General considerations about compact modeling
  - a need for standardization

Introduce compact modeling and SPICE3 kit
  - SPICE3 kit using C language
  - SPICE3 kit using Verilog-AMS language and model compilers

Introduce new solution using GCC front end
  - Give advantages of this solution

Conclusion
In the 70’s:
- One simulator and few devices
  - no need for standard

In the 90’s:
- many simulators are available
- more and more compact models

ISSUE
- same model BUT different simulation results between SIMULATOR A and SIMULATOR B.

SOLUTION
- CMC proposed to use SPICE3 as a reference
- SPICE3 kits for compact models are delivered to EDA companies to qualify the compact model into the simulator
SPICE3 Kit
CMC Standardization

COMPACT MODELING TEAM
- Write code in SPICE3 format
- Provide test bench in SPICE3 language
- Provide up-to-date documentation of the model

EDA COMPANY
- Qualifies its customized implementation of the compact model by running SPICE3 test bench successfully
- Send feedbacks to compact modeling team
SPICE3 kit interacts with:

1. The netlist parser
   - specifies the syntax in spice netlist
2. The output processor
   - specifies how results are displayed
3. The Jacobian matrix
   - Simulator must solve $I - J \cdot V = 0$
   - Kit specifies how to stamp the CM into the Jacobian $J$
4. The solvers
   - interacts with solvers to improve convergence
   - this is poor approach: simulation results may be simulator-dependant!!
More devices - More simulators available
- Each simulator support specific syntax
  - SPICE3 CM coding is customized
  - SPICE3 CM test bench are customized

This means:
- SPICE3 kit requires manual changes
- SPICE3 test bench requires manual changes
- Partial Derivatives are computed by hand

AS A CONSEQUENCE
- Simulators can have inconsistent implementation
- Difficult to know which simulator is right:
  - #define CHARGE 1.6E-19
  - #define CHARGE 1.602E-19

Next slide gives an example of SPICE3 kit and its complexity
SPICE3 kit are complex. They are made of lot of files. Grand Total : 8K lines of code!

**COMPLEXITY**
- Strong dependencies between files
- Partial derivatives of equations are done by hand
- Very easy to break consistencies between files

**WORKS BUT TOO COMPLEX**
- Next slide presents a new approach adopted by most CM developers

Numbers in black are number of lines in source code
SPICE3 Kit in Verilog-AMS language

SOLUTION:
- Implement compact models in the Verilog-AMS language
- Use Model Compiler to create SPICE3 Kit or Simulator Specific implementation

ADVANTAGES
- SPICE3 kit created automatically
- SPICE3 test bench can be specified in the Verilog-AMS language
- Reduce discrepancies between simulators of the same model

DRAWBACKS
- Push all simulators to have efficient Verilog-AMS model compiler
- Loss in performances:
  - CM run slower
  - CM have bigger memory usage
Example of Verilog-AMS Coding

- Syntax closed to C language - makes it easy to learn
- The CM has about 20 lines of code
- The equivalent SPICE3 kit would contain thousand of lines
Experiment:
1. from original handcrafted SPICE kit reverse engineer BSIM3 in Verilog-AMS
2. create new SPICE3 kit using a model compiler
3. run transient analysis
   - check that Newton-Raphson iterations are identical
   - check that simulation results are identical

Results and Conclusion:
- Speed is 20% slower
- Memory is 10%-20% more

Conclusion:
Performances of models compilers are OK
Verilog-AMS Compilers not dedicated to compact modeling:
- Poor performances observed
- Possible explanations:
  - Verilog-AMS design models have hundred of lines, few variables, event-driven
  - Compact models have thousands of lines, hundreds of variables and not event driven - UNEXPECTED

Alternative:
- use compilers dedicated to Compact Modeling
  - ADMS
  - Tiburon
  - ...

Source: Verilog-A Compact Model Coding Depeyrot al. 2010 NSTI
ADMS creates the SPICE3 kit at once
- ADMS is a source to source compiler
- ADMS translates Code A to Code B
  - Code A is the Verilog-AMS language
  - Code B is the C language for the SPICE3 API
- Instructions to convert from code A to code B are provided in xml language
- Note that code B could be pdf format for documentation applications
It is a three steps process

- This command adds some back annotation information to the code to improve the performances of the final implementation.

- This command creates the SPICE3 kit of the rpoly CM.

- This command creates the NGSPICE dynamic library of the rpoly CM.

- Next slide proposes a more compact approach using a GCC front-end.
Compilation
- Process that converts code written in C language, C++ language, ... into machine binary code
- In GCC this step is done in two steps:
  - create an internal tree representation called GIMPLE representation
  - dump GIMPLE representation into binary code specific to targeted CPU

Linking
- Process that put all together binary codes and makes an executable binary file

Building
- Process that combines both sequences: compiling + linking

Our approach is to create the GIMPLE representation directly from the Verilog-AMS language: IT IS CALLED GCC Verilog-AMS Front-End
ADMS does not produce C language code
ADMS produces GIMPLE code
GCC compiles GIMPLE code and produces .o object file
GCC links .o object to ngspice interface and produces .so file
Advantages:
- avoid intermediate coding in C language
- take advantage of GCC optimization techniques
- speed-up the process of building models into simulators
- make possible interactive applications that require fast re-compilation
- get gdb debugger for free

Drawback
- Approach tied to GCC compiler
- user forces to adopt GCC compiler
ADMS creates the GIMPLE tree representation from the Verilog-AMS code.

Edges: relationships between different pieces of code used by the GCC compiler.

GCC compiler techniques can be applied to ADMS:
- tree reduction
- pattern matching
- dead code detection

Simplified view of the GIMPLE tree of a Verilog-AMS model.
### Example of Optimization Code Partitioning

<table>
<thead>
<tr>
<th>Number of evaluations</th>
<th>Partitioning in SPICE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>few times</td>
<td>model section (process variables, doping, temperature)</td>
</tr>
<tr>
<td>some times</td>
<td>instance section (geometries)</td>
</tr>
<tr>
<td>many many times</td>
<td>load section (bias dependant)</td>
</tr>
</tbody>
</table>

Move invariant code can have a very big impact on speed performance.

Optimization done by applying tree operations on the GIMPLE tree.
Easy to implement and debug.
Compact modeling can easily be done using the Verilog-AMS language. Using a dedicated model compiler speed and memory performances using Verilog-AMS is acceptable. ADMS can create a Verilog-AMS front end for the GCC compiler in order to speed up the building of Verilog-AMS models into simulators. Applications that require fast re-compilation of Verilog-AMS code become feasible:

- fast re-computation of partial derivatives for sensitivity analysis
- fast re-computation of partial derivatives used by advanced algorithms in compact model characterization

Finally I would like to acknowledge the great work done by MOS-AK and Wladek Grabinski for the adoption of the Verilog-AMS language as compact modeling language.

THANK YOU!