Meta Design Framework

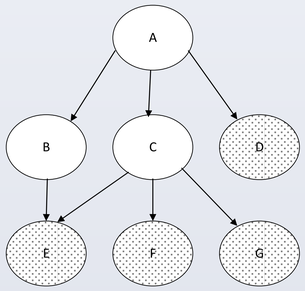
Sanjeev Singh, Jonathan Sadowsky

Juniper Networks

Building Designs Programmatically

A framework using an open source object oriented language “Ruby” is presented to build complex Verilog designs programmatically.

A design is comprised of leaf level Verilog modules and accompanying design hierarchy as shown below. Our framework translates the leaf Verilog modules into Ruby classes. The design hierarchy is then created as a data structure using object classes and their methods within the programming language. The framework contains a toolkit to convert the design hierarchy data structure into Verilog output containing instantiations and signal connectivity.



Objectives

RTL design is getting way too entangled with generic RTL logic being mixed with implementation specific logic like Vendor specific library elements, backend logic like BIST etc.

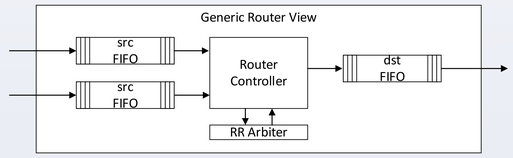
We would like to “untangle” the hardware design by keeping these components separate and using a program to create the final Verilog output.

Our objective is

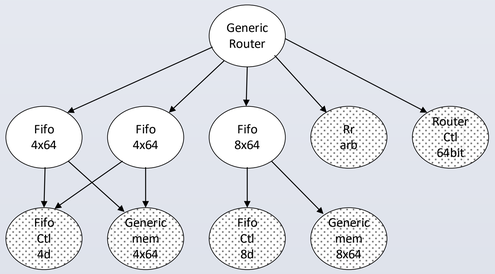
1. Keep RTL generic logic pristine across implementations.
2. Able to keep the rules for stitching and creating the final design in one place.
3. Able to incrementally reuse/modify the rules using OOP

Generic Router Design

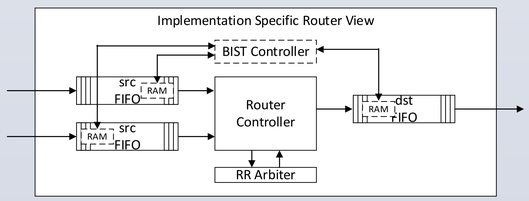
To demonstrate the framework a generic router was designed. This router consisted of two source FIFOs, a destination FIFO, a round robin arbiter, and a router control module. These were all leaf level Verilog modules except the FIFOs which were wrappers around a FIFO controller and a generic memory.

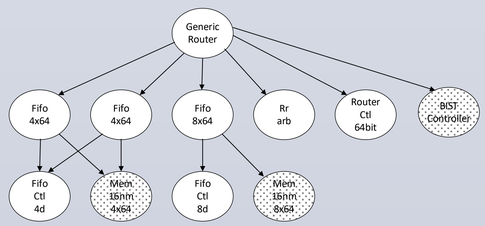


The Ruby script was used to create the hierarchy as shown below. The shaded class objects are proxy class objects for the already written Verilog leaf modules. The unshaded class objects were created by the Ruby script. All connectivity overrides were part of the Ruby script.



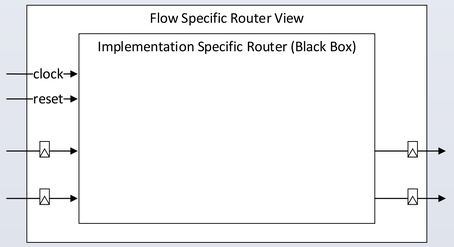
Adding Vendor Memory Cells

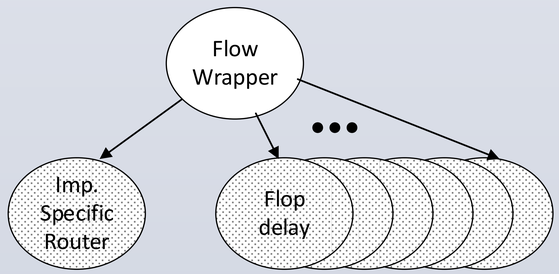




Adding Router Flops

Finally, we wanted to demonstrate that the meta design framework could be used to support multiple independent levels of hierarchy. In order to do this, we envisioned a flow which reused the implementation specific design as a black box, and added logic around that black box. The new design simply added delay elements on each input and output port except clocks and reset as seen below.





Conclusion

We were very pleased to be able to successfully create a generic RTL design and keep it pristine across multiple implementations using the meta design framework described in this paper.

Readability and reusability of the design were improved by separating the differing implementations into different design scripts (instead of using Ifdefs or multiple files).

Re-verification of existing variations was not required as new variations were created.

In addition, we were able to take advantage of the power and flexibility of an advanced programming language to build the design which made complex connectivity simple.

We also believe that the described framework will increase designer’s creativity when building complex Verilog designs

References

[1] Snyder, W. (n.d.). Verilog-mode Emacs. Retrieved from http://www.veripool.org/wiki/verilog-mode

[2]Matsumoto, Y. (n.d.). Retrieved from https://www.ruby-lang.org/

[3]Snyder, W. (n.d.). Verilog-Perl. Retrieved from http://www.veripool.org/wiki/verilog-perl

[4]Wall, L. (n.d.). Retrieved from https://www.perl.org/

Contact

Sanjeev Singh (sanjeevs@juniper.net)

Jonathan Sadowsky (jsadowsky@juniper.net)

Code Available at https://github.com/sanjeevs/verilog_gen

Download From https://rubygems.org/gems/verilog_gen