INTRODUCTION
Verifying that a specific implementation of a processor is fully compliant with the specification is a difficult task. Due to the very large total stimuli space it is difficult, if not impossible, to ensure that every architectural and micro-architectural feature has been exercised. Typical approaches involve collecting large test-suites of real SW, as well as using program generators based on constrained-random generation of instruction streams, but there are drawbacks to each.

SystemVerilog Constrained-Random (CR) ‘single class with constraints on all possible fields approach’ can be unwieldy and hard for solvers to handle, especially when dealing with a series of instructions with many for each constraint. Slow solving performance, testbench bugs hidden in the complexity, or simply not having the flexibility to fully describe the wanted scenarios for verification are quite common issues. When the problem is adequately modeled, achieving significant coverage is close to impossible.

There are custom options other than SV CR – creating an instruction stream generator from scratch, for example C++ may seem like a viable option, but the complexity and vast solution space quickly makes this approach unmanageable. Moreover, languages like C++ have no notion of functional coverage, which makes it hard or impossible to assess when the verification is complete.

In this article we present an alternative approach, using a portable, environment independent model of the stimulus. The CPU instruction set, and various contexts in which the instructions appear, are all modeled by a graph.

LEVERAGING A NEW PORTABLE STIMULUS APPROACH
Graph or rule based stimulus descriptions have been used for a long time in software and recently in hardware, including in CPU verification for modeling instruction streams in RT-level verification.

The key elements of this methodology are as follows:

- A declarative graph is used to completely model the Instruction set
  - Compact and intuitive format – well known in the SW community
  - Based on Accellera Proposed Standard for Portable Stimulus
- Separate Coverage model describes scenarios and corner-cases
  - E.g. Combinations of opcode, source and destination, interesting sequences of instructions...
- Graph is loaded in simulation together with DUT and traversed systematically to fulfill the coverage goals

The graph is described in a compact and intuitive format (BNF-like) – well known in the SW community. The format is based on Accellera Proposed Standard for Portable Stimulus – meaning that stimulus is independent of the simulation environment so the same model can theoretically be used with DUTs at any level in any language (or even used for other code generation or documentation purposes).
MODEL EXAMPLE
Here’s an example of modelling one of the instruction types (SETHI/BRANCH) using SystemVerilog, compared with the same example using the graph-based notation. As we can see, even for this simple example the result is a set of nested if-statements and complex constraints, which quickly makes it hard to debug and impacts solver performance in a negative way.

As a comparison, a single rule in the graph-based notation completely describes all forms this particular instruction type can take:

The visualization of this particular rule looks like the graphic in figure 4:

By building a hierarchy of these simple rules it is easy to create a highly structured model of any instruction set in a very compact and readable format.

SIMULATION INTEGRATION
One key element of the graph notation is the interface. An interface is an abstraction of the connection between the graph and the rest of the verification environment. By separating the connection from the graph implementation it is possible to achieve a high degree of horizontal re-use, since the same graph can drive many different verification environments. In a UVM environment for example, a graph can be used to fill the member fields of a UVM sequence item, replacing the calls to randomize() as shown below. The same graph could also be used to call functions in a library of C/C++ functions running on an embedded CPU.
BENEFITS FOR FUNCTIONAL COVERAGE
The graph-based description can be augmented with information about the specific verification goals, as described perhaps by functional coverage metrics in SystemVerilog. Smart traversal algorithms can ensure that all instruction types are broadly exercised, hitting all corner cases efficiently.

The table below was taken from trial runs using a subset of the Sparc V8 instruction set that compared the two approaches.

In each case # of instructions’ is both the number of coverage bins and the number of instructions actually generated during the simulation.

MODELING PIPELINE CONSTRAINTS
Many modern GPU’s and DSP’s are based on Very Long Instruction Word (VLIW) architectures, where one instruction word can contain multiple parallel instructions, all executed in the same cycle(s). Due to the particular pipeline implementation and resources, there might be micro-architectural constraints that apply. For example, it might be possible to combine an ADD and a MULT instruction, but not two MULT since the pipeline only has one multiplier.

To model pipelines or sets of parallel instructions, a slightly different graph can be used, where each instruction is stored and the next one is generated following rules defined.

A new level of hierarchy is created as described earlier with two instances of a sub-graph that includes valid instructions that can be executed in parallel with another.

<table>
<thead>
<tr>
<th>TESTCASE</th>
<th># OF INSTRUCTIONS</th>
<th>GRAPH COVERAGE</th>
<th>C·R COVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Opcodes</td>
<td>74</td>
<td>100%</td>
<td>61%</td>
</tr>
<tr>
<td>Arith Op X Dest reg</td>
<td>1600</td>
<td>100%</td>
<td>54%</td>
</tr>
</tbody>
</table>

Table 1. Graph based simulation coverage vs Constrained-random
One of the useful aspects of the graph notation is that graphs can be made reactive. By using an interface as described above, a graph can read states and variables from the DUT or testbench, and modify its behaviour based on the state(s). One way to take advantage of this is to create a separate independent graph to generate interrupts. This graph would monitor the current state of the CPU, and when a specific condition occurs, like a specific instruction or sequence of instructions, an interrupt is injected.

SUMMARY
In this article we have shown how graphs can be used to efficiently model CPU/GPU/DSP instruction sets. A single graph can model an entire instruction set, or separate graphs can emphasize different scenarios, re-using modular sub-graphs of certain types or groupings. This approach enables efficient coverage closure of scenarios and corner-cases, e.g. combinations of opcode, source and destination, or interesting sequences of instructions fed into a pipeline.

Also, graph models are independent of language or simulation environment, and are re-usable at any abstraction level.

A demonstrator model is presented, based on the Sparc V8 instruction set.

REFERENCES
The SPARC Architecture Manual, Version 8, Sparc International Inc.
25 Video Courses Available Covering
• Formal Verification
• Intelligent Testbench Automation
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• PowerAware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 6000 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com