HOW TO (AND HOW NOT TO) WRITE A COMPACT MODEL IN VERILOG-A

Geoffrey J. Coram

Analog Devices, Inc.
804 Woburn St., Wilmington, MA 01887

ABSTRACT

Verilog-A was recently enhanced to provide greater support for compact modeling. In order for Verilog-A to become the standard language for compact model development and implementation, two more steps are necessary: compact model developers must become familiar with the language, and simulators must run compact models written in Verilog-A almost as quickly and reliably as those hand-coded in C. This paper addresses both of these steps: it provides a quick introduction to writing compact models in Verilog-A and, by indicating the sorts of techniques that compact model writers may use, helps simulator vendors understand the sorts of optimizations that are expected from their Verilog-A interfaces.

1. INTRODUCTION

The standard language of compact modeling has been C since about 1985, when Spice was re-written from FORTRAN (Spice2) into C (Spice3). Most recent compact models have been written in C, although some still use FORTRAN. The Compact Model Council [1] has preferred C code in the past, but it encourages the release of Verilog-A source code for its next-generation MOSFET model standardization effort. All the developers of candidate models for the CMC effort have indicated the intent to write Verilog-A implementations of their models.

The Verilog-AMS hardware description language [2], and specifically its analog-only subset called Verilog-A, was recently enhanced to provide greater support for compact modeling by the release of the Verilog-AMS Language Reference Manual (LRM) version 2.2. This version of the LRM was developed over the course of the past two years with involvement from compact model developers, vendors of commercial simulators, and others, and is expected to be approved by the Accellera [3] board of directors in September 2004. Many of the language extensions in the new LRM were among those proposed by Lemaitre [4].

Even before the addition of these compact modeling extensions, Verilog-A was an excellent language for compact model development and a dramatic improvement over C. The advantages are listed in Section 2. MATLAB is often used for compact model development, due to its ease of use and powerful data manipulation and plotting routines. However, MATLAB models cannot be used in circuit simulators. Verilog-A is almost as easy to use as MATLAB, and it can be used directly in circuit simulators – as well as in parameter extraction software, which provides methods for handling measured data. Verilog-A support, compliant with version 2.1 of the LRM, is available in a number of commercial simulators and parameter extraction tools, as well as in proprietary simulators of several semiconductor companies. Support for version 2.2 of the standard will happen soon, because most of the features are already available for built-in Spice models and for proprietary C interfaces to the simulators. The Verilog-A interface simply needs to connect the new syntax to existing functions in the C interfaces.

This paper is specifically intended for compact model developers who want to learn how to write compact models in Verilog-A. Section 3 provides a quick overview of the most useful elements of the language, which are scattered through the Language Reference Manual. Unlike the LRM, reference [5] is a user's guide, but intended more for circuit designers writing high-level behavioral models than for compact model developers. This paper focuses only on the analog features of Verilog-A and does not address mixed-signal modeling. Section 4 highlights a few tricks that might be useful, while Section 5 notes areas of particular concern for compact modeling.

This paper has a second intended audience: the programmers developing the Verilog-A interfaces for circuit simulators. Compact models written in Verilog-A should run reasonably fast, and simulator developers must invest some effort to make this happen. It might be acceptable for a new, automatically-compiled model to run slower than an older hand-coded model, if the features of the new model are compelling. For example, circuit designers accepted the BSIM3 MOSFET model for its improvements, even though it ran slower than the MOS level 3 model. And yet, it is not necessarily the case that Verilog-A models will run more slowly. Circuit simulators are not written in assembly language; modern C compilers are very good at automatically generating the assembly-level code. A good Verilog-A compiler could well produce more efficient code than a person would by hand, particularly if the person concentrates on developing the correct equations for the model. Section 6 considers some areas in which the compiler needs to take responsibility for optimizations. Section 7 suggests some additional compiler features to aid the development of good compact models.

Verilog-A has the potential to rejuvenate the field of compact model development by lowering the barrier for getting new models into simulators. Years ago, every semiconductor company had its own copy of the Spice source code and was therefore free to develop its own compact models. Most semiconductor companies have switched to commercial simulators, but they do not have the leverage to insist that proprietary models be implemented in these simulators. With Verilog-A, new models can be added almost as quickly as the equations can be defined. The semiconductor company has control over the implementation schedule, and it can maintain control of the intellectual property.

2. ADVANTAGES OF VERILOG-A

The main reason for preferring Verilog-A for compact modeling over general-purpose programming languages is that it frees the model developer from the burden of handling the simulator interface. The simulator interface includes a myriad of things such as reading the model parameters, checking the topology, allocating memory, initializing values or recomputing them for a new temperature, loading the Jacobian matrix and right-hand-side vector – each of which is a separate function in Spice that is called at a specific time during a simulation. Any given Spice-like simulator may also have chosen to implement these functions differently than Berkeley Spice. For example, the right-hand-side vector is loaded with different values depending on whether the Newton-Raphson algorithm solves for the vector of unknowns directly or solves for the vector of differences from the previous vector of unknowns. The simulator may not even be Spice-like: it could use the harmonic balance algorithm. Compact model developers using Verilog-A need not be concerned with such details.

