently. The causes of the limitations are described next.

1.1 Inflexible partition between instance and model parameters

The implementor of the model decides a priori which parameters should be instance parameters and which should be model parameters. This works in most cases, but the increasing need for Monte Carlo analysis is causing problems. Monte Carlo analysis generally requires a different partition. Typically one enters the ‘as drawn’ geometry parameters on the instance, and then the parameters that are used to convert from the ‘as drawn’ to the ‘effective’ geometry on the .model. Unfortunately, with Monte Carlo, you want to model the instance-to-instance variation of things like undercut, which is described using model parameters. This need for model parameters to vary on a per-instance basis implies that the .model data structures must be duplicated for each instances, with only a small number of parameters differing between all of the structures. This is tremendously expensive in terms of storage. It also adversely affects the simulators initialization time and cache performance. With Monte Carlo, one would like to re-partition the parameters, making some of the model parameters instance parameters.

1.2 Parameters shared between many types of components

There are situations where users would like to specify parameters once to be shared between many different types of components. This is true in electrical circuits, but has not been a burning issue because there are generally only a small number of basic component types, so it is not a burden to duplicate the shared parameters. However, there are some situations where the number of component types that need the shared parameters is very large and it would be too much of a burden to copy the shared parameters into model statements for every type of component that would possibly be used. There are two examples of this. First is the substrate parameters for distributed interconnect models. Here the shared information is the layer thicknesses, material properties of the substrate, etc. The various models that might be supported include lines, bends, tees, crosses, couplers, spirals, splitters, etc. The second example is MEMS. Again, it the substrate parameters that need to be shared, but this time it includes the mechanical and perhaps thermal properties of the substrate, along with the electrical and common geometrical properties of the substrate needed by the distributed components. The type of MEMS components include beams, gaps, anchors, and plates.

1.3 Model levels

As a variation of the above, one might want to implement a collection of distinct but related modules. Consider providing a family of MOS modules that all use the same parameters, but are for different applications. One might provide a simple model for digital circuits, and more accurate model for analog circuits, and a comprehensive model for RF circuits. Currently, these would all have to be implemented in a single model today if they are to share a model statement.

1.4 Binning

With binning, users would like to automatically select the model statement to be used based on the value of various instance parameters. In this case, multiple model statements must have the same name and must include a mechanism that will distinguish between the model statements based on the values given for instance parameters.

1.5 Hierarchy

Recently, important effects that need to be modeled have been identified only after the models have been released, and the groups that are responsible for defining the models have been unable to enhance the models in a timely manner. For this reason, the models have been supplemented by combining a collection of components into a model defined hierarchically (as a subcircuit). In SPICE, subcircuits cannot accept model statements, and so a subcircuit must be defined for every set of model statements that might be needed to implement a distinct hierarchical model. While this is not a big problem, it does represent extra work for the modeling group, especially when it comes time to update the structure of the model.

2.0 Proposal

In this proposal, the concept of a model statement is introduced into Verilog-A/MS, but in a way that is inherently more powerful than the approach used in SPICE. The goal is to develop a set of extensions to Verilog-A/MS so that all of the fundamental capabilities provided by way of a SPICE model file today can be provided by Verilog-A/MS.

2.1 Paramset

The proposal assumes that the model equations are contained in a module for which no distinction is made between instance and model parameters. The SPICE model statement will be formulated in a more Verilog-like syntax and renamed *paramset*. The name has been changed from *model* to try to reduce the confusion that results when talking about the model definition and the model parameters[1]. In addition, the SPICE model statement is substantially enhanced by adding the concept of user defined parameters for the paramset. These parameters will play the role of instance parameters.

1. The name *paramset* can be changed, however using *model* would create considerable confusion because the parameters of the paramset become the instance parameters. If the name were changed from paramset to model, then the model parameters would become the instance parameters, which would be terribly confusing to SPICE users.

Each paramset will be associated with a particular module, and the parameters of the module are accessible within the paramset. The paramset itself may have parameters, which become instance parameters for those instances that reference the paramset. The parameters of the module would be assigned values in the paramset. They might be constants or constant expressions, as in Spice model cards. Or they could be functions of the parameters of the module or the paramset.

