Extensions to Verilog-A to Support Compact Device Modeling

Laurent Lemaitre, Geoffrey Coram[†], Colin McAndrew, Ken Kundert[‡]
Motorola, Inc., Geneva, Switzerland,
[†]Analog Devices, Inc., Wilmington MA, [‡]Cadence Design Systems, Inc., San Jose, CA
laurent.lemaitre@motorola.com, geoffrey.coram@analog.com, colin.mcandrew@motorola.com, kundert@cadence.com

Abstract

This paper discusses extensions to Verilog-A that address compact modeling needs. It reviews compact modeling and analog circuit simulation, and then presents a simple Verilog-A compact model for a capacitor. Based on this example, extensions are presented that make Verilog-A better suited to compact modeling. A tentative implementation of each extension is proposed and described. The paper concludes with a summary of the extensions, implemented in a revised capacitor model.

1. Introduction

Models for semiconductor devices and circuits range from the microscopic (so-called TCAD models based on first-principles, fundamental device physics) to the macroscopic (behavioral models for complete integrated circuits). For the purpose of analog circuit simulation the former are too computationally complex and the latter are not sufficiently accurate. Physically based analytic models, termed compact models, that balance computational complexity and accuracy, like the BSIM3 model [1], are required for circuit simulation.

Compact modeling involves four phases:

Derive physics-based constitutive equations

Encode these equations in a computer language

Implement the code in a circuit simulator

Validate the compact device model implementation

In most compact model development there is little synergy between the approaches chosen for each phase of the process. Consequently, improvements in models and large-scale distribution of models happen slowly. Bug fixes in the implementation of a compact model in one simulator can take a long time to propagate to other simulators. Improvements in the physics of compact models to account for new physical effects in advanced manufacturing processes are even slower in reaching the broad electronic design audience.

Recently, proposals have been made to use a new model development methodology that increases the synergy between the phases listed above [2],[3],[4]. A common theme of the proposals is the use of Verilog-A as the standard language for modeling. This choice increases the degree of standardization of compact models, to the extent that the Verilog-A standard itself is implemented consistently.

At present, the Verilog-A syntax [5],[6] lacks constructs that are necessary to fully describe compact models and make them ready for automatic implementation into electrical simulators.

The main goal of this paper is to synthesize many of the proposals made so far in standardizing compact modeling using Verilog-A, and to build a requirements list of extensions that enables development of ready-to-implement compact models in Verilog-A.

The extensions presented here result from discussions that occurred within a subcommittee of the Verilog-A language definition committee under the auspices of Accellera [6], a consortium that defines the Verilog-A standard. Tentative implementations of some extensions were proposed in [4]. However, not all of the proposals fitted well with the overall features of Verilog-A.

In the next two sections we will briefly discuss the way that compact models are implemented in simulators. We will present some mechanisms involved during the DC analysis of an electrical circuit. These are prerequisites to better understand the need of certain extensions that are described below.

2. How Simulators Implement Compact Models

The implementation of a compact model in a simulator conceptually requires handling of the following components:

A set of model parameters

A set of instance parameters

Model initialization

Instance initialization

Evaluation (of branch constituent relations)

Post-processing

Model parameters are typically physical or empirical parameters that are independent of a particular instance of a device. Examples include the gate oxide thickness T_{OX} of a MOSFET, the saturation current per unit area J_S of a junction, and the effective width variation Δ_W of a resistor.

Instance parameters are those that depend on a particular instance of a device, and typically define layout attributes of a device. Examples include MOSFET gate width w and gate length l. Some models allow "model" parameters to be specified for an instance (e.g. temperature coefficients for resistors), and statistical parameter variations for mismatch also need to be defined per instance.

Model initialization is a block of code that takes care of parameter defaulting, parameter range checking, and parameter clamping and temperature variation. This code pre-computes variables that do not depend on instance parameters or controlling electrical bias.

Other variables can depend on instance parameters, but not controlling electrical bias, and these are calculated in the

instance initialization block, e.g. the transconductance parameter $\beta = \mu_0 C_{OX} (w/l)$ of a MOSFET.

Model evaluation is a block of code that typically calculates the values of currents, charges, and noise as a function of branch voltages, i.e. it implements the physical equations of a compact model. Since a simulator's solver routine calls the model evaluation code repeatedly, it is important to keep the evaluation block as simple as possible. Note that not all parts of the model evaluation block are required for all analyses in a simulator; charges and noise are not required for DC analyses, for example.