Verilog-A simulators automatically compute symbolic partial derivatives of the currents and charges in a compact model and determine the proper insertion of these values into the Jacobian matrix for Newton's method. This work must be done by hand in C. Thus, even if one is modifying an existing compact model in C, in which most of the interface work has already been done, one still has to compute new derivatives, and possibly get new matrix pointers, when introducing a new dependence. The model developer is focused on getting correct equations for currents, which are compared against measurements; the derivatives are harder to verify. As a result, almost every hand-coded compact model has had some derivative errors in its first release.

Verilog-A also provides a strong system for defining model parameters. The declaration statement includes the default value and can specify the range of valid values. The default value may also be a function of other (previously-declared) parameters. For example, some parameters may have different defaults for NMOS and PMOS devices; this may be incorporated directly in the parameter declaration statement, rather than requiring special procedural code.

As a result, Verilog-A is an extremely efficient language for writing compact models. To cite one example: self-heating was added to a Verilog-A version of the BSIM3 MOSFET model in about two days; the same addition to the native C code took two weeks. Further, the Verilog-A implementation was used to find a bug (typographical error) in the C implementation of the derivatives.

3. QUICK PRIMER

Verilog-A is a reasonably simple language. One can learn most of the basic concepts simply by reading through an example, such as the diode model in Listing 1.

The rest of this section consists of a brief discussion of the elements used in this example. More details on the concepts can be found in [5] or, of course, in the Verilog-AMS Language Reference Manual [2]. A few complicated compact modeling examples can be found on the Internet [6].

Verilog-A *modules* replace the primitives of Spice. In most simulators that support Verilog-A, a module can be instantiated in a Spice netlist as if it were a primitive or subcircuit. Presently, however, not all simulators allow Spice .model cards for Verilog-A modules. Modules can contain other modules, making them something of a hybrid between primitives and subcircuits.

The device's terminals "a" and "b" – called *ports* in Verilog-A – appear in the parentheses following the module's name. Terminals of compact models should be declared **inout**; Verilog was originally a language for modeling digital logic, and the direction indicated signal flow. The terminals are also declared to be *electrical*. The definition of *electrical* comes from the included file "disciplines.vams," which contains the definitions of *disciplines*. Disciplines are essentially user-defined data types, where the definition includes the through and across variables, the units, and the absolute tolerance. For compact modeling, the main discipline of interest is *electrical*, with the across variable V and the through variable I. Self-heating modules may also reference the *thermal* discipline. An internal node can be defined by declaring its discipline, but not including it in the port list following the module name, as is done here for `int`.

This module declares two branches, `res` and `dio`. The example treats them as mnemonic shorthand for the corresponding pairs of nodes, but there are subtleties not exposed here that are relevant when multiple branches exist between two nodes.

The parameter declarations should follow the port and

```
`include "disciplines.vams"
`include "constants.vams"
module diode(a,c);
   inout a,c;
   electrical a,c,int;
   branch (a,int) res;
   branch (int,c) dio;
   parameter real is = 10p from (0:inf);
   parameter real rs = 0.0 from [0:inf);
   parameter real cjo = 0.0 from [0:inf);
   parameter real vj = 1.0 from (0:inf);
`ifdef __VAMS_COMPACT_MODELING__
   aliasparam phi = vj;

   (*desc="jct. voltage"*) real vd;
   (*desc="current"*) real id;
   (*desc="depl. charge"*) real qd;
   (*desc="depl. cap."*) real cd;
   (*desc="conductance"*) real gd;
`define GMIN ($simparam("gmin"))
`else
   real vd, id, qd;
`define GMIN (1.0e-12)
`endif

   analog begin
      V(res) <+ I(res) * rs;
      vd = V(dio);
      id = is * (limexp(vd/$vt) - 1.0);
      if (vd < vj) begin
         qd = cjo * vj * (1.0 -
            2.0 * sqrt(1.0 - vd/vj));
      end else begin
         qd = cjo * vd * (1.0 +
            vd / (4.0 * vj) );
      end
`ifdef __VAMS_COMPACT_MODELING__
      gd = ddx(id, V(int));
      cd = ddx(qd, V(int));
`endif
      I(dio) <+ id + `GMIN * vd;
      I(dio) <+ ddt(qd);
      I(dio) <+ white_noise(2 * `P_Q * id,
                  "shot");
      V(res) <+ white_noise(4 * `P_K *
                  $temperature * rs,
                  "thermal");
   end
endmodule
```

Listing 1. Verilog-A description of a diode. Verilog-A keywords are in boldface.

branch declarations. The parameter definition must include a default value. This value can be a simple number, including exponential notation or standard scale factors. The value can also be an expression containing previously-declared parameters; see Section 4.1. Parameter ranges can be specified in the declaration, such as (0:**inf**) in the example. Following standard mathematical notation, brackets `[]` indicate that the range includes the endpoints, but parentheses `()` exclude the endpoints. One may also **exclude** specific values. The simulator will automatically generate an error message if a user specifies a parameter that is out of range.