Assume that a behavioral MOS model is defined called BSIM6 and consider the following example:

```
paramset n180nm bsim6;
    parameter real l=0.18u;
    parameter real w=0.25u;

    .as = .ad = 1u*w;
    .ps = .pd = 2*(0.15u + w);
    .type = "n";
    .vto = 0.25;
    .kp = 20u;
    .tox = 100n;
    .nsub = 6.02e23;
    .xj = 4e–7;
    .vsat= 200k;
    ...
endparamset
```

In this example, I have defined a new composite statement in Verilog-A and Verilog-AMS called a *paramset*. It contains parameter and variable declarations and assignment statements. The first line of the statement contains the keyword *paramset*, the name of the paramset, and the name of the underlying module for the paramset. The opening line of the paramset is followed by the parameter declarations and then a series of statements. As mentioned previously, all of the module parameters are accessible from within the paramset. Any name preceded by a period (other that parameter names in argument lists) is considered a module parameter. The paramset itself may have local variables and parameters defined. The paramset can contain any type of statement that is allowed in functions. In particular, it may contain assignments, conditionals, and iterators.

To use a paramset, one simply uses the name of the paramset in lieu of the name of a module when instantiating an instance of the underlying module. So in this case, rather than using

```
l = 200n;
w=1u;
a = 1u*w;
p = 2*(0.15u + w);
bsim6 #(.l(l), .w(w), .as(a), .ad(a), .ps(p), .pd(p)) M1 (.d(n1), .g(n2), .s(n3), .b(n4));
```

one would use

```
n180nm #(.l(200n), .w(1u)) M1 (.d(n1), .g(n2), .s(n3), .b(n4));
```

The paramset parameters completely replace the module parameters (one cannot specify module parameters on an "instance" of a paramset).

With the proposal as it currently exists, we have a proposal that adds the concept of SPICE model parameters to Verilog-A/MS. In addition, it offers the following advantages:

1. Rather than a hard coded partition of the parameters between instance and model, the user can specify which parameters should be available to be specified on the instance statements.
2. Since the number of instance parameters can be minimized, the memory requirements are reduced.

Despite these advantages, the proposal so far does not address all of the issues described in Section 1.0. However, it can be enhanced. First consider the problem of defining a set of parameters that can be shared with devices of different type.

2.2 Constants Module

Consider the case of trying to model a system constructed from MEMS components. Here there are technology parameters that ideally would be shared between multiple components; parameters such as material and layer information (mechanical, thermal, and electrical properties). To implement this, assume that a module is created at the top-level of the design that contains only constants. Then, one could use Verilog's "out-of-module references," or OOMRs, to access these values from the various paramsets and modules that define the available component models.

As a simple example of how this might work, assume that the following module

```
module semico250nmCMOS;
    localparam real tox=100n;
    localparam real nsub=6.02e23;
    ...
endmodule;
```

is defined and is instantiated at the top level using

```
semico250nmCMOS process;
```

This creates a common place to place the technology parameters. They can be used through out the language, and particularly in paramsets and module definitions, by giving the full path to the constants. For example,

```
tox = process.tox;
```

This proposal use the *localparam* modifier as a way of indicating that the value is a constant. Localparams act like parameters in the sense that their value must be given when defined and that value cannot be changed within the module, but they cannot be set by passing values into the module when instantiating it. Parameters must be used because variables are not set at elaboration time. In addition, variables currently cannot be the target for OOMRs, because in other situations they could create ambiguous results since the value they export would depend on when the module was evaluated relative to when the OOMR was evaluated. The use of localparams is not required. One could instead use input parameters, which would allow their value to be overwritten when instantiated.

This same approach can be used to support Monte Carlo simulations. Here we assume that the random variables are defined in a block that externally looks like a constants module. Additional information would be needed to describe the statistics of the random

variables, such as variance, distribution, correlation, etc. Whether that block is defined inside the Verilog-A/MS or outside, the values can be accessed using a hierarchical name.

2.3 Output Variables

With the proposed extensions to Verilog-A/MS to support compact modeling, one can declare certain variables to be output variables by giving them a description. In other words, the act of giving a variable a description marks the variable as interesting and makes it available for output. This is done as follows,

```
real gm ---"Transconductance";
```

Once marked for output its name and value is now available to the simulator for inclusion into operating point reports, etc.

