Considerations for Development and Support of Exportable UVM IP

Thomas Loftus

AMD

September 18, 2013

SNUG Austin

Agenda

IP Export Challenges

Strategy

Development

Quality Control/Delivery

Conclusion

IP Export Challenges

Dynamic Role as Importer/Exporter

SOC Development Team Structure

- At AMD teams work at roughly three levels of IP hierarchy:
  - **IP creators:** low level (memory controller, CPU core)
  - **Module integrators:** lower level IP blocks, sub modules (data fabric, CPU core complex)
  - **SOC clients:** complete SOCs, large modules

Exporting, Importing, Vertical Reuse

- Export
  - Delivered IP, usually for use in higher level blocks
- Import
  - Received IP, usually of lower level in design hierarchy
- Vertical Reuse:
  - “Using a verification component in a different hiearchy level” [Litterick]
- Note: In the 3-tier model described, teams in the middle tier will find themselves both exporting and importing components

Use UVM Best Practices – Of Course

- UVM can be viewed as a set of recommendations with a supporting SystemVerilog library.
- UVM already encourages Reusability, Scalability, and Flexibility, stressing a development style that supports:
    - Functionality encapsulation
    - Transaction-level modeling
    - Active/passive configuration and connection
    - Functional checks and coverage
    - Messaging
    - Test Control
    - Sequences
    - Parameterized interfaces and classes

Exportable IP Has Extra Challenges

- Specifications are often incomplete or out of date
  - A stable "golden spec" is unlikely throughout development
- Dynamic technologies and product definitions
  - Imported components and export requirements change
  - SOC customers are at different stages of development
  - Customers will have changing requirements from sub-IP exporters
- Interactions between different tiers of the development hierarchy require iterative processes
  - IP is seldom delivered to the client without iterative revision

Handling The Extra Challenges

- **Bottom Line:** Verification is not getting any easier!
- This presentation features recommendations for addressing the extra challenges of developing IP in a multi-tiered, multi-client environment

Strategy

Pre-processing

- **Use code generators in your IP development …**
  - Auto-generation of UVM code can ensure scalability at compile-time
  - For example, auto-generated, customized functions using a base function can save coding time
  - If used in conjunction with an external “feature definition file”, multiple client topologies can be supported from one code base
- **Don’t require lots of pre-processing by the client**
  - Simplify pre-processing for exportable IP
  - Client wants a build process that is simple and straightforward
  - Require minimal or no pre-processing of the source by the client

Switches

- **Have a sensible coding standard for switches**
  - Don’t add switches for specific client projects. Switches should be for features
  - Ensures that switches make sense for the future

- Bad:

```
`ifdef PROJECT_X
```

- Good:

```
`ifdef QUAD_CORE
```

Parameters – Configuration

- **Testbench/RTL should be readily configurable ...**
  - Compile-time via configuration object parameters
  - Run-time via command line

- **But not too configurable!**
  - There is a cost in maintenance for making everything configurable
  - Look for balance between lower-level parameters and those supported in interfaces, balancing current vs future needs
  - This is subjective, but a hideously long macro that defines dozens and dozens of parameters … is probably bad

Parameters – Hardcoding

- **Don’t use hard-coded parameters at lower levels**
  - They tend to undermine UVM object orientation
  - They may be unavoidable, so minimize use if you must use them
    - If used, they should reasonably exceed anticipated values needed

      For example:

```
int mask = 16’hffff; // maximum mask value needed
```

Parameters – Parameterized Intf.

- **Don’t use parameterized classes.**
  - Parameters are compile time constant
    - Their values cannot come from run-time configuration objects
    - Forces literal numbers to be used at virtual interface definition
    - This limits scalability
  - Once you start using them, you have to use them all the way up the hierarchy:

```
class top #(parm1, parm2) …;
class lower #(parm1, parm2) …;
class bottom #(parm1, parm2) …;
…
virtual interface_a #(parm1, parm2) intf_0;
…
```

Parameters – Parameterized Intf.

- **Use parameterized interfaces where possible**
  - See paper by Yun & Zhang
  - minimizes hard-coded values
  - avoids complexity at higher class levels

```
class top …;
class lower …;
class bottom …;
…
abstract_a intf_0;
…
```

  - Parameterized interface class extension:

```
class concrete_a #(parm1, parm2) extends abstract_a;
```

Parameters – Parameterized Intf.

