CMC Policy on Standardization of Verilog-A Model Code

For compact models accepted as standards by the Compact Model Coalition (CMC), CMC policy states that the Verilog-A description of the model defines the standard. Any compliant implementation of the standard model shall give the same outputs (terminal currents, noises, and operating-point information) as the standard Verilog-A code, when provided with the same standard parameters (names and values) and applied biases. Simulator vendors may add additional parameters and provide additional operating-point information to implementations in their tools at their discretion.

This document describes the aspects of the standard that are determined by the Verilog-A code. A set of CMC macros is provided as an appendix of this document.

General Principles:

QA testing

A guiding principle for this document is to ensure that the CMC QA tests can be easily run with either the Verilog-A code as delivered or with the built-in model. This provides the end-user with the ability to verify that a particular implementation of a model is compliant with the standard. In some scenarios, switching from Verilog-A to the built-in model is as easy as removing the "verilogaFile" option in the test setup file (usually named "qaSpec"). One might also have to change the model name set by nTypeSelectionArguments (or pTypeSelectionArguments); however, there should be no need to change terminal names or parameter names.

Additionally, users may want to run a simulation in which an existing built-in model is replaced by a Verilog-A description of a new release (such as beta code).

Syntax

Verilog-A is the analog subset of the Accellera standard Verilog-AMS, defined at present by the Language Reference Manual (LRM) version 2.4.[1] All references to Verilog-A or Verilog-AMS in this document refer to this version of the LRM, which can be downloaded from https://accellera.org/images/downloads/standards/v-ams/VAMS-LRM-2-4.pdf . Verilog-AMS is case-sensitive and uses ASCII characters (not Unicode), and thus Verilog-A modules must conform to these conventions. Module names, parameter names, and variable names are called "identifiers" in Verilog-AMS, and the syntax for a simple identifier is expressed (in Backus-Naur form) as

simple_identifier[2] ::= [a-zA-Z_] { [a-zA-Z0-9_$] }

[1] Accellera is presently working on integrating analog and mixed-signal extensions into IEEE Standard 1800 System Verilog.

[2] From Annex A.9.3 of VAMS LRM 2.4.

This syntax means that a simple identifier must start with a letter between ‘a’ and ‘z’ or ‘A’ and ‘Z’ or an underscore, and then is followed by any sequence of letters, numbers, underscores, and dollar signs ($).

For the purposes of this document, an additional syntax is defined, which restricts the letters to lowercase. The syntax for such identifiers is

interface_identifier ::= [**a-z**] { [**a-z0-9_**] }

If a module uses lowercase letters for interactions from the “outside” (namely, the simulator), these modules will match the conventions found in case-sensitive simulators. Note that the parameters listed in Annex E of the LRM, SPICE compatibility, are all lowercase. Specific requirements of the following section shall determine the restrictions of the identifiers used for the module name, parameter names, operating-point names, and noise names. By restricting these to only lowercase names, there is no possibility of two identifiers (such as a parameter and an operating-point variable) having the same name, which could cause confusion in case-insensitive simulators.

Any variables that are strictly internal to the module may use uppercase or mixed-case names; use of the dollar sign ($) is discouraged.

Specific Requirements:

1. Module name

The module name shall be derived from the name of the model. The module name shall be an interface identifier, as defined above. Models that were developed by the CMC, or for which the CMC owns the copyright, shall have module names ending with “_cmc”. Models developed by other groups shall append “_va” to the model name.

Examples:

```
module r2_cmc(n1,n2);
module hisimhv_va(d, g, s, b, sub, temp);
```

2. File name

The name of the primary file (i.e., the file that contains the module name) shall be constructed by appending “.va” to the model name, in lowercase. Additional files may be included from the primary file.

3. Terminal names

Terminals shall be named with lowercase letters. Although the terminal names are not important in many Spice-like implementations, where terminal connections are made by ordered list rather than by name, the names are important when reporting terminal currents or operating-point information such as capacitances. For models that correspond to SPICE primitives, terminals (referred to as “ports” in Verilog-AMS) shall use the names found in Table E-1 of Annex E of the Verilog-AMS LRM. (These names are all lowercase.)