The compact modeling extensions provide for parameter aliases; a simulator supporting the extensions will allow `vj` to be specified as `phi` for this module. This example is constructed so that it will run even in a simulator that does not support the extensions. The accent grave character (`) indicates a compiler directive, replacing the pound sign (#) in C. The token `__VAMS_COMPACT_MODELING__` is pre-defined by a compiler that supports the extensions.

Variables can be declared at top level. They can also be declared within named blocks. Like C, these declarations must occur at the top of the block. With the compact modeling extensions, the variables are all marked with descriptions; this is intended to make them available to print as part of the operating point information or to save as a waveform.

The behavior of the module is contained in the **analog** block. A module can have only one **analog** block, which can contain arbitrarily many statements. Blocks are defined with **begin** and **end**, which replace the curly braces {} in C.

Verilog-A uses <+ to indicate a *contribution* to the voltage or current of a branch. The parasitic resistance uses the voltage formulation `V(res) <+ I(res)*rs;` to permit `rs=0`. The diode branch uses separate current contributions for the dc and capacitive currents; these values are combined to yield the total current in the branch.

Basic mathematical functions, such as are found in C, are available in Verilog-A. Those typically found in compact models are the operators `+ - * /` and the functions **sqrt**, **pow**, **ln**, **exp**, and **abs**. Improved convergence for semiconductor junctions can be obtained by using **limexp** instead of **exp**. Capacitive currents are defined using the time-derivative of charge, with the **ddt** operator. The compact modeling extensions also provide the partial derivative operator **ddx**, which is most useful for operating-point information such as capacitances and conductances. In some models, such as the BSIM3 MOSFET model [7], thermal noise is computed from these partial derivatives.

The simulation temperature is requested with `$temperature`, and the thermal voltage is requested with `$vt`. The compact modeling extensions provide a method of requesting other important values from the simulator, such as the minimum conductance `$simparam("gmin")`. The `$` indicates a *system function*.

Conditional statements can be implemented with **if** and **else**. The conditional expression can use the usual logical operators `&&` `||` `!`, the relational operators `>` `>=` `<` `<=`, and the equality operators `==` `!=`. Verilog-A also has a **case** statement and a **for** loop, though these are not common in compact models.

Small-signal noise sources can be added to a model using the **white_noise** and **flicker_noise** functions. Shot noise and thermal noise, both types of white noise, are present in this model. The values of Boltzmann's constant and the electron charge are obtained from the header file "constants .vams," which contains several physical and mathematical constants with the prefixes `P_` and `M_`, respectively.

4. CLEVER TRICKS

Verilog-A has a number of other useful features that are not included in the diode model of the previous section. Parameter handling in Verilog-A is much easier than in C; some interesting examples are presented in this section to show the power of Verilog-A. On the other hand, debugging is somewhat harder, because one does not naturally have access to the low-level computations such as derivatives, or even the values during Newton-Raphson iterations.

4.1. Parameter tricks

It was mentioned above that parameters can have default expressions involving previously-declared parameters. For example, a resistor model might have its resistance parameter declared as follows:

```
parameter real r = rho * length / width;
```

assuming that `rho`, `length`, and `width` were declared before this line. The default expressions can also include conditional code:

```
parameter integer mobmod = 1  from [1:3];
parameter real uc =
     (mobmod==3) ? -46.5e-3 : -46.5e-12;
```

The range expression can contain previously-declared parameters, as well. The following lines constrain `p2` and `p3` such that `(1-p2-p3) > 0`.

```
parameter real p2 = 0  from [0:1);
parameter real p3 = 0  from [0:1-p2);
```

The LRM requires the start of the range to be strictly less than the end of the range, but it is possible to limit the range to a single value:

```
parameter integer level = 11010
                          from [11010:11011);
```

4.2. N and P type devices

Compact models are generally formulated in terms of the N-type device, *i.e.*, NMOS or NPN. The equations in the documentation are written for the N-type device. The C code includes a special flag to allow it to work for the P-type device as well. In Spice, the primitives actually have different names, but the underlying code is the same. In some commercial simulators, the newer device models have one primitive name and a parameter to switch between N- and P-types. This latter approach works in Verilog-A, and the implementation could be almost a direct translation of the C code. It is mentioned here because one Verilog-A BJT model initially did not include it. One certainly would not want to write the model equations twice, nor would one want two compiled modules in memory when one would suffice.

The simplest way to add this functionality is to declare an integer parameter `type`:

```
parameter integer type = 1 from [-1:1] exclude 0;
```

Then, the behavioral equations might include lines like the following:

```
Vds = type * V(d,s);
...
I(d,s) <+ type * id_chan;
```

The compact modeling extensions provide a way to make this clearer for the user, by using a **string** parameter:

```
parameter string type = "NMOS"
                  from {"NMOS", "PMOS"};
localparam integer sign=(type=="NMOS")?1:-1;
```

The **localparam** cannot be specified (or *overridden*, in Verilog terminology) when the module is instantiated, but, being a parameter, it allows the compiler to make certain optimizations.