The post-processing block of code computes electrical quantities that are not required for simulator analyses. Typically this is operating-point data that are targeted for display, to provide useful information to designers. Examples are conductances, capacitance coefficients, unity gain (transition) frequency, and power dissipation. (Note that conductances and capacitance coefficients may be computed for use in Newton's method, and so the post-processing may just manipulate this information, for example summing capacitance coefficients at a node, for convenient display of useful information.)

3. Quick Review of Analog Circuit Simulation

This section summarizes how simulators handle the routines that make up a compact model.

Assume that the pseudo-code shown below refers to a circuit of four resistors and a voltage source.

```
.model myR resistor
+ rsh=1.0 tc1=0.001 tc2=-1.3e-4
r1 (1 2) model=myR w=2.0 l=1.0
r2 (2 3) model=myR w=3.0 l=1.0
r3 (3 0) model=myR w=1.0 l=2.0
r4 (3 0) model=myR w=1.0 l=2.0
v1 (1 0) dc=1.0
dc device=v1 dc=(1 3 4 5.5 6)
```

The last statement asks the simulator to run a DC analysis, comprising five DC simulation steps, where the voltage source value is changed to each value in the list.

Before starting any simulation the simulator initializes the model variables of each model declared in the netlist. Then it initializes the instance variables of each instance in the netlist by evaluating the instance initialization code. For this example, the simulator evaluates the resistor model initialization code once, for the model declaration referred to as `myR`, and for example computes the sheet resistance at the current simulation temperature. The simulator then evaluates the resistor instance initialization code four times, once for each resistor instance, to compute the resistance from the instance geometry and the model's sheet resistance.

After the initialization phase the simulator runs its first simulation step. At this point the topology of the circuit is frozen. The simulator engine solves for a solution to the given topology of the circuit. It executes the resistor evaluator as many times as necessary until convergence is reached. Although most simulators would solve a simple linear circuit in one iteration, assume that convergence is reached after 4 Newton iterations for the first two bias points and 6 Newton iterations for the remaining 3 bias points. The resistor evaluation is called 26 times for each resistor instance, or 104 times overall. After convergence is reached for each bias, the post-processing routine is called for each resistor (if printing of operating point information is requested).

In summary, the DC analysis involves:

1 call to the resistor model initialization

4 calls to the resistor instance initialization

104 calls to the resistor evaluation

20 (possible) calls to the resistor post-processing

This highlights the need to carefully design the routines that make up a compact model. Poor partitioning of compact model code leads to slow and inefficient implementation of a model. The current version of Verilog-A does not give a model author control over the code partitioning. However, analysis of the dependency tree of the model code allows automated partitioning. The proposed extensions to Verilog-A therefore do not include block partition specifications.

The next section gives an example of a compact model implemented in Verilog-A. It is a simple model for a capacitor. We will use this example throughout the paper to point out how extensions to the language could improve support of compact modeling.

4. Simple Compact Model in Verilog-A

Fig. 1 gives a Verilog-A implementation of a simple capacitor compact model (including deliberate problems).

At the top, a small drawing represents the equivalent network of the model. The compact model includes parasitic resistance (calculated as the distributed resistance of a top poly plate), and allows for first and second order voltage coefficients of capacitance.

All begin-end blocks of code are assigned a name that refers to the desired functionality of the block. These are the pieces of code that will make up the initialization, the evaluation, and the post-processing routines of the capacitor model, and make the code partitioning easily identifiable.

The model described in Fig. 1 has some limitations, and is not completely consistent with how compact models are normally defined for, and work in, a simulator. So modifications of, and additions to, Verilog-A are required to allow complete description of a compact model.

5. Proposed Extensions to Verilog-A

This section describes specific extensions to Verilog-A to make it better suited to definition of compact models. This includes aspects that deal with concepts central to compact models that are not implementable in Verilog-A, with aids to documentation, and with tighter linking of models and simulator algorithms.

5.1 Units

Units are a valuable, sometimes mandatory, piece of information for parameters and variables. We propose to

allow the specification of units for parameters and variables by use of a quoted string in the declaration statement. The units should follow the recommendations of "International System of Units" [7]. Besides making sure parameters are specified in the correct units and displayed variables are interpreted properly, the units field can also be used to automatically generate model documentation.