Examples:

```
module hicum_va(c, b, e, s);
module hisimhv_va(d, g, s, b, sub, temp);
```

4. Parameters

Parameters shall be named with interface identifiers as defined above. They shall be declared using the CMC macros (MPRcz, MPRnb, etc.). Appropriate units, ranges, and descriptions should be supplied. Specifically, any parameter that cannot physically be negative shall have a range that excludes negative values. Recommended macros for parameter declarations are provided as an appendix of this document.

Examples:

```
`MPRoz(tox, 3n, "m",   "oxide thickness");
`MPRcz(rd, 0.0, "Ohm", "drain resistance");
```

For those model developers that want some indication of which identifiers (names of variables or parameters) correspond to parameters, a simple macro may be helpful:

```
`define PAR(x)    x
```

Equations in the module can then be written using the macro

```
Cgd = `PAR(cgdo) * weff;
```

where Cgd and weff are variables and cgdo is a parameter.

5. Operating-point values

Operating point values shall be declared using the CMC macros (OPP, OPM, OPD) with appropriate units and descriptions. The names of operating-point variables shall be interface identifiers.

In order that QA test results match between the Verilog-A and built-in, and so that simulations replacing the built-in with newer Verilog-A should give the same results, the values of operating point values are part of the standard. These values are also available for access via the CMC Open Model Interface (OMI), and the values must be consistent across simulators. If a value in a simulator's built-in implementation differs by a minus sign from the Verilog-A, that implementation is not compliant with the standard.

Examples:

```
`OPP(vsat, "V", "saturation voltage")
`OPM(iavl, "A", "avalanche current")
`OPD(reff, "Ohm", "effective resistance")
```

6. Noise source names

All noise sources shall be given a name by providing the optional last argument to the `white_noise` and `flicker_noise` functions in Verilog-A. The name shall be an interface identifier unique to the physical source. Since the Verilog-A LRM requires that noise from all sources with the same name be combined, it is better to name “rd” and “rs” for source and drain resistance, rather than lumping all resistor noise together by naming them all “thermal.”

Examples:

```
I(d,di) <+ white_noise(fourkt*gd,"rd");
I(s,si) <+ white_noise(fourkt*gs,"rs");
```

However, if there are two (or more) contributions for the same physical noise, e.g. different `flicker_noise` calls based on a parameter like noimod, then these names should be the same. (Do not use names like “flicker_noimod1” and “flicker_noimod2.”)

Example:

```
if (noimod == 1 || noimod == 4) begin
      I(di,si) <+ flicker_noise(kf * pow(idc,af), ef, "flk");
else if (noimod == 13) begin
      I(di,si) <+ flicker_noise(kf * gm *gm / CoxWL, af, "flk");
end
```

Note in this example the frequency exponent differs in the two cases.

7. Reference temperature

If the model requires a reference temperature to indicate the temperature at which the specified parameters are valid, the module shall declare a parameter named tref or tnom.

Example:

```
`MPRco(tref, 27.0, "degC", -273.15, inf, "Reference temperature")
```

This parameter should never be obtained from the simulator, which would prevent accurate simulations of models that require different values of tref.

8. Temperature offset

The module shall provide a parameter to specify the temperature offset of the device ambient temperature (excluding self-heating) from the circuit ambient temperature. This parameter shall be named dtemp, and trise shall be provided as an alias (using the Verilog-A feature `aliasparam`).

Example:

```
`IPRnb(dtemp, 0.0, "K", "Device temperature offset")
aliasparam trise = dtemp;
```

Having a consistent name for a temperature offset for all instances simplifies the process of specifying such an offset for a circuit or portion thereof, such as might be done for electrothermal simulation.

9. Gmin

Model developers have responsibility for inserting gmin currents and should follow the recommendations of the CMC gmin subcommittee. The value of gmin shall be obtained by calling the simulation parameter function

```
$simparam("gmin", 0.0)
```

Note that the default value is 0.0, to avoid adding artificial leakage in a simulator that does not use or require gmin.

10. Scaling

For certain processes (generally, fine-line CMOS processes), the layout is scaled or shrunk optically to take up less area in silicon. Often, this scaling can be applied directly to the shapes in an existing layout, without involvement of the layout engineer or circuit designer. If a layout is scaled to 90%, then linear dimensions in the layout, such as length, would be scaled by 0.9, but areas would be scaled by the square of this factor, or 0.81.

