Meta Design Framework: Building Designs Programmatically

Sanjeev Singh, Jonathan Sadowsky
DVCon March 3, 2015

***Abstract*-A framework is being presented to build complex Verilog designs programmatically. A design is comprised of leaf level Verilog modules and accompanying design hierarchy. These leaf level Verilog modules are converted into object classes in an object oriented programming language. 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. Building the design using an object oriented programming language allows the designer the full processing power of the language to create programmatic designs. Furthermore, the framework allows the designer to take previously built designs (data structures) and modify them for the variations required for specific implementations.**

I. INTRODUCTION

This paper covers the concept of using a meta design framework for building complex Verilog designs. This paper addresses some of the issues that are present when building designs using static hierarchy (pure Verilog code, or other parsed code like Verilog-Mode Emacs [1]). A solution to these problems is then presented using a programming language along with a defined set of tools, methods, and processes. This paper then presents an implementation of this solution using the Ruby [2] object oriented programming language along with various tools available online. Finally, a sample design is coded to demonstrate how the framework can be used to generate design hierarchy along with implementation specific variations.

II. THE PROBLEM OF STATIC HIERARCHY

Building hierarchy and connectivity in Verilog designs is currently being done with static files. A file contains the instance declarations for sub-modules, as well as port/connectivity mappings. This can be done several different ways, all similar. The designer codes the file statically, perhaps with some assists to help with connectivity where port names are automatically wired (e.g., the auto-instance feature in System Verilog). If more than one configuration is desired, a designer can add Ifdefs to the code in order to have different compile time variations. This results in code that can be difficult to read, as well as difficult to modify in future. This is because one can't always easily tell what code is being executed as Ifdef structures can become very complex. This also increases design risk as each change to the golden design requires existing variations of the design to be re-verified. In order to add the new feature (e.g., with Ifdefs), the original design and all previous variations must be re-regressed to ensure that the change has not broken any of those design views.

Furthermore, although there are tools to help with the automation of connectivity, they still require effort on the part of the designer to be able to maximize their potential. For example, instancing multiple copies of the same module requires that the designer manually map the ports as auto-instancing is not capable of prepending a prefix. There are tools like Verilog-Mode Emacs which can do this prepending, but this requires that the designer understand the rules of the tool being used, which can be very different based on which tool is selected. This is especially difficult for designers to read when working on modules previously coded using a tool they are unfamiliar with.

III. PROGRAMMATIC HIERARCHY

This paper presents a solution to problems of static hierarchy. Rather than having hierarchy and connectivity of design be defined in static files, this solution proposes that hierarchy and connectivity of a design be defined as a data structure of an object oriented programming language. The designer can use scripts to build design data structure, as well as modify existing data structures. Using the full power of a programming language to define port mappings, allows complex naming rules to be expressed succinctly. Finally, a toolkit is provided to handle all auto-instancing of ports that are not explicitly overridden by the designer.

A. Verilog Module Class Object

The meta design framework is envisioned to be implemented using an object oriented programming language. The basic class in the framework is the Verilog module object. This object contains all of the methods and data structures required to build up the hierarchy of the design, as well as convert the design into Verilog output files. There are two flavors of this object, a full-flavored version, and a 'proxy' version. Only difference between the two is that the 'proxy' object is a placeholder in the data structure for Verilog files that already exist (e.g., the leaf level Verilog). These 'proxy' classes do not need to be "built", but they are required such that the other classes in the data structure can reference them and query them to determine information about the module they are representing. As shown in Fig. 1, the designer would create a data structure of these objects. The shaded objects are 'proxy' class objects. In addition to creating a data structure to represent the hierarchy of the design, the designer has the ability to modify an existing data structure. Table 1 shows the object methods required to create and/or modify the design hierarchy.

B. Parameter Management

Some Verilog modules are parameterized and the framework should support the ability to query and modify those parameters. The toolkit supplies the methods in Table 2 in order to manipulate parameters. The most complicated issue with parameters is when evaluating a child's port's attributes. For example, if a child port is a vector of size [CPARAM:0], then the parent needs to know how to locally create that port's connected wire declaration. If the child's parameters are not modified, and CPARAM defaults to 3, then the wire connected needs to be declared as [3:0]. If a parent links CPARAM to the parents parameter: CPARAM=PPARAM*2, then the wire needs to be defined as [PPARAM*2:0] and the parent, when its RTL is created, needs to have a PPARAM defined so it can be set by subsequent parents.