Two caveats should be mentioned here. The sign change only applies to voltages and currents; the temperature and power for self-heating networks do not get adjusted.

```
temp = Temp(tnode) + $temperature;
```

A more significant concern relates to the `$limit` function. The syntax for `$limit` requires that the first argument be a branch voltage, so one cannot write

```
$limit(type * V(b,e)) // INVALID!
```

Also,

```
type * $limit(V(b,e)) // WRONG!
```

is not the correct application of limiting; one instead needs:

```
if (type==1) begin
   Vbe = $limit(V(b,e),"pnjlim",vcrit);
   Vbc = $limit(V(b,c),"pnjlim",vcrit);
end else begin
   Vbe = $limit(V(e,b),"pnjlim",vcrit);
   Vbc = $limit(V(c,b),"pnjlim",vcrit);
end
```

One could also simply use **limexp** if the branch voltage is only used in one exponential.

4.3. Debugging

The compact modeling extensions include the system task `$debug` that prints its arguments on every iteration, as well as the **ddx** function to request derivative information. The format specifier `%m` can be used to print the hierarchical name of the module.

```
gm = ddx(id_chan, V(g));
$debug("%m :  gm = ", gm);
```

In a simulator without the extensions, one can still trick the simulator into providing derivative information. This trick can also be used to double-check the matrix stamp for simulators with the extensions. One can add an extra terminal and contribute to the current of that terminal any variable in the module:

```
module mymos(d,g,s,b,test);
...
I(test) <+ Vgsteff;
```

Using the following netlist (in Spectre format):

```
X1 (d g s b test) mymos
R1 (test 0) resistor r=1
Vd (d 0) vsource type=dc dc=1 mag=1
```

the ac node voltage of `test` will be the partial derivative of `Vgsteff` with respect to `V(d)` (assuming there is no parasitic drain resistance).

5. HAZARDS

The automatic derivative calculations of Verilog-A eliminate the chances of the model developer making derivative errors. However, there are still hazards that the language cannot protect against – and some new hazards that are not present in C or MATLAB.

There is one particular hazard in Verilog-A that bears mentioning, because at least three independent model developers have encountered it. **log** means the base-10 logarithm; **ln** is the natural logarithm.

Many of the other hazards are also present in C and MATLAB, so that compact model developers should already be aware of them. For example, one must pay proper attention to operator precedence and guard against division by zero. The results of binary operations may depend on their arguments; for example, `b = 1/2;` results in a value of zero in C and Verilog-A (but not MATLAB), because the arguments are integers.

5.1. Floating-point exceptions and hidden derivatives

Some mathematical functions are not well-behaved for all arguments. This is of particular relevance for compact modeling, since during Newton-Raphson iterations, the voltages may take on "unreasonable" values. One can help ensure that **exp** does not produce too large a result by using **limexp**, though some compact modelers prefer to explicitly linearize the exponential above a breakpoint.

Division by zero is another potential numerical problem. An interesting wrinkle is that Verilog-A hides some of the equations that need to be checked. For example, if `x = sqrt(y)`, then there is no problem computing `x` when `y=0`, but the automatically-computed derivative will include `1.0/sqrt(y)` if `y` depends on a node voltage. It should be mentioned in this context that it is not acceptable to tinker with the derivative; one simulator computed the derivative of `sqrt(x)` as `1.0/sqrt(x + ε)`, but chose an ε that happened to be larger than `x` for a particular application. Since the language is general-purpose, one cannot *a priori* choose an ε that is appropriate for all applications.

`abs(x)` does not have a derivative at `x=0`, and therefore it should not be used in expressions that depend on the voltages. The resulting behavior would be non-physical and could cause convergence problems. The following code was present in one Verilog-A model with self-heating:

```
Tk = $temperature + DTA + abs(V(dt));
```

On the first iteration, node voltages are frequently initialized to zero, *i.e.*, `V(dt)=0`; this means that the simulator hits the derivative problem exactly. If the simulator chooses to set the derivative of **abs** equal to zero at the origin, then the Jacobian matrix would incorrectly indicate that the device currents have no dependence on the self-heating temperature, inhibiting convergence. Also, this particular formulation could converge with `V(dt)` negative.

The **pow** function also requires care: it can give a result that is too large, it is not defined for some arguments, and its derivative is not defined for some arguments where the function itself is defined. In many cases, it is preferable to replace `pow(a,b)` by the equivalent `exp(b * ln(a))`, particularly when the same `a` is raised to a few different powers. The exponential function requires fewer cycles on most computers, and one has the option of using **limexp**.

Other mathematical operators like **tan** or **sinh** can also have problems, but they are not used in compact modeling.

5.2. Watch those if statements

if statements are always a concern in compact modeling. Compact models need to have currents that are continuous functions of their input voltages. For Newton's method to work, the functions should also have continuous first derivatives. In order for a compact model to predict correct harmonic distortion, the higher-order derivatives must also be continuous. A physical system has continuous derivatives of all orders. It is difficult to enforce that continuity across an **if** statement.

Verilog-A provides a new trap for model writers to fall into. If one tried to directly translate the depletion capacitance equation from the BSIM3 C code [7], one might get:

```
if (vbs == 0.0) begin
   qbs = 0.0; // BAD IDEA! dqbs/dvbs=0!
   capbs = czbs + czbssw + czbsswg;
end else if (vbs < 0.0) begin
   qbs = ...
```

Note that, while capbs is assigned a value, this is not the value of the derivative that the simulator uses (capbs might be used for debugging or operating-point information). Capacitive currents in Verilog-A are contributed with

```
I(b,s) <+ ddt(qbs);
```

and the simulator computes the derivative of qbs from the assignments that were made.