While it is generally obvious what scaling rules should be applied to common parameters such as length (l) and width (w), there are many additional parameters for layout-dependent effects; additionally, some dimensions, such as fin heights, are not affected by the scaling. Therefore, the scaling rules for a model shall be implemented in the Verilog-A code by using one of the two scaling macros defined in the appendix, `LS and `QS, to specify the scaling; the actual scaling operations shall be done by the simulator. Note this scaling only applies to instance parameters, which are determined by the layout; model parameters are not scaled, as the model parameters are assumed to be extracted from wafers of the scaled process. (For parameters that are both model and instance, the scaling shall only be applied to parameters specified on the instance line.)

With this macro:

```
`define LS , scale="linear"
```

the parameter declaration:

```
`IPRcz(length, 1.0u, "m" `LS, "length")
```

(where `IPRcz` is defined in the appendix) would expand into this:

```
(* units = "m" , scale="linear", desc="length" *)
    parameter real length = 1.0u from [0.0 : inf);
```

Note the comma is part of the definitions of the macros `LS and `QS, and there is no comma between the units and scaling macro. This way, we do not need to redefine all the parameter macros to accept an additional argument.

Since the Verilog-A code defines the model standard, if the Verilog-A code does not implement scaling rules, then the standard does not include scaling; a simulator that is compliant with the standard shall not perform any scaling.

11. Warning and error messages

Warning messages shall be reported using $warning. The %m specifier can be used to identify the instance that is generating the warning.

```
$warning("Warning (%m): message");
```

Error messages shall be reported using `$error`. This function (technically, a "system task") gives more explicit instruction to the simulator. Historically, models have use the `$finish` task:

```
$strobe("ERROR: message");
$finish(0);
```

However, `$finish` in some contexts simply means the analysis is done (e.g., enough cycles have been run or a conversion has finished), without implying any sort of error condition. The two lines above shall be replaced with one:

```
$error("ERROR: message");
```

The `$error` task is preferred to `$fatal`, because simulators should print all error messages if more than error condition is in effect, whereas simulators may abort after the first `$fatal` message.

12. Physical constants

Models should define their own physical constants, to ensure the same values are used for the model as implemented in different simulators or parameter extraction tools. In order that CMC standard models for different devices in the same manufacturing process are consistent, the values of the elementary charge (e) and Boltzmann constant (k) should use the exact values defined for the International System of Units (SI), namely:

Elementary charge	1.602176634e-19
Boltzmann constant	1.380649e-23

The SI units were redefined in 2019 to make these values exact. These values are also listed in the CODATA 2018 and CODATA 2022 internationally recommended values; see `https://physics.nist.gov/cuu/Constants/index.html`

The Verilog-AMS 2023 Language Reference Manual (LRM) defines macros for these values (`P_Q_NIST2018` and `P_K_NIST2018`), which will become available in simulators that support the new LRM.

Implementation timeline:

All new models should be reviewed for compliance with these guidelines before acceptance as a standard. Existing standard models should endeavor to adopt these guidelines in subsequent releases, but they are not required to release new code simply to implement the recommendations. Any changes that break backwards-compatibility should be postponed until the release of a new version, following the version numbering policy in the “CMC Compact Model Release Specification” policy document.

