Get the Best Performance From Your Verilog-A Model

Introduction

Although models written in Verilog-A particularly transistor models, will simulate out-of-the-box, the following policies should be followed to obtain better performance from SmartSpice.

New models which follow these policies will also perform better.

Outline

1. Computation of parametric values
2. Internal node count
3. The noise block
4. Macros instead of functions
5. GMIN!

Policies

1. Compute parametric values only once:

```
parameter length = 2;
parameter width = 3;
real     area;

analog begin
   area = length * width;
   ….
end
```

Here, area is repeatedly computed at each iteration. However, the computations after the first are unnecessary.

This is because area is derived solely from length and width which are declared as parameters, it is illegal to modify their values at runtime (LRM 2.3.1 Sec. 3.4), and since variable values are retained through subsequent iterations (LRM 2.3.1 Sec. 5.6.1.3), it is possible to compute area only once at the start of the simulation.

This change achieves the desired result:

```
parameter length = 2;
parameter width = 3;
real     area;

analog begin
   @(initial_step) begin
    area = length * width;
    ….
    end
    ….
end
```

The event block triggered by the *initial_step* event executes only once at the start of simulation (LRM 3.2.1 Sec 5.10.2). Verilog-A also recognizes the *initial_step* and categorizes the assigned variables differently to considerable benefit.

Some models (PSP, BSIMCMG) which directly support ADMS make insertion of the *initial_step* simple. These models use "`define" to provide flexibility for use outside ADMS:

```
`ifdef ADMS
   `define INITIAL_MODEL@(initial_model)
   `define INITIAL_INSTANCE
                       @(initial_instance)
`else
   `define INITIAL_MODEL
   `define INITIAL_INSTANCE
`endif

analog begin
   `INITIAL_MODEL
   begin
     area = length * width;
     …
   end
end
```

When not targeting ADMS, "`INITIAL_MODEL" and "`INITIAL_INSTANCE" are substituted by space, which is not what we want. The following simple change remedies this:

```
`ifdef ADMS
   `define INITIAL_MODEL @(initial_model)
   `define INITIAL_INSTANCE
                      @(initial_instance)
`else
   `define INITIAL_MODEL @(initial_step)
   `define INITIAL_INSTANCE
                        @(initial_step)
`endif
```

2. Minimize the number of internal nodes.

Because parasitics in the model reduce the speed of simulation, models are typically implemented with different levels. The basic level implements fewer parasitics than the highest level, uses fewer internal nodes for simulation, and thus simulates faster. The Verilog-A LRM has no facility comparable to SPICE's level parameter, so other means have been discovered to achieve a similar result. One simple approach is to have a different model name (module name) for each level. A more complex approach requires the Verilog-A parser to automatically deduce the proper level based on the parametric values provided. These different Verilog-A code styles achieve the result of different levels from one Verilog-A source model. Two styles, *conditional compliation* and *node collapse* are discussed below:

In *conditional compilation*, (e.g. EKV3 model) localized use of "`ifdef"s provide unique model (module) names for each level and precise control over the internal node count:

```
`ifdef DC
        module xyz       (d,g,s,b);
`endif
`ifdef RF
        module xyz_rf    (d,g,s,b);
`endif
….
`ifdef RF
        electrical internal_d,
        internal_s, internal_b;
`endif
….
`ifdef RF
        I(internal_d,internal_s) <+ ….;
`endif
        endmodule
```

When the module is compiled with "RF" defined, the unique module name becomes "xyz_rf", precisely 3 internal nodes are added to the model, and a contribution across the internal nodes is present. These features occur *conditionally* on "RF" being defined, hence the name *conditional compilation*. Note the *conditional compilation* style is applied during the compilation phase.

In *node collapse* (e.g. PSP, BSIMCMG) a special Verilog-A if-then-else construct is employed and the Verilog-A parser is expected to recognize and then act on the construct during the simulation setup phase to differentiate the model into a specific level:

```
parameter rg;

if(rg != 0)
  I(g, internal_g) <+ V(g, internal_g)/rg;
else
   V(g, internal_g) <+ 0;
```

In this code, when the parasitic "rg" is non-zero a resistor with value "rg" is placed between nodes "internal_g" and "g". When the parasitic is absent, i.e. "rg" is 0, a short is made between the nodes by the 0 volt voltage source. The if-then-else construct forms a switch set by the constant parameter "rg" and the Verilog-A parser is expected to recognize during setup that when "rg" is zero the switch is always "on", nodes "g" and "internal_g" are shorted, and thus the two nodes can be collapsed into one, reducing the node count, Hence the name, *node collapse*.

While it is true that a zero-volt voltage source shorts its terminal nodes allowing *node collapse*, the Verilog-A LRM specifically reserves the use of a zero-voltage source for a different purpose. Specifically, when a current probe is needed a zero-volt source should be used (LRM 2.3.1 Sec 5.4.2.1). Verilog-A does not implement *node collapse* but interprets the zero-volt source as per the LRM.

Verilog-A correctly simulates the if-then-else construct, but neither zero nor non-zero values of "rg" improve the internal node count. There is always one node for "internal_g" and, worse yet, an internal branch node is reserved for current through the voltage source. There are always three nodes, even when "rg" is zero in which case only one is necessary.

We recommend searching the Verilog-A model for this if-then-else construct and replacing it with the *conditional compilation* style to reduce the internal node count where applicable.

3. Do not execute the noise block when noise analysis is not performed.

When noise analysis is not requested, calculations on behalf of noise analysis are 0 valued and thus can be avoided. Fortunately noise analysis statements often appear grouped together in a Verilog-A model:

```
I(…)  <+ white_noise(…);
I(…)  <+ white_noise(…);
….
```

And this simple solution avoids perfoming the calculations until needed:

```
if (analysis("noise")) begin
   I(…)  <+ white_noise(…);
   I(…)  <+ white_noise(…);
   ….
end
```

4. Use macros instead of function calls where it makes sense:

Function calls are expensive to execute. When a function is frequently called replace it with a macro:

```
analog function real myexp;
   input x;
   real x;
   begin
      myexp = exp(x);
   end
endfunction

analog begin
   y = myexp(z);
end
```

This can easily be changed into a macro expansion:

```
`define myexp(_x) (exp(_x_))

analog begin
   y = `myexp(z);
end
```

5. Watch out for GMIN!

Models that require a value for "gmin" may hard-code a "gmin" value into the model:

```
   `define GMIN 1.0e-12
```

or

```
   parameter real GMIN = 0.0;
```

In this case, the SmartSpice option for changing circuit "gmin" will not reach the model. Make the model respond to the smartspice "gmin" option by replacing references to GMIN with:

```
   $simparam("gmin")
```

Conclusion

Applying the five described policies to your Verilog-A model results in optimal code for the best simulation performance in SmartSpice. Application of the policies to existing models can often be completed in as little as thirty minutes by an engineer familiar with the Verilog-A language. This is easily recovered by the savings in simulation time through the optimized model.