Consider also the following example:

```
    if (dT < 1e-10) dT=0.0;
```

If the simulator converts the Verilog-A to C code (which most do), the result for that line and a few preceding it would read as follows:

```
    Tamb = $temperature;
    Tdev = Tamb+dt+V(tnode);
    AD_dTdev_dtnode = 1.0;
    dT = Tdev-Tnom;
    AD_ddT_dtnode = 1.0;
    if (dT < 1e-10) {
       dT=0.0;
       AD_ddT_dtnode = 0.0;
    }
```

where the derivatives calculated by the simulator are denoted by the leading AD_ While the value of dT is changed only slightly in the **if** statement, its true derivative is zeroed out. This would probably degrade convergence performance and yield incorrect results in a small-signal analysis.

5.3. Numerical precision

In some cases, it is important in which order some operations are performed, due to the finite precision of computer arithmetic. For example, when computing the expression (1.0 + 2.0e-30) - 1.0, the result will be 0.0 because the computer is unable to distinguish the quantity in parentheses from the exact value 1.0. In most cases, this sort of numerical problem can be avoided by writing the expression correctly. However, the concern in Verilog-A is that the model developer does not have access to the derivative expressions. Compact model developers should always be aware of situations where numbers of vastly different orders of magnitude are combined.

5.4. Inappropriate functions

Verilog-AMS is a general-purpose hardware description language. As such, it includes a number of features that are not appropriate for compact modeling. It is unlikely that a compact model developer would use some of these, such as **transition**, **slew**, **last_crossing**, and the Z-transform operators, so a warning against using any of these functions should not be necessary. On the other hand, some functions such as **absdelay** and **analysis** may be tempting.

The **absdelay** operator should not be used to model an ideal delay. An ideal delay is not physical, so it does not belong in a compact model. Furthermore, circuit simulators have to go to a lot of trouble to handle an ideal delay in a transient analysis. Laplace transforms may also be difficult to handle in the time domain, though not as difficult as an ideal delay. It may be better to explicitly compute an equivalent RC or RLC filter.

The **analysis** function provides a method of executing some statements conditioned on the type of analysis. One might think to use this to speed up a model evaluation by skipping calculations. For example, if one is not doing a transient analysis, one might think to skip the charge calculations:

```
if (analysis("tran"))
   q = ...
```

Unfortunately, this code would result in the capacitances also being zero, and a small-signal ac analysis would give incorrect results. Also, some simulators perform "pseudo-transient" analysis to find the dc operating point; the device charges are expected to help smooth out sharp nonlinearities, but the charges would not be present in this example.

Because the computations for the noise functions in the BSIM3 model are complicated, one Verilog-A implementation wrapped the evaluation as follows:

```
if (analysis("noise")) begin
   flicker=strongInversionNoiseEval(vds,
                                    temp);
```

This worked fine for standard Spice analyses, but would not give any noise for Spectre's periodic noise analysis (PNoise) or the noise analysis of a harmonic balance simulator.

The Verilog-A compiler must be sophisticated enough to determine when the results of calculations are required and when they are not.

5.5. Events

Events are another mixed-signal concept that is generally not appropriate in compact modeling. Compact modelers should not use any events other than **above**.

A common trick along the lines of the **analysis** examples in the last section is to use the event @(**initial_step**) to perform initialization of variables that only depend on parameters. For example, one might compute

```
@(initial_step) begin
   isdrain = jsat * ad;
```

This code would get executed only on the first timepoint, and would be skipped for the rest of a transient analysis. However, the **initial_step** is true for every iteration of the first timepoint, so the simulator is still re-computing the expressions. Also, prior to the release of LRM 2.2, the behavior of **initial_step** for a dc sweep was not defined: some vendors believed it should be true for every point of a dc sweep, so that one could put temperature updates inside @(**initial_step**) and still perform a correct temperature sweep. However, this meant that the code was executed for every iteration of every point of the dc sweep, even if the swept variable did not affect any of the calculations. Instead, this sort of optimization should be left to the simulator. The Verilog-A compiler must determine what expressions have derivatives with respect to the node voltages in order to fill in the Jacobian matrix; those expressions that do not depend on the voltages should be pre-computed once.

The **above** event is useful for generating warning messages when a device enters a particular region of operation.

```
@(above(Vds - Vdsat))
  $strobe("Device is saturated");
```

The advantage of the **above** event over a simple **if** statement is that the message will only be issued once for each crossing, rather than at each timepoint (or dc sweep point) that the device remains saturated.

5.6. Constants

Constants are another potential hazard. The two lines below were found in different Verilog-A models:

```
`define TNOM (272.15+27.0)
Tnom = tnom + 273.0;
```

Neither one used the correct value 273.15 for the conversion from Celsius to Kelvin. This value is available from "constants.vams" as the token `P_CELSIUS0. It is worth mentioning here that `$simparam("tnom")` returns a value in Celsius, but `$temperature` is in Kelvin.