The unit is specified as a string, immediately following the default value:

```
parameter real w=1.0u "m";
```

This declaration states that parameter *w* is defined in meters and its default value is 1.0 micrometer.

5.2 Descriptions

Descriptions of parameters, terminals, and variables are useful for understanding what they are (names are not always obvious), as for example printed in a DC operating point analysis, and also for automatically generating model documentation. However, at present Verilog-A does not support a mechanism for defining descriptions of these quantities.

We propose adding a double-dash construct that indicates that the following entry (a string), which should be the last entry in a specification, be allowed for terminals and parameters, modules, and variables. It is expected that all external terminals and all parameters should have descriptions, and that variables that are printed as part of DC operating point information should also have descriptions.

For example:

```
parameter real w=1.0u "m" -- "width";
```

gives a description of the parameter *w*.

Note that it is not possible just to define the last string field in a declaration to be the description. The range specification for parameters is optional, so if this was not specified and there was one string field, then it would be ambiguous as to whether it was a unit or a description.

Good modeling practice would be that units and descriptions are explicitly defined for all parameters.

5.3 String Variables

As the previous examples have show, Verilog-A should be extended to include string data types. Specifically, for some parameters, like a device type (`p` or `n`) it is useful and more obvious to have them be of type string.

It is proposed to add a data type string to Verilog-A, and to be able to perform string comparison operations.

5.4 Detecting if a Parameter is Specified

Sometimes it is useful in a model to determine if a parameter is specified (in a model card or on an instance line). Specifically, if there are multiple ways of defining how a certain part of a model is to be calculated, and the default parameters are not, or are not able to be, defined in such a way that it can be unambiguously determined that one definition option is not to be used (for example by having a default of –1.0 or 9.99e99, which is inelegant), then being able to detect if a parameter is specified can direct how calculations of one aspect of a model are to be handled.

For example, in the MOSCAP varactor model [8] the substrate sheet resistance can be specified either directly, or via the substrate junction depth and resistivity. We propose to introduce a function `$param_given` that allows detection of whether a parameter is specified.

For the example defined in the previous paragraph

```
if ($param_given(rsub)&&$param_given(xj))
    rs=(rsub/xj)*(l/w)/8
else
    rs=rshsub*(l/w)/8
```

5.5 Model and Instance Parameters

One big difference between the existing way Verilog-A models are defined and handled in simulators and how compact models are treated is that for compact models parameters are sequestered into model parameters (defined on a .model card) and instance parameters (defined on a device instance).

We propose to allow the keywords `model` and `instance` to be used before a parameter declaration, to specify the type of the parameter:

```
instance parameter real w  =1.0u "m";
model    parameter real rsh=1    "Ohm/sq";
```

Details of model and instance parameter specifications are still being clarified. Note that at present Verilog-A parameters are instance parameters.

5.6 Parameter Aliases

It is sometimes convenient to be able to have aliases defined for parameters. There are two main situations where this is useful.

First, if there are multiple common names for one quantity it allows acceptance of them all. For example, in various simulators and models the local temperature rise with respect to ambient is variously referred to as `dtemp`, `trise`, and `dta`.

Second, sometimes the numeral zero (0) and the letters o or O are confused. So having aliases can avoid frustration.

```
model parameter real  vto=0.4;
      parameter alias vt0=vto;
```

The alias specification precludes both `vto` and `vt0` being specified on a model card, which is not preventable if both are declared as parameters. Model equations use `vto` only.

5.7 Interaction with Simulator Variables/Algorithms

Although conceptually a model should be independent of a particular simulator, specific numerical algorithms used in models, such as homotopy, require specific action by models. Also, there are global simulator parameters such as `gmin`, `shrink`, and `imax`, which models need to know.

Therefore interaction with simulator variables and algorithms can be necessary. For access to variables two forms are proposed.

```
gmin=$simparam("gmin",1);
```

would return an error if the simulator does not know the parameter gmin, else would return the value of `gmin`. And

```
gmin=$simparam("gmin",0,1e-12);
```

would return the value of `gmin` if it were known, else would return the third argument, here 10^{-12}. This third argument could be a default, or a value (like –1) that allowed unambiguous detection that the parameter was not known to the simulator, so conditional action could be taken.