This same idea is also available within paramsets. Additional output variables can be defined within a paramset, and their values can be made functions of module and paramset parameters or of output variables from either the module or the paramset. This implies that it is possible to read the value of an output variable for the module from within the paramset, but it is not possible to write it. As such, a module output variable must not be the target of an assignment statement within a paramset. However, it is possible to declare a local variable within the paramset that has the same name as an output variable of the module, and in doing so the local variable will take the value of the output variable, but it can be locally overwritten. Whether the local variable is an output variable will depend on whether it has a description. So, this provides a mechanism for either hiding undesired module output variables or modifying the values and descriptions of the module's output variables.

The following example assumes that *gm*, *cpi*, and *cmu* are output parameters for the base module and the following statements are found within a paramset to create an additional output parameter *ft*. They declare *ft* as being output parameters for the paramset and then computes its value.

```
real ft ---"Transition frequency";
ft = .gm/(2*`M_PI*(.cpi+.cmu));
```

2.4 Binning

In order to support different versions of a model, such as might be the case when parameter binning is employed, make the following enhancements to the existing proposal

1. Assume that different paramsets with the same name, yet different parameter declarations, can coexist in the same design.
2. Assume that the parameters to the paramset can take range limits, and that if more than one paramset exists with the same name and parameters, range limits must be provided on the parameters so that the domain of each paramset can be distinguished.
3. That paramsets can hierarchically refer to other paramsets. In this way a paramset refers to a master, which may be either a module or another paramset.

The idea is that an instance will point to a "group" of paramsets (all of which have the same name), and that the paramset used can be determined by the value of the parame-

ters specified on the instance. Consider the following paramset, and assume that it is one of many with the same name, and that each has the same parameters, but different range limits for at least some of those parameters.

```
paramset n180nm bsim6;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    localparam real area=l*w from [0.0,5p);

    .as = .ad = 1u*w;
    .ps = .pd = 2*(0.15u + w);
    .type = “n”
    .vto = 0.25;
    .kp = 20u;
    .tox = 100n;
    .nsub = 6.02e23;
    .xj = 4e–7;
    .vsat= 200k;
    ...
endparamset
```

This is the same paramset given on page 4 except that range limits have been added to several parameters, and that a new parameter has been defined. In particular, range limits were added to *l* and *w* and a new localparam *area* has been added.

If an instance statement like the following were encountered

```
n180nm #(.w(1u)) m1 (.d(n1), .g(n2), .s(n3), .b(n4));
```

it would use the paramset given above because the value given for w=1μ is within the specified range for that parameter. If *w* were set to a value outside the range of this paramset, say to 30μ, then another paramset would be used. If no paramset with the name *n180nm* supports this value, an error is issued. In this way, binning is provided.

The localparam *area* shows how a more sophisticated form of binning is implemented. In this case, the selection is based not just on the values of individual parameters, but based on some function of multiple parameters. For example, the following instance

```
n180nm #(.l(0.25u), .w(25u)) m1 (.d(n1), .g(n2), .s(n3), .b(n4));
```

would not use the paramset given above because the area is not within the specified limits.

Following this approach directly might result in many of the same parameter values given in multiple paramsets. To avoid this, the paramsets can be arranged hierarchically, with the common parameters given in the common paramset, and only those parameters that are different between the various bins given on the top-level paramsets. Then, a paramset may refer to either a module or another paramset.

2.5 Model Levels

The current trend with models is to make them more accurate and more complete, but this also results in them being more expensive in terms of both the time required to evaluate them and the memory they require. To address this, the models themselves are becoming partitioned into multiple versions with a varying capabilities. A relatively

simple stripped down version might be used early in the design process or in less demanding parts of the circuit. More complicated and comprehensive versions would be used during the verification process, particularly on more sensitive circuits. In addition, one can imagine models being tailored for particular applications. There might be a version where the temperature effects are modeled in great detail that is preferred for bias circuits, and one in which the dynamic behavior is carefully modeled for high frequency or high speed applications.