- **System Verilog OOP**
  - Classes are used to implement abstraction
  - Base class defines virtual methods
  - The derived class implements the virtual methods
  - Instances of the derived class can be used a one of a base class
  - Common bases class methods can operate on different derived classes
  - System Verilog supports this
    - A class can be used to abstract the parameterized interfaces

```
class BASE;
   pure virtual function void funct_0();
endclass

class DERIVED extends BASE;
  function void funct_0();
        // different implementation
  endfunction
endclass

BASE b;
DERIVED d = new;

b = d; // instance of derived used as base
```

Parameters - Packing

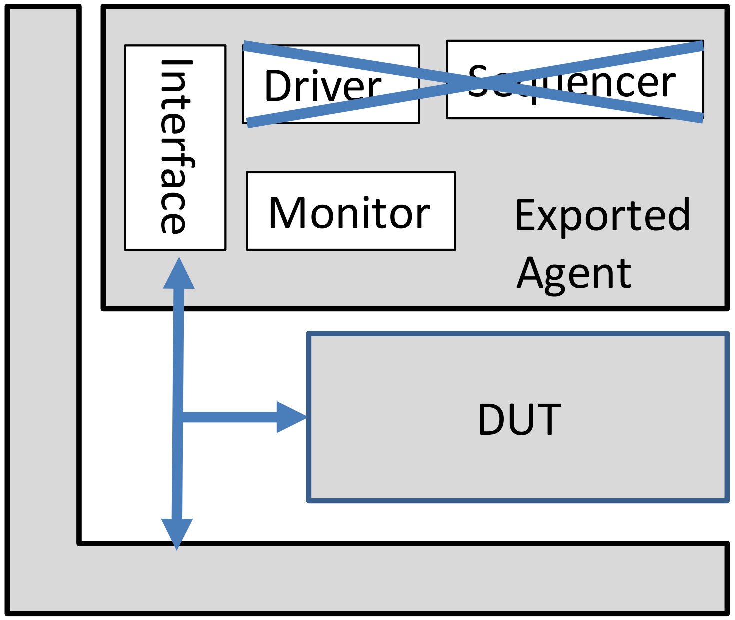
- **Don't pack tons of parameters into classes/interfaces**
  - Try to limit to values of use to the client at each level
    - Too many parameters can make use and maintenance difficult
    - Client should only have to control useful parameters
- **Don't pass many parameters through to top level**
  - Try to limit parameters available at top level
    - Passing parameters from lower levels to top may introduce needless complexity at higher levels
    - Push parameterization to lower levels as much as possible

Interconnect

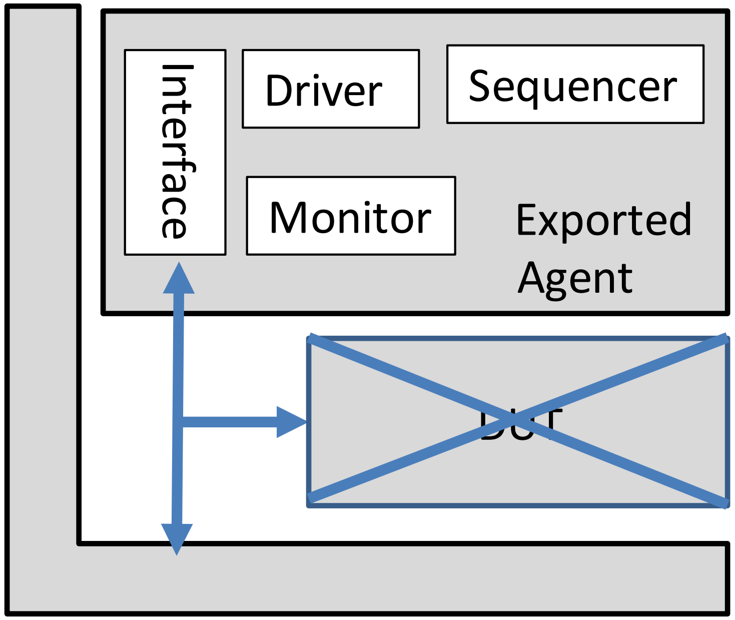
- **Don't rely on a manual process to set up interconnect**
  - In an iterative, vertical reuse environment, manually generating interconnect would cause enormous delay on every iteration
  - You need to automatically generate interconnect between modules with well-defined signal names
    - Ultimately faster and less prone to human error
    - Allows all tools to use signal names in build flow
    - Helps integrate IP with minimal effort to compile and to do initial reset

Active/Passive – Export Both.

- Passive
  - Exported as default



- Active
  - Explicitly selectable by client
  - May help client when some active components are not yet available in DUT



OVM vs. UVM Run Phases

- **Use OVM run phases**
  - Clients may not yet support UVM run phases
    - May not be necessary if your client has adopted UVM in a “from scratch” manner with no risk of having to support “legacy” OVM

Development

Directory Organization

- **Separate exportable IP from its testbench**
  - Only the exportable IP is supported, as a rule
  - Avoid time-critical "code bloat" by just updating the IP directory

- **Isolate the exported IP from client code**
  - Client testbench should be in its own directory, not within the exportable IP directory structure.
  - Client may build and check the IP testbench separately

Coding Guidelines

- **Make drain/idle checks switchable to warnings**
  - Incomplete transitions may be acceptable in client testbench
- **Raise objections in the testbench, not in exportable code**
  - Objections in exportable code can interfere with client processes
- **Include "begin" and "end" in all constructs**
  - Classic "gotcha," adding a quick debug message:

```
if (flag)
   `uvm_info("####", "flag is asserted", 0); // inserted
   do_flag_action();
```

  - vs

```
if (flag) begin