Similarly simulator specific action is proposed to be specified by `` `ifdef `` conditionals, for which known vendor and simulator macros are defined. Details are still being discussed, but as these do not need to be evaluated at run-time they do not need to be defined as additional functions.

5.8 Access to Derivatives

Conductances and capacitance coefficients are useful quantities to report to designers, and are conventionally provided as part of an operating point analysis for compact models. When Newton's method is used during simulation these quantities are often available, but this is not guaranteed (for example when Jacobian bypass is used). Therefore we propose addition of a function `$ddx()` to explicitly allow access to derivative information, typically for operating point analysis. The example in Fig. 1 is simple enough to allow manual definition of the effective capacitance, although there is a deliberate error in the definition of `Ceff`, however for a complex model it can be tedious and error prone to define conductances and capacitance coefficients manually, and even minor updates to a model can require significant effort to recalculate and code derivatives. Also, a key attribute of Verilog-A is that it does not require derivatives and expects them to be calculated when needed.

Typical quantities calculated for a MOSFET would include:

```
gm = $ddx(Ids,V(g,s)); // transconductance
cgs=-$ddx(Qg,V(s,g))-$ddx(Qg,V(s,d))
    -$ddx(Qg,(Vs,b)); // gate-source cap
```

5.9 Optional Terminals

Some models include optional terminals. For example BJTs have historically been able to be defined with 3 or 4 terminals, depending on whether or not there is a substrate terminal. Also, models that include self-heating may want to have a self-heating terminal available for thermal coupling to adjacent devices, or may want the self-heating terminal hidden if there is no interaction with adjacent devices.

These two cases require different actions. For the first, calculations associated with elements connected to the substrate terminal are bypassed, and the topology of the equivalent network changes. For the second, the self-heating node is still required for simulation, and the equivalent network topology is not changed, and the self-heating terminal is effectively open circuited. Having multiple optional terminals raises other issues. Evaluation of optional terminals is on going.

6. Example

Fig. 2 shows the simple capacitor model with many of the extensions described above. All terminals, parameters, and output quantities have descriptions, and the last 2 have units defined (the narrow column width makes it a bit cluttered to read). The parameters are declared as either instance or model parameters. Any variable (here `Ceff` and `pwrR`) declared at the top level, outside the analog block, is an output variable.

Expecting that code partitioning is automatic, the named blocks used to sequester the code have been removed.

A string variable for device type is included, and this allows polarity dependent calculations to be done more obviously, rather than relying on use of a real parameter as a flag. The handling of the integer number of contacts is also done in a cleaner manner.

The calculation of the output display quantity `Ceff` now does not require manual (and error prone) calculation of a derivative, but is specified directly.

7. Conclusion

The goals of this paper are

to promote Verilog-A as a standard language for compact model development and definition

to highlight the need for extensions to Verilog-A to support the full functionality needed by compact models

This paper has defined some extensions to Verilog-A that make it more suited to the description of compact models. These extensions are now being refined by a subcommittee and will be proposed to Accellera for incorporation in future definitions of the Verilog-A language.

References

[1] http://www-device.eecs.berkeley.edu/~bsim3/

[2] L. Lemaitre, C. McAndrew, and S. Hamm, “ADMS – Automated Device Model Synthesize,” *Proc. IEEE CICC*, pp. 27-30, 2002.

[3] M. Mierzwinski, P. O’Halloran, B. Troyanovsky and R. Dutton, “Changing the Paradigm for Compact Model Integration in Circuit Simulators Using Verilog-A,” *Proc. Nanotech*, pp. 376–379, 2003.

[4] L. Lemaitre, C. McAndrew, and W. Grabinski, “Standardization of Compact Device modeling in High Level Description Language,” *Proc. Nanotech*, pp. 372-375, 2003.

[5] Open Verilog International, “Verilog-AMS, Language Reference Manual,” Version 1.9, Dec. 15, 1999.