The mathematical constants, such as `M_SQRT2 and `M_PI, are also found in "constants.vams" and defined out to 20 significant digits. Compact modelers are advised to use these tokens, rather than typing the values in by hand.

The values of the physical constants are more controversial, because Berkeley Spice [8] and many simulators derived from it use different values than the best accepted values for these constants available from the National Institute of Standards and Technology (NIST) [9]. The Verilog-AMS LRM [2] contains a listing for "constants.vams" that uses values from NIST. However, some simulators come packaged with a file that uses the Spice values. As an example of the differences, consider the values of Boltzmann's constant *k* given in Table 1.

It is hoped that a future revision of the LRM will define tokens such as `P_K_SPICE and `P_K_NIST2004 that will be consistent across all simulators and not change if NIST updates the best accepted values.

Source	Value (J/K)
Spice [8]	1.38062e-23
NIST (2002) [9]	1.3806505e-23
textbook (Sze) [10]	1.38066e-23

Table 1. Boltzmann's constant

6. OPTIMIZATIONS

It is of critical importance that that compact models written in Verilog-A run quickly for circuit simulation. Even if the new model has additional features, circuit designers will resist a slower model. This resistance has been seen in the BSIM4 MOSFET model, which adds a number of features important for smaller-geometry processes over its predecessor, BSIM3. Foundries have released Spice model libraries with crude subcircuits around BSIM3 models to include gate leakage without moving to BSIM4. Only if Verilog-A compact models run reasonably efficiently will Verilog-A become the dominant language of compact modeling. This section points out some obvious ways in which Verilog-A compilers can optimize the code generated for Verilog-A compact models. Further optimizations are certainly possible beyond what is presented here. Verilog-A compiler authors should review the C code produced for publicly-available Verilog-A models [6]. It would also likely prove valuable to write a common model such as BSIM3 in Verilog-A, and then compare the automatically-generated C code with the original distributed C code.

6.1. Common subexpressions

Verilog-A compilers should aggressively combine common subexpressions. The diode current is

```
id = is * (exp(vd/$vt) -1.0);
```

and the automatic derivative is

```
gd = is/$vt * exp(vd/$vt);
```

Someone hand-coding the model would not make two calls to the exponential function; the Verilog-A compiler should not either. Assuming the Verilog-A compiler generates C code, the C compiler cannot be counted on to make this optimization, because it does not know if the function call (**exp**) has any side effects.

The Verilog-A compiler should look for efficient subexpressions. The costs of mathematical operations are generally known for various processors, with addition costing less than the exponential. Similar to the way an optimizing C compiler works, the Verilog-A compiler might try several approaches to find the best subexpressions to factor out and re-use. A person faced with the same task might well choose a subexpression for readability that results in less efficient code. In a quick perusal of the BSIM3 code, a few separate occasions were found where multiple expressions were all divided by the same value; since division is more expensive than multiplication, it would have been more efficient to generate a subexpression for the inverse of the value and then perform multiplications. The optimization in C compilers is also limited in scope; it may not be able to consider a value and all its partial derivatives at the same time.

6.2. Dependency trees

Another critical area for compact model compilation is the creation of a dependency tree. The compact model has a large number of equations, and not all of them are needed at every phase of every type of analysis. Additionally, some compact models have a large number of calculations that need to be performed only once for a given set of parameters at a given temperature. However, as mentioned in Section 5 (specifically, 5.4 and 5.5) it cannot be the compact model developer's responsibility to determine what equations are not needed. The developer should not need to know details of the simulator's homotopy methods for dc convergence or its advanced analyses such as RF noise analysis.

Not only are some equations not needed at certain points in the analysis, other equations never need partial derivatives computed. The diode model of Listing 1 could be extended to model flicker noise by the addition of these two lines:

```
flick = kf * pow(id, af);
I(dio) <+ flicker_noise(flick, ef, "1/f");
```

(along with declarations of the parameters `af` and `ef`). The compiler should recognize that `flick` is only used in a small-signal noise expression, and thus its derivatives are never needed.

6.3. Eliminating nodes

In the diode model of Listing 1, the parasitic resistance was defined by a voltage contribution that allows zero resistance. In the case that `rs=0`, the internal node `int` is shorted to the anode, regardless of the operating point. The simulator should be able to eliminate the extra node. In fact, the simulator may save two rows in the matrix: in the modified nodal analysis used by most Spice simulators, a voltage contribution requires an extra row for the branch current in addition to row for the extra node itself.

Because of the extra row for the branch current, compact model writers may prefer to contribute to the current in a conductance formulation. In order to allow `rs=0`, a switch branch is required.

```
if (rs == 0)
   V(res) <+ 0;
else
   I(res) <+ V(res) / rs;
```

Verilog-A compilers should handle this gracefully and not introduce extra rows. Note that the compiler must construct a correct dependency tree to know when to make this optimization. A model for an ideal switch, which switches between open (`I(sw)<+ 0;`) and closed (`V(sw)<+ 0;`) would have a conditional expression depending on the control voltage, and it would therefore be ineligible for this optimization.