There are two way in which one might want to support different versions of a model. In the first, the model itself is developed as a family of models, and it is simply a matter of passing in a parameter, such as a level parameter, into the model to indicate which version should be used. In the second, one might wish to use different models for each version. For example, one might want to use MOS0 for the simplest and fastest model and BSIM4 for the most complex. Of course, the user could do this today by specifying the master name for each instance such that they get the desired level, but such is a very burdensome and error prone process. Ideally, the user would give one master name that refers to a family of models, and the actual model used would depend on instance parameters, global parameters, etc. This can be done with paramsets if we extend them in the following way:

1. Assume that different paramsets with the same name can associated with different modules. For example, there could be two paramsets named nmos, one of which is associated with a module named MOS0 and the other is associated with a module named BSIM4.
2. Assume that the parameters to the paramset can take discrete values from a finite range. For example, the parameters could be of string type, and that the range could be a list of legal strings.
3. Assume that the default value for a parameter may fall outside the valid range given for that parameter.
4. Assume that when multiple paramsets are present that share the same name, the presence or absence of parameter can be used to determine which paramset is used.

Consider the situation where one wants to use either MOS0 or BSIM4 based on a parameter passed in by the user. In this case one might use ...

```
paramset n250nm mos0;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    parameter string vers="analog" from {"digital"};
    ...
endparamset

paramset n250nm bsim4;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    parameter string vers="analog" from {"analog"};
    ...
endparamset
```

Here we have two paramsets with the same name that both accept a parameter *vers* that takes a default value of *analog*. When the user creates an instance that refers to *n250nm*, then one of these two paramsets will be used depending on whether *vers* was given, and

if so, what value was used. If *vers* was given as analog, then the BSIM4 version is used as it is the only paramset that will accept *analog* as a legal value for *vers*. If *vers* is given as *digital*, then the MOS0 version is used. And if *vers* is not given, then again the BSIM4 version is used as it is the only paramset for which the default value for *vers* is a legal value.

Alternatively, one might want to choose which model to use based on the current phase of the design process, with the idea that simpler models are used early in the design process and more complete models used later. In this case one might use ...

```
module design;
    parameter string phase="initial design" from {"initial design", "final verification"};
    ...
endmodule;

paramset n250nm mos0;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    localparam string phase=design.phase from {"initial design"};
    ...
endparamset

paramset n250nm bsim4;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    localparam string phase=design.phase from {"final verification"};
    ...
endparamset
```

Finally, one might want to use a different paramset based on the presence or absence of parameters. For example, before layout one can generally just specify *w* and *l* for transistors, with things like *as*, *ad*, *ps*, and *pd* being approximated from *w* and *l*. However, after layout the actually values for *as*, *ad*, *ps*, and *pd* are known and should be accepted by the model. In this case, supporting the extra parameter increases the memory required to represent each instance, which acts to slow simulations and reduce the capacity of the simulator. One can avoid this extra expense when it is not needed using

```
paramset n180nm bsim6;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);

    .as = .ad = 1u*w;
    .ps = .pd = 2*(0.15u + w);
    ...
endparamset

paramset n180nm bsim6;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    parameter real as=50f from [0,inf);
    parameter real ad=50f from [0,inf);
    parameter real ps=1u from [0,inf);
    parameter real pd=1u from [0,inf);
    ...
```

```
endparamset
```

In this case, an instance for which a value was specified for either *as*, *ad*, *ps*, and *pd* would use the second paramset, all others would use the first.

2.6 Paramset Resolution Criteria

When providing several paramsets with the same name, it may happen that several would be suitable for use with a particular instance. This may occur when the parameters specified on an instance are compatible with multiple paramsets. An example would be if an instance of n180nm, whose paramsets are defined immediately above, were specified with only values given for *l* and *w*. In this case, the instance would be compatible with either of the two paramsets given above. A different situation would be if multiple paramsets supported the same parameters, but had range limits on those parameters whose ranges overlapped. In either case a single instance would be compatible in multiple paramsets. However, in the second case the developer of the paramsets has the ability to eliminate the ambiguity if so desired by eliminating the overlap in the range limits. It is not possible to avoid the ambiguity in the first case. As such, additional paramset resolution rules are enforced to allow the ambiguity to be eliminated if desired.