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

[7] http://www.bipm.fr/enus/3_SI/base_units.html

[8] J. Victory, C. C. McAndrew, and K. Gullapalli, “A Time-Dependent, Surface Potential Based Compact Model for MOS Capacitors,” *IEEE EDL*, vol. 22, no. 5, pp. 245-247, May 2001.

```
//
// simple capacitor model
//  t    R    i   C   b
//  o──/\/\/──o──| |──o
//
`define TNOM (`P_CELCIUS0+27.0)
module    C (t, b);
 inout       t, b;
 electrical  t, b;
 electrical  i;
 branch     (t, i) R;
 branch     (i, b) C;
 parameter real w   =1u from (0.1u:+inf);
 parameter real l   =1u from (0.1u:+inf);
 parameter real nc  =1  from [1   :2   ];
 parameter real rsh =1  from (0   :+inf);
 parameter real ca  =1f from (0   :+inf);
 parameter real tcr =0;
 parameter real vc1 =0;
 parameter real vc2 =0;
 parameter real type=0; // 0=n, 1=p
 analog begin : module
  real dT, rsh_t, c, r, Ir, Qc, Ceff, pwrR;
  begin : initializeModel
   dT    = $temperature-`TNOM;
   rsh_t = rsh*(1.0+tcr*dT);
  end
  begin : initializeInstance
   c   = w*l*ca*1e12; // unit conversion
   if (nc>1.5)
    r  = rsh_t*(w/l)/12;
   else
    r  = rsh_t*(w/l)/3;
  end
  begin : evaluateStatic
   Ir  = V(R)/r;
  end
  begin : evaluateDynamic
   if (type>0.5) // inelegant
    Qc  = c*V(C)*(1-V(C)*(vc1/2-V(C)*vc2/3));
   else
    Qc  = c*V(C)*(1+V(C)*(vc1/2+V(C)*vc2/3));
  end
  begin : loadStatic
   I(R) <+Ir;
  end
  begin : loadDynamic
   I(C) <+ddt(Qc);
  end
  begin : postProcess
   if (type>0.5)
    Ceff = c*(1.0-V(C)*(vc1/2-V(C)/3));//error!
   else
    Ceff = c*(1.0+V(C)*(vc1/2+V(C)/3));//error!
   pwrR = V(R)*Ir;
  end
 end // analog
endmodule
```

Fig. 1 Simple Capacitor Model

```
//
// simple capacitor model
// with Verilog-A extensions
//  t    R    i   C   b
//  o──/\/\/──o──| |──o
//
`define TNOM (`P_CELCIUS0+27.0)
module    C (t, b) -- "capacitor model";
 inout       t, b;
 electrical  t     -- "top plate terminal";
 electrical  b     -- "bottom plate terminal";
 electrical  i     -- "internal node";
 branch     (t, i) R;
 branch     (i, b) C;
 instance parameter real w    =1u "m"
       from (0.1u:+inf) -- "width";
 instance parameter real l    =1u "m"
       from (0.1u:+inf) -- "length";
 instance parameter integer nc=1  ""
       from [1   :2   ] -- "# contacts";
 model    parameter real rsh  =1  "Ohm/sq"
       from (0   :+inf) -- "sheet resistance";
 model    parameter real ca   =1f "F/um^2"
       from (0   :+inf) -- "cap per area";
 model    parameter real tcr  =0  "/C"
                          -- "R temp coef1";
 model    parameter real vc1  =0  "/V"
                           -- "C voltage coef1";
 model    parameter real vc2  =0  "/V"
                          -- "C voltage coef2";
 model    parameter string type="n" ""
                           -- "bulk type (n/p)";
 parameter alias rs = rsh;
 real Ceff "F" -- "effective capacitance";
 real pwrR "W" -- "power dissipation";
 analog begin : module
  real dT, rsh_t, c, r, Ir, Qc, ty;
  dT    = $temperature-`TNOM;
  rsh_t = rsh*(1.0+tcr*dT);
  if ($param_given(type) && type=="p")
   ty =-1;
  else
   ty =+1;
  c   = w*l*ca*1e12; // unit conversion
  r   = rsh_t*(w/l)/(3*nc*nc); // integer nc
  Ir  = V(R)/r;
  Qc  = c*V(C)*(1+V(C)*(ty*vc1/2+V(C)*vc2/3));
  I(R)<+Ir;
  I(C)<+ddt(Qc);
  Ceff= $ddx(Qc,V(C)); // no manual error
  pwrR= V(R)*Ir;
 end // analog
endmodule
```

Fig. 2 Capacitor Model with Extensions