C. Connectivity Methods

The base class object has built-in methods to handle auto-wiring of ports, as well as query methods to determine ports and attributes of ports. Table 3 lists out methods available for connectivity. When writing a new class, connectivity programming supplied by the designer are contained within a connectivity method which is not executed until after the complete hierarchy is created. This is important as the program, outside of the class object being created, can modify the hierarchy of this class object.

D. Creating Verilog

Once the designer has completed the definition of design hierarchy, parameter linking, and any connectivity overrides, it is time to build the design and create Verilog. Each class is responsible for creating its own Verilog module as an output. The methods in Table 4 are used for managing the build. This is a recursive process as all children are required to be built before the parent module is built. This is because the children's ports aren't necessarily known until the child has been built. For example, in Fig. 1, object *A* cannot be built until object *C* is built because a higher level program may have deleted an old child class from object *C* and replaced it with object *G*. So, the parent must first call the build method of all of its children, and only then can it run the methods required to write out Verilog. Once it is built, its ports are known and its parent can then build.

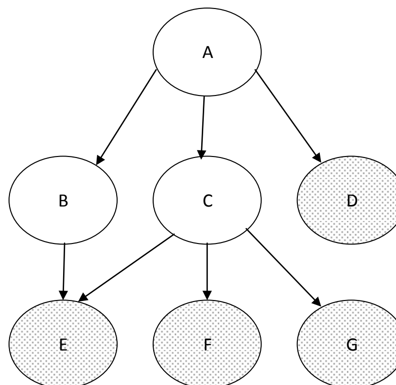

Figure 1: A Data Structure of Class Objects Defines the Design Hierarchy

TABLE 1: OBJECT METHODS FOR MANIPULATING THE DESIGN HIERARCHY

Method	Description
Create Proxy Class	A method which is provided, as input, a pointer to an existing Verilog module. The module is parsed to determine the ports, port attributes, and parameters of the module. This returns a new class which inherits from the Verilog Module class object. This method sets the 'proxy' flag to indicate that this references existing Verilog. The designer can then add this into the data structure being generated. In Fig. 1, objects labeled *D*, *E*, *F*, and *G* were created using this method.
Add Child	This takes a Verilog Module class object and adds it to the list of child objects. The object can be either created using the 'Create Proxy Class' method, or it can be created more programmatically by creating a new instance of the Verilog Module class object. For example, in Fig. 1, when the class for A is created, it has additional code to add in children *B*, *C*, and *D*.
Delete Child	This takes a relative reference from the object where the method is being called and removes a class object from the data structure. For example, in Fig. 1, if class *A* is reusing *C*, but wants to delete *G*, it would reference *C.G* when it calls its delete method.
Replace Child	This is a shortcut combination of calling Add Child and Delete Child individually.

TABLE 2: MANAGING DESIGN PARAMETERS

Method	Description
Set Parameter	The parent instance can call this method to link a child's parameter to a value or parameter of the parent. This does not set the parameter, rather, it just relates the parameter to internal state of the parent. When the child is instanced, the parameter in the instantiation is set. When child ports are queried, parameters in ports are evaluated and replaced with the new value.
Defaults	If child parameters are not overridden by a parent, the default values of parameters are used for connectivity management.

TABLE 3: CONNECTIVITY METHODS

Method	Description
Get Port Object List	A class object (including 'proxy' class object) contains methods to return information about the ports of the object. This returns a list of port objects which can themselves be inspected.
Get Port Object	Given a specific name as input, returns the port object that has a name matching the input.
Port Methods	Ports have methods to return information about their attributes including: - Direction (input/output/inout) - Type (wire/reg/Interface/etc.) - Packed vector declaration - Unpacked array declaration
Auto-Wire Methods	The following auto-wire rules are applied to the design (these have a lower priority than user specified overrides): - A port will be connected to a signal of the same name as the port. - If a child instance has a port as an input and another child instance has a port with the same name as an output, the two ports are connected. If multiple children have the port as inputs, all are connected. - If all child instances have the same output port name then a wire of width equal to the sum of the widths of all the child ports is connected as an output port of the parent module. - If all child instances have input ports of the same name, then a wire of the maximum of the child port widths is connected as an input port of the parent module. - Note the toolkit does not check for illegal or non-synthesizable behavior, rather, it is left up to the lint tools to perform this task.
Connectivity Overrides	The user can specify that for a port, it should be connected to something else other than a wire of the same name as the port. Once the wire to port mapping has been given, the newly created wire follows the rules in the auto-wire method section of this table.
Port overrides	Some signals that would normally become outputs of the parent module need to be suppressed as the child instances output is not used. There need to be methods to prevent the auto-wire methods from wiring out of the port. Likewise, some child inputs might be required to be tied to constants and these methods allow this capability so they do not become parent input ports.