The paramset resolution algorithm is:

For each instance,

1. Find all paramsets for which
   a) the parameters overridden on the instance are parameters of the paramset
   b) the parameters of the paramset, with overrides and defaults, are all within the allowed ranges
   c) the localparams of the paramset, computed from parameters, are within the allowed range
2. choose the paramset which has the fewest number of un-overridden parameters
3. choose the paramset with the greatest number of range limited localparams

Even with this algorithm, there is no assurance that the ambiguities of the type given in the first case are eliminated. For example, an instance with a parameter α might be associated with two paramsets, the first accepts instances with parameters α and β, and the second with parameters α and γ. In this case, either paramset could be used. However the ambiguity can be eliminated if desired by adding a third paramset that only accepted instances with a parameter α.

2.7 Corners

Providing different model parameter sets for different process corners is possible using the concepts already presented. Consider adding a corner field to the *design* constants module defined above (page 10).

```
module design;
    parameter string corner="tt" from {"ss", "tt", "ff", "sf", "fs"};
    ...
endmodule;

paramset n250nm bsim4;
```

```
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    localparam string corner=design.corner from {“tt”};
    ...
endparamset
```

In this case, the paramset would be used because the value of *design.corner* is allowed by the range limit for the *corner* localparam in the paramset.

2.8 Paramsets of Paramsets

As defined so far, paramsets reference modules. Paramsets can also refer to other paramsets. In this way, on can define a base paramset and then use another paramset to refine it. For example, one could define a paramset for a MOS model that defines the traditional instance parameters, *w*, *l*, *as*, *ad*, *ps*, *pd*, etc. Then, another paramset could be defined that offers only *w* and *l* as instance parameters, with *as*, *ad*, *ps*, and *pd* being computed from *w* and *l*.

Another application would be to reduce the amount of redundant specification of model parameters when supplying a set of corners. In this case, a base paramset gives all parameter values that are shared between all the corners, and then a paramset is used to specify each corner, as follow

```
paramset n250nm n250nm_base;
    ...
    localparam string corner=design.corner from {"tt"};
    .vth0 = vth0_tt;
    ...
endparamset

paramset n250nm n250nm_base;
    ...
    localparam string corner=design.corner from {"ff"};
    .vth0 = vth0_ff;
    ...
endparamset
...

paramset n250nm_base bsim4;
    ...
    parameter vth0 = 0.3;
    ...
endparamset
```

2.9 Hierarchy

With SPICE, a .model statement can only be associated with a primitive component (a built-in). With Verilog-A/MS a module can contain both behavior and structure, so it has combined the concept of a subcircuit and a primitive. As such, paramsets (the proposed equivalent to .model statements for Verilog-A/MS) can be naturally applied to both structural and behavioral models. In addition, since the paramsets are used to implement model levels, models at different levels could have a different character. For example,

the simpler models could be purely behavioral whereas the more complicated models could be composites of other models, or in fact, they could be small circuits.

2.10 Monte Carlo

With Monte Carlo, the parameter values are taken to be functions of some random variables. For the moment I will assume that some mechanism it defined to allow us to define those random variables, with the actual mechanism described later. I will further assume that from the paramset, the values of the random variables are accessed in the same manner as one would access the values given in a constants module.

Start by assuming that object named *statistics* is defined that contains the declaration for a collection of random variables (described in the next section), partitioned between process variables (value is shared for all instances where it is used) and mismatch variables (each use of the variable gets draws a different value). Assume further that the *statistics* object declares the statistical averages, the distributions associated with each variable, and the correlation between variables. Then, providing instances whose parameters vary as a function of these random variables is simply a matter of using the variables when specifying the paramsets.

```
paramset n180nm bsim6;
    parameter real l=0.18u from [0.1u,0.25u);
    parameter real w=0.25u from [0.1u,25u);
    localparam real area=l*w from [0.0,5p);

    .as = 1u*w + statistics.da;
    .ad = 1u*w + statistics.da;
    .ps = 2*(0.15u + w) + statistics.dp;
    .pd = 2*(0.15u + w) + statistics.dp;
    .ld = 50nm + statistics.dld
    .type = "n"
    .vto = 0.25 + statistics.dvto;
    kp = 20u + statistics.dkp;
    .tox = 100n;
    .nsub = 6.02e23;
    .xj = 4e–7;
    .vsat= 200k;
    ...
endparamset
```

Several, but not all, of the random variables used in the above paramset, would be mismatch variables. As the value for these would be different for every instance, they would be identified and treated internally in a way that is very similar to paramset parameters as they will be different for every instance.