The resistance and conductance formulations describe the same analog behavior, and in principle, they could be implemented identically by the Verilog-A compiler. However, the convergence behavior of one or the other formulation may be better in certain cases. For example, the so-called "small resistor problem" results when resistances of vastly different orders of magnitude are combined and one value is lost due to finite numerical precision. Authors of Verilog-A compilers should consider the implications carefully before changing one formulation to the other.

6.4. Adding nodes

According to the current version of the Verilog-AMS LRM [2], the following two lines are illegal:

```
I(dio) <+ id + ddt(qd);
I(cap) <+ C * ddt(V(cap));
```

Syntactically, the **ddt** is an "analog_expression," and one cannot combine two analog expressions with a binary operator. This restriction is expected to be removed in a future revision of the LRM. The diode module of Listing 1 properly places the **ddt** on a separate line.

Most Verilog-A compilers are able to handle both those lines correctly, although some of them are known to add an extra row for these types of expressions. If the variable `C` depends on the node voltages, then an extra row is needed; however, this is not the correct way to model a nonlinear capacitor: the expression should be **ddt**`(C * V(cap))`. Verilog-A compilers should not introduce unnecessary rows, and they should issue a warning when the module requires them.

6.5. The chain rule

Some further optimizations may be found in considering the chain rule for derivatives. For example, many equations in BSIM3 depend on Vgsteff, which itself depends on Vgs, Vbs, and Vds. Any equation that depends on Vgsteff, therefore, should really have three partial derivatives calculated. However, the source code from Berkeley [7] cleverly calculates only one derivative with respect to Vgtseff (though it is confusingly named _dVgs) and then completes the chain rule at the end with the following lines:

```
Gds += Gm • dVgsteff_dVd;
Gmb += Gm • dVgsteff_dVb;
Gm *= dVgsteff_dVg;
```

Prior to these lines, `Gm` was the partial derivative of the channel current with respect to `Vgsteff` and `Gds` did not include that portion of dependence of the channel current on `Vds` that came through `Vgsteff`.

6.6. Model versus instance parameters

Along with the questions of what code is needed for what simulation, another implementation detail that should be hidden from the compact model developer is the difference between model and instance parameters. Certainly, from a simulator standpoint, it is important not to require storage for each of BSIM3's several hundred model parameters for each instance of a device that uses the model. However, the partitioning of parameters between instance and model is not fixed. Some users want the flatband voltage – which is usually a model parameter – to be an instance parameter, in order to facilitate mismatch modeling. Other users might not need the drain and source areas to be instance parameters, and it might be more efficient to calculate a simple expression (`AD = 2 * W * HDIF`) rather than storing the value for a million transistors. Thus, the compact model developer cannot be expected to partition the parameters.

The compact modeling extensions for Verilog-A provide the **paramset**, which extends the capabilities of the Spice .model card. However, the paramset is only available for netlists written in Verilog format, rather than Spice format. Simulator vendors need to determine how to support .model cards in their netlist format in an efficient manner.

7. VERILOG-A DEBUGGER

This section provides some suggestions for how a Verilog-A compiler and simulator could assist in debugging a compact model. The first two ideas below are fairly simple, but checking continiuty could require some sophistication.

7.1. Error messages

It is important that the simulator running a Verilog-A model produce error messages that are clearly related to the original Verilog-A code, rather than the generated code. As was noted in 5.1, **sqrt**`(0)` is well-defined, but its derivative is not; the error message should implicate the square root function, not the division by zero.

7.2. Warning mode

Most C compilers accept the argument "+w" to turn on additional warnings. A Verilog-A compiler could accept this argument to check for the following types of unusual conditions.

1. Variables that have a value assigned, but the value is not used (output variables, such as `cd` and `gd` in the diode example, should not cause warnings).

2. Variables that are used before they are explicitly assigned a value (note that Verilog-A specifies that variables are all initialized to zero).

3. Internal nodes that are created by the syntax of expressions (see Section 6.4).

4. Contributing to both the voltage and the current of a branch (the last contribution wins, setting the type).

5. Accessing or contributing to the current of a node.

The last item requires some explanation. The expression

```
Id=I(a);
```

creates a short-circuit branch from the port to ground and measures that current; the intent was to measure the terminal current, which is done with

```
Id=I(<a>);
```

Compact models should also contribute to branch currents rather than node currents. These lines are from a MOSFET model:

```
I(gate) <+ cqgate;
I(drainp) <+ cqdrn;
I(bulk) <+ cqbulk;
I(sourcep) <+ -(cqgate+cqdrn+cqbulk);
```

Each of these lines creates a branch from the respective node to ground and specifies a current in it. If the currents do not sum to zero, then the model appears not to obey Kirchoff's current law. It takes one fewer line to contribute to branch currents, and current conservation is automatically enforced.

```
I(gate,sourcep) <+ cqgate;
I(drainp,sourcep) <+ cqdrn;
I(bulk,sourcep) <+ cqbulk;
```

7.3. Continuity

It is tedious, yet essential, to check the continuity of equations across **if** statements. If the equations and the derivatives are not continuous, then convergence could be impaired. All the **if** statements in which the conditional expression has a dependence on the node voltages must be checked. The compact model compiler has all the pieces necessary to perform the check, and it would be of great benefit if it would perform the check automatically.