Appendix: Recommended macros for CMC compact models

```
//  Macros for the model/instance parameters
//
//  MPRxx    model parameter real
//  MPIxx    model parameter integer
//  IPRxx    instance parameter real
//  IPIxx    instance parameter integer
//  BPRxx    both (model and instance) parameter real
//     ||
//     cc    closed lower bound, closed upper bound
//     oo    open   lower bound, open   upper bound
//     co    closed lower bound, open   upper bound
//     oc    open   lower bound, closed upper bound
//     cz    closed lower bound = 0, open upper bound = inf
//     oz    open   lower bound = 0, open upper bound = inf
//     nb    no bounds
//     ex    no bounds with exclude
//     sw    switch (integer only, values  0 = false  and  1 = true)
//     ty    switch (integer only, values -1 = p-type and +1 = n-type)
//
//  LS    linear scaling for instance parameters
//  QS    quadratic scaling for instance parameters
//
//  OPP   operating point parameter, includes units and description for printing
//  OPM   operating point parameter, multiply value by $mfactor (eg: currents, charges)
//  OPD   operating point parameter, divide value by $mfactor (eg: resistances)


`define MPRnb(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter real    nam = def;
`define MPRex(nam, def, uni, exc,      des) (* units = uni,                    desc = des *) \
    parameter real    nam = def exclude exc;
`define MPRcc(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from[lwr : upr];
`define MPRoo(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from(lwr : upr);
`define MPRco(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from[lwr : upr);
`define MPRoc(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from(lwr : upr];
`define MPRcz(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from[0.0 : inf);
`define MPRoz(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter real    nam = def from(0.0 : inf);

`define MPInb(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter integer nam = def;
`define MPIex(nam, def, uni, exc,      des) (* units = uni,                    desc = des *) \
    parameter integer nam = def exclude exc;
`define MPIcc(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from[lwr : upr];
`define MPIoo(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from(lwr : upr);
`define MPIco(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from[lwr : upr);
`define MPIoc(nam, def, uni, lwr, upr, des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from(lwr : upr];
`define MPIcz(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from[0 : inf);
`define MPIoz(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from(0 : inf);
`define MPIsw(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from[0 : 1];
`define MPIty(nam, def, uni,           des) (* units = uni,                    desc = des *) \
    parameter integer nam = def from[-1: 1] exclude 0;
```

```
`define IPRnb(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def;
`define IPRex(nam, def, uni, exc,      des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def exclude exc;
`define IPRcc(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from[lwr : upr];
`define IPRoo(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from(lwr : upr);
`define IPRco(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from[lwr : upr);
`define IPRoc(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from(lwr : upr];
`define IPRcz(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from[0.0 : inf);
`define IPRoz(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter real    nam = def from(0.0 : inf);

`define IPInb(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def;
`define IPIex(nam, def, uni, exc,      des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def exclude exc;
`define IPIcc(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from[lwr : upr];
`define IPIoo(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from(lwr : upr);
`define IPIco(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from[lwr : upr);
`define IPIoc(nam, def, uni, lwr, upr, des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from(lwr : upr];
`define IPIcz(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from[0 : inf);
`define IPIoz(nam, def, uni,           des) (* units = uni, type = "instance", desc = des *) \
    parameter integer nam = def from(0 : inf);

`define BPRnb(nam, def, uni,           des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def;
`define BPRex(nam, def, uni, exc,      des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def exclude exc;
`define BPRcc(nam, def, uni, lwr, upr, des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from[lwr : upr];
`define BPRoo(nam, def, uni, lwr, upr, des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from(lwr : upr);
`define BPRco(nam, def, uni, lwr, upr, des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from[lwr : upr);
`define BPRoc(nam, def, uni, lwr, upr, des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from(lwr : upr];
`define BPRcz(nam, def, uni,           des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from[0.0 : inf);
`define BPRoz(nam, def, uni,           des) (* units = uni, type = "both", desc = des *) \
    parameter real    nam = def from(0.0 : inf);


`define LS , scale="linear"
`define QS , scale="quadratic"


`define OPP(nam, uni, des)    (* units = uni,                          desc = des *)  real  nam;
`define OPM(nam, uni, des)    (* units = uni, multiplicity="multiply", desc = des *)  real  nam;
`define OPD(nam, uni, des)    (* units = uni, multiplicity="divide",   desc = des *)  real  nam;
```

Appendix: Revision History

Changes from 1.2.1 to 1.3.0 (2024-June-14): Add section on physical constants.

Changes from 1.2 to 1.2.1 (2023-Mar-20): Replace OPP by OPM for iavl example; remove hyphens from “lower-case.”

Changes from 1.1 to 1.2 (2022-Dec-9): Add scaling section and macros, add BPRxx macros.

Changes from 1.0 to 1.1 (2020-July-27): Section 8 Temperature offset, example changed from `MPR(dtemp, …) to `IPR(dtemp, …)

Initial version 1.0 (2019-Dec-12).