2.11 Statistics Blocks

Statistics blocks are used to define random variables that are used to describe the statistical variations in component parameters. A statistics block can contain any statement that can be found in a function declaration. In addition, it contains named or unnamed "process" and "mismatch" blocks. Process blocks are used to specify values for random variables that exhibit a batch-to-batch type variations, and mismatch blocks specify the

values for random variables that exhibit on-chip or device mismatch variations. Generally one declares a collection of variables within the statistics block and give those variables within the process or mismatch blocks. These variables are accessed from foreign modules with their hierarchical name. The process blocks are reevaluated once every simulation run whereas the mismatch blocks are reevaluated every time one of its variable values is accessed. Generally the values of the variables are the result of a call to a random function, and every evaluation of the block changes the values of its variables.

During a Monte Carlo analysis, the variables specified in the process block are updated once per Monte Carlo iteration, and are used to represent batch-to-batch or process variations, whereas the variables specified in the mismatch block are updated on a per-use basis and are typically used to represent device-to-device mismatch for devices on the same chip.

```
statistics sh3stats;
    real vto, rsh, tox, kdxl, rshsp, xisn, xisp;
    integer seed;
    seed = 5430932;
    process begin
        vto=rdist_normal(seed, 0.5, 0.1);
        rsh=rdist_normal(seed, 100, 12);
        tox=ln(rdist_normal(seed, 1e–8, 1e–9));
        kdxl=rdist_uniform(seed, 15e–9, 25e–9);
    end
    mismatch begin
        rshsp=rdist_normal(seed, 25, 2);
        xisn=rdist_normal(seed, 10, 0.5);
        xisp=rdist_normal(seed, 10, 0.5):
    end
endstatistics
```

It is possible to create correlated random variables by using linear combinations of independent random variables.

2.12 Hierarchical Virtual Parameters

In addition to the parameters that are declared on a paramset, 7 hierarchical virtual parameters are accepted and accessible. In Verilog-A/MS, any parameter that begins with a $ shall be considers a virtual parameter. These are parameters that are accepted by instances of all types of modules, but are not explicitly declared in the module defintion. Currently, 7 such parameters are accepted: *$m*, *$n*, *$x*, *$y*, *$angle*, *$vflip*, and *$hflip*.

1. The value of *$m* is intended to contain the shunt multiplicity factor for a module (the number of identical devices that should be combined in parallel and modeled). *$n* is the serial multiplicity factor (the number of identical devices that should be combined in series and modeled; this may only make sense for two terminal devices). *$x* and *$y* are the offsets of the location of the center of the device relative to the center of the context in which it exists (in meters). *$angle* is the rotation of the device relative to the context in which it exists (in degrees CCW). Finally, *$vflip* and *$hflip* specify the device is flipped either vertically or horizontally (*$vflip* specifies a vertical flip which would be about a horizontal axis (before rotation)).

2. These parameters are predefined and have values for all instances. If not otherwise set, the values are *$m* = 1, *$n* = 1, *$x* = 0, *$y* = 0, *$angle* = 0, *$vflip* = 1, and *$hflip* = 1.
3. The values of *$m*, *$n*, *$x*, *y$*, and *$angle* are all real numbers. The value of *$vflip* and *$hflip* are either –1 or +1.
4. The values are available within paramset or a module. The values are a function of both the value of the parameters specified on the associated instance statement and on the value of the parameter within the hierarchical context in which the instance is found. The resolved values within the paramset or module are given by the following relationships

$$\$m_{\text{resolved}} = \$m_{\text{specified}} \times \$m_{\text{hier}} \tag{1}$$

$$\$n_{\text{resolved}} = \$n_{\text{specified}} \times \$n_{\text{hier}} \tag{2}$$

$$\$x_{\text{resolved}} = \$x_{\text{specified}} + \$x_{\text{hier}} \tag{3}$$

$$\$y_{\text{resolved}} = \$y_{\text{specified}} + \$y_{\text{hier}} \tag{4}$$

$$\$angle_{\text{resolved}} = \text{mod}_{360}\ (\$angle_{\text{specified}} + \$angle_{\text{hier}}) \tag{5}$$

$$\$vflip_{\text{resolved}} = \$vflip_{\text{specified}} \times \$vflip_{\text{hier}} \tag{6}$$

$$\$hflip_{\text{resolved}} = \$hflip_{\text{specified}} \times \$hflip_{\text{hier}} \tag{7}$$