Consider the following lines from the BSIM3 source code [7]:

```
if (Abulk0 < 0.1)
{ T9 = 1.0 / (3.0 - 20.0 * Abulk0);
  Abulk0 = (0.2 - Abulk0) * T9;
  dAbulk0_dVb *= T9 * T9;
}
```

To check continuity, one must evaluate these lines at the breakpoint, `Abulk0 = 0.1`. One finds that `T9 = 1.0`, `Abulk0 = 0.1`, and the derivative is unchanged.

If the code above were translated into Verilog-A, the derivative `dAbulk0_dVb` would not appear in the source code. When the module was compiled, the compiler would generate this derivative. Therefore, the compiler would be aware that the conditional expression depends on the node voltages. This would trigger a check of the final value of

`Abulk0` and its derivative at the breakpoint. Some symbolic manipulation will be necessary for more complicated expressions. In some cases, the compiler may need to understand numerical precision, such that **ln**`(1 + x)` is numerically indistinguishable from 0 for `x < 1.0e-16`.

7.4. Preventing ill-formed models

The Verilog-A language prevents the creation of two types of incorrectly-formulated models. The first type is a model that does not have consistent ac and transient behavior. The Spice small-signal ac analysis is intended to compute the limiting behavior of the response of a circuit to a transient sinusoidal stimulus, when the amplitude of the sinusoid is decreased to zero. Therefore, ac and transient behavior of a model should be consistent. Some transistor models have violated this rule by altering the Jacobian matrix for ac analysis in order to capture non-quasi-static effects. Verilog-A does not allow the compact model writer access to the Jacobian matrix. The matrix used for transient analysis must be consistent with the matrix for ac analysis.

The other type of poorly formulated model is one that is capacitance-based instead of charge-based, that is, one that does not use charge as the state variable. The original recognition that a capacitance-based model does not conserve charge, due to numerical effects of the integration method, was made by Yang [11]. A more illustrative explanation is given in [12]. Verilog-A requires that models be charge-based, because no provision is made for integration of a capacitance model.

Compact model developers may want to use these approaches to create a new model. It is incumbent upon the developers of Verilog-A compilers not to provide proprietary extensions to support these ill-conceived approaches.

8. CONCLUSION

This paper has provided a wide-ranging overview of the features and dangers in Verilog-A compact modeling. If compact model developers follow the suggestions in this paper, their models are more likely to work as intended. Verilog-A compiler authors have been given some hints on common ways that the language may be used, so that they can ensure that their simulator will run the models correctly and efficiently.

Verilog-A is on the path to becoming the preferred language for compact modeling. Verilog-A provides a quick method of enhancing compact models to model new physics of advanced processes. Research groups, both academic and industrial, can easily make their new equations available. Simulator programmers can concentrate on innovative simulation techniques to distinguish their simulators rather than struggling to keep up with standard models.

9. ACKNOWLEDGMENTS

The author would like to thank Dr. Colin McAndrew for the invitation to present this tutorial, as well as for his support for automatic model compilers over the past many years. Thanks are also due to the compact model developers whose models provided many of the examples cited here – usually unintentionally.

10. REFERENCES

[1] http://www.eigroup.org/cmc/

[2] *Verilog-AMS Language Reference Manual*, version 2.2, Accellera, 2004.

[3] http://www.accellera.org

[4] L. Lemaitre, *et. al.*, "Extensions to Verilog-AMS to Support Compact Device Modeling," *Proc. 2003 IEEE International Workshop on Behavioral Modeling and Simulation (BMAS 2003)*, San Jose, CA.

[5] K.S. Kundert and O. Zinke, *The Designer's Guide to Verilog-AMS*, Boston: Kluver Academic Publishers, 2004.

[6] A few models, including the Philips MOS11 MOSFET transistor model, are available from http://www.designers-guide.com/VerilogAMS/ The Philips Mextram 504 bipolar transistor model is available at http://ectm.et.tudelft.nl/data/artwork/design/code/ahdl/mextram.va Silvaco International has some Verilog-A models available for "non-commercial use" for registered users at https://src.silvaco.com/ResourceCenter/en/downloads/verilogA.jsp

[7] BSIM3 (version 3.2.4) source code from the University of California, Berkeley. Available from http://www-device.eecs.berkeley.edu/~bsim3/

[8] Spice3f4 source code from the University of California, Berkeley. Available through anonymous FTP to ic.eecs.berkeley.edu in the subdirectory pub/Spice3/ or by contacting software@eecs.berkeley.edu. The value is found in the file src/lib/fte/cpitf.c

[9] http://physics.nist.gov/constants and specifically http://physics.nist.gov/cuu/Constants/Table/allascii.txt

[10] S.M. Sze, *Physics of Semiconductor Devices*, 2nd. ed., New York: John Wiley & Sons, 1981.

[11] P. Yang, *et. al.*, "An Investigation of the Charge Conservation Problem for MOSFET Circuit Simulation," *IEEE J. Solid-State Circuits*, vol. SC-18, Feb. 1983.

[12] K.S. Kundert, *The Designer's Guide to SPICE and Spectre*, Boston: Kluver Academic Publishers, 1995.