TABLE 4: BUILD METHODS

Method	Description
Build	First calls the build method of each child instance. This method controls the order of the build process.
Apply Overrides	Any user defined overrides for instance's ports are applied to map the port name to a different name.
Write Verilog	Writes the Verilog output files

IV. A PRACTICAL IMPLEMENTATION

The toolkit was built using open source tools. The supporting script to scan Verilog leaf level modules and generate proxy class objects for them was built using the Verilog-Perl [3] utility along with a Perl [4] based wrapper script. Remaining functionality was built using Ruby, an open source object oriented programming language.

Implementing build scripts in a true object oriented programming language like Ruby provided us flexibility to implement complex build scenarios. Like traditional OOP languages, it supports inheritance which allows functionality in the existing classes to be reused. Moreover, in Ruby, existing classes are never closed. One can always extend or override existing methods in any class. This allows incremental build scripts to be written which can open an existing design object and make the minimum number of modifications to support a specific implementation variation.

Using these features allows the designer to clearly express logic ideas like "Design A is the same as Design B except that the memory instances are replaced". This transformation logic is maintained in separate files and the changes can be made without modifying the original design files.

V. DEMONSTRATION OF UTILITY

In order to prove the claims being made, a sample design was created to demonstrate the meta design framework using Ruby and Perl scripts. A generic router (Fig. 2) was created using the tools and methods described earlier. 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 source FIFOs were 4 entries deep and the destination FIFO was 8 entries deep. The generic version of the router operates on a 64-bit data bus. The leaf level cells were written in pure Verilog. All hierarchy and connectivity was created in pure Ruby. The toolkit was then used to build the design.

The Ruby scripts for this view of the design built a data structure as seen in Fig. 3. 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. The design had a Makefile which ran the Ruby scripts to generate the resultant Verilog. A Verilog module was created by the toolkit for each unshaded class object with necessary instantiation and connectivity for the design.

For the demonstration, the generic design was modified for a specific implementation. For the purposes of this paper, the design was envisioned as being targeted for a view in a 16nm process. In this process, the generic memories in the FIFOs needed to be replaced with vendor specific SRAMs. Finally, a BIST controller needed to be added at the top level of the design (Fig. 4). This was to be accomplished without making any changes to the generic router Ruby scripts.

An implementation specific Ruby script was written that first called the generic router Ruby script in order to load the generic router view's data structure into memory. The new script then called the toolkit's 'replace' methods to replace the generic memories in the FIFOs with the vendor specific SRAM memories required. The toolkit was also used to add a new BIST controller instance at the top level. These changes to the hierarchy are shown in Fig. 5 shaded. The necessary connectivity overrides were also added to the script. The Makefile was modified to add a target for this specific variation of the design. Running make on the design resulted in the necessary RTL being generated. This was accomplished without any edits to the generic design scripts. The more traditional way of adding Ifdefs to create a variation would have forced us to rerun the generic regression and code coverage, etc.

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 in Fig. 6.

This view first builds the implementation specific router using the Makefile, and then in the Ruby scripts, the resultant top level Verilog module is included as a leaf proxy class object. The Ruby script also has a loop to query the ports of the implementation specific router and add delay elements for each port that does not match a pattern string for clocks and resets. Based on the port direction and width, the delay element was parameterized and wired appropriately. When the Ruby script was finally executed, it generated the Ruby class object data structure as shown in Fig. 7. The script also created a single Verilog wrapper module with all necessary instantiations and connectivity.

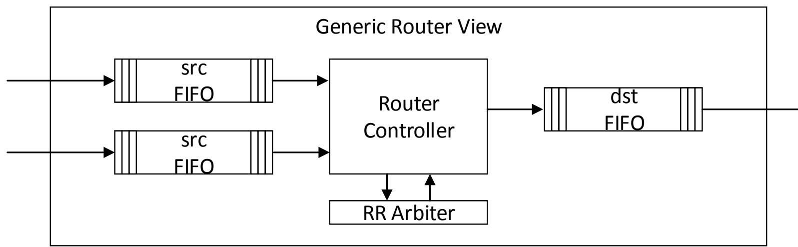

Figure 2: Generic Router Block Diagram

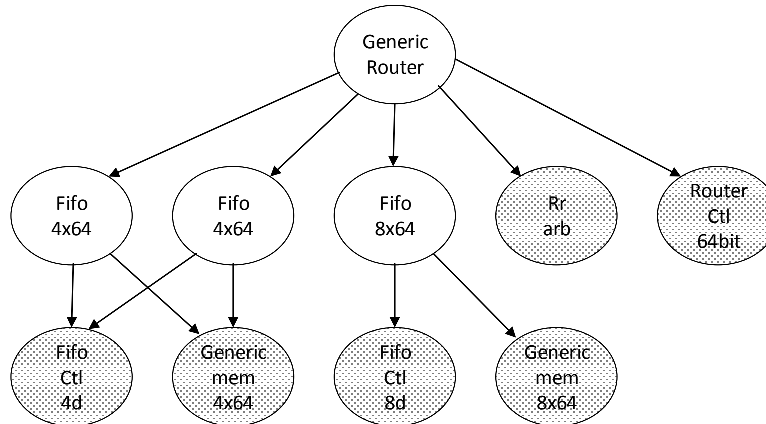

Figure 3: Generic Router Class Object Data Structure

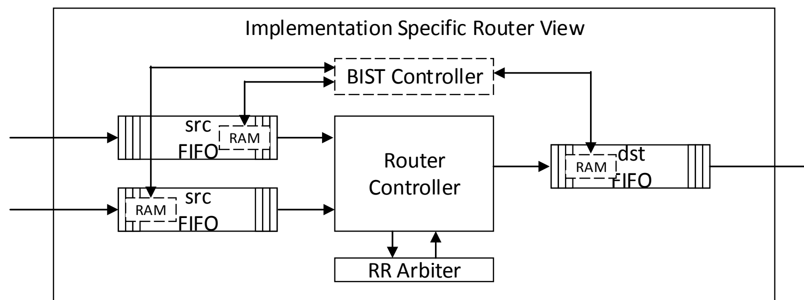

Figure 4: Implementation Specific View of Router

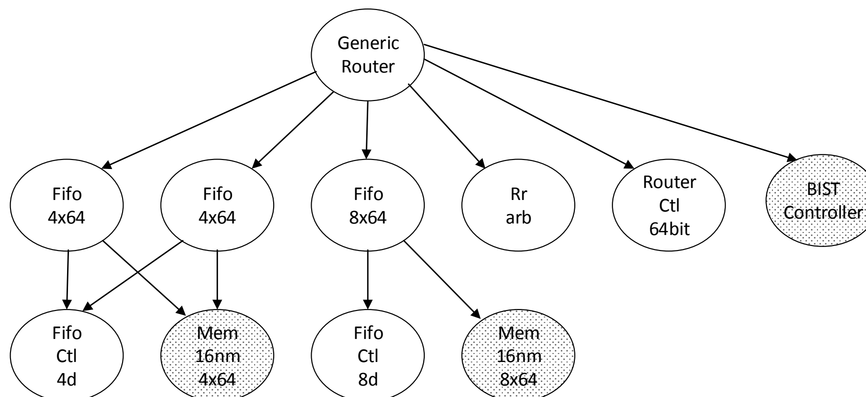

Figure 5: Implementation Specific Router Class Object Data Structure

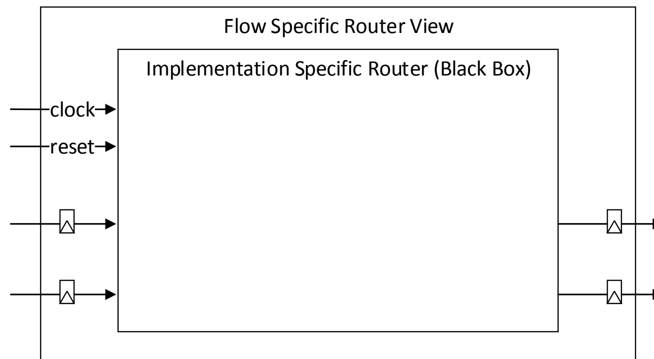

Figure 6: Flow Specific Wrapper around the Router

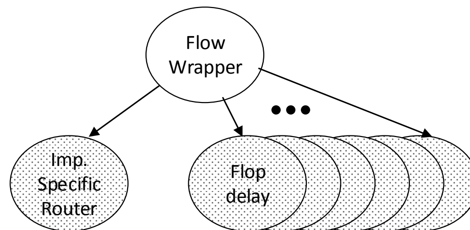

Figure 7: Flow Specific Wrapper Class Object Data Structure

VI. SUMMARY

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.

The framework is available for download as a Ruby Gem from [5]. Source code for the framework can be obtained from [6].

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/
[5] RubyGems https://rubygems.org/gems/verilog_gen
[6] GitHub https://github.com/sanjeevs/verilog_gen