5. The output of every current source in the analog blocks is multiplied by the resolved value of *$m* of the paramset/module that contains the block. The value returned by every current probe within an analog block is divided by the resolved value of *$m* associated with that block.
6. The output of every voltage source in the analog blocks is multiplied by the resolved value of *$n* of the paramset/module that contains the block. The value returned by every voltage probe within an analog block is divided by the resolved value of *$n* associated with that block.
7. Parameters are applied to an instance in the following manner,

   ```
   M1 #(..., .$m(2)) bsim6 ( ... )
   ```

   (we also should consider using $m(2) rather than .$m(2) as a way of shortening the parameter list).
8. The value of hierarchical parameters are accessed from within a module or paramset in the following manner,

   ```
   real m = $m;
   ```

The idea with these parameters is that the size, location, and orientation of every instance can be specified relative to the size, location, and orientation of the block in which it is found. In this way, once the size, location, and orientation its parent is know, then its size, location, and orientation is also known.

The parameters *$m* and *$n* directly affect the behavior of the modules by implicitly affecting the behavior of the access functions. The other hierarchical parameters would only affect the behavior of the module if their values were explicitly included in the behavioral description.

2.13 Statistical Descriptions from Layout

With the constructs available it is possible to describe statistical mismatch variations in component parameters as a function of size, location, and orientation of devices as placed in the layout. Consider the parameter *drho* which is assumed to vary linearly across the die, with a random slope. This can be modeled using 4 random variables, *drho*00, *drho*01, *drho*10, and *drho*11. Then the value of *drho* at *x* and *y* would be

$$drho(x, y) = (1 - x)drho00 + x \times drho10 + (1 - y)drho01 + y \times drho11 \quad (8)$$

This would be an approximation to the value of *drho* that would be used for a device with a center that falls at *x*, *y*. This approximation can be improved by including corrections for size (*m* and *n*) and orientation (*angle*). For example, *drho*(*x*, *y*) might become

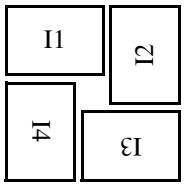
$$drho(x, y) = [(1 - x)drho00 + x \times drho10 + (1 - y)drho01 + y \times drho11]/\text{sqrt}(m \times n) \quad (9)$$

This could be described with statistics blocks and paramsets in the following way

```
statistics rhostats;
    process begin
        rho00=gauss(.mean(100), .std(20));
        rho01=gauss(.mean(100), .std(20));
        rho10=gauss(.mean(100), .std(20));
        rho11=gauss(.mean(100), .std(20));
    end
    mismatch begin
        drho=gauss(.mean(0), .std(5));
    end
endstatistics

paramset n180nm bsim6;
    ...
    rho = rhostats.drho/sqrt($m*$n) + (
            (1 – $x)*(1 – $y)*rhostats.rho00 + $x*(1 – $y)*rhostats.rho10 +
            (1 – $x)*$y*rhostats.rho01 + $x*$y*rhostats.rho11
        );
    ...
endparamset
```

Then a matched quad of devices in the configuration shown on the right can be instantiated as follows,



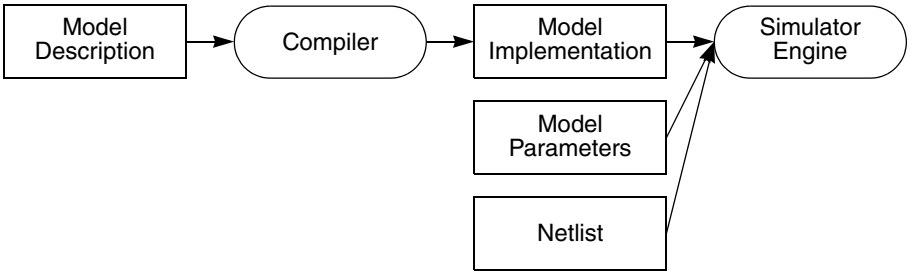

```
n180nm #( ..., .$x(–2), .$y(4), .$angle(0)) I1 ( ... )
n180nm #( ..., .$x(4), .$y(2), .$angle(90)) I2 ( ... )
n180nm #( ..., .$x(2), .$y(–4), .$angle(180)) I3 ( ... )
n180nm #( ..., .$x(–4), .$y(–2), .$angle(270)) I4 ( ... )
```

In this way the matching of the 4 devices as a function of their placement and orientation is automatically determined. In this example, the orientation is never used. This is because *rho* varies linearly across the chip. It would come into play if the variation were more a more complicated function of location.

3.0 Implementation

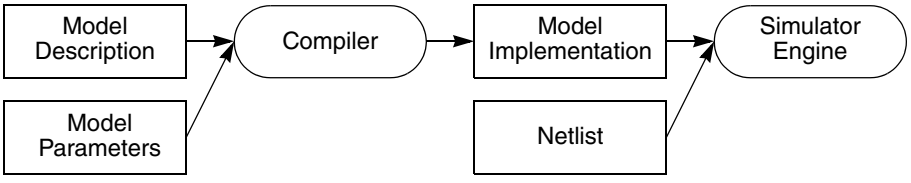
In SPICE, component parameters were partitioned into instance and model parameters for reasons of both user convenience and for efficiency. The efficiency came from the fact that many parameters can be placed in a single data structure that is shared between many instances, thereby reducing the memory required for the simulation. The model flow for this methodology is shown in Figure 1.

FIGURE 1 *Traditional SPICE model flow.*



In this proposal, the partition is determined by the paramset, which is independent of the module description. It is the paramset parameters that act as the instance parameters, all others are model parameter. So given that the partition between instance and model parameters is not specified when the module is written, how is the efficiency of SPICE maintained? One possible way is to compile a version of the model for each paramset used as shown in Figure 2. Thus, the behavioral model description would be combined with each of its paramsets and compiled generating a implementation of the model for each paramset. This has the benefit of creating smaller more efficient model implementations as paramsets generally have many fewer parameters than the underlying model description. The values of the extra parameters of the model description are constants, and so can be "compiled away". Thus, their values need not be saved in the internal data structure for the instance and many of the model equations that use these values can be simplified by pre-evaluation using the known values.

FIGURE 2 *Proposed model flow.*



4.0 Compatibility with Existing Model Files

To assure rapid adoption, it will be necessary for Verilog-A/MS to be compatible with existing SPICE model files. There is a tremendous amount of investment in those files, and they will not be replaced quickly, if ever. So from a practical perspective a Verilog-A/MS simulator must be able to either directly read existing model files, or it must be

possible to write an automatic translator. Fundamentally it should be possible to do either as long as Verilog-A/MS does not need information that is not available from in the model files. And in fact, there is one such piece of information, and that is the choice of which parameters should be instance (paramset) parameters. In SPICE, this information is contained within the simulator and is not available from the model files. As such, in order to enable the translation of SPICE model files, an extra file is needed, one that contains the list of instance parameters. This file should be very small and easy to create, and so is not considered a material barrier. Since it is not needed in a purely Verilog-A/MS implementation, the format of this file is not described. Instead, the format would be defined for each implementation by the vendor. It is hoped that easily translation of SPICE model files to Verilog-A/MS files will encourage a quick migration away from proprietary formats to industry standard formats.

5.0 Summary

In this proposal, the larger issue of the SPICE modeling infrastructure is addressed. A proposal is made that attempts to support all of the capabilities needed in today's SPICE model files. This was needed to assure that modeling infrastructure in the future is as simulator neutral as possible. The ideas is that once Verilog-A/MS becomes a viable language for compact modeling (once efficient robust implementations of the Verilog-A/MS extensions for compact modeling become available and are adopted), then foundries would provide model parameters that support all of the normal capabilities (binning, corners, Monte Carlo, levels, etc.) as they do today, but they would do it in an industry standard simulator neutral format that includes the definition of the models themselves. In this way, the models would be released with the process, would likely be tailored for the process, and would be usable in a wide variety of simulators, and each of the simulators would interpret the models in the same way and would have access to all of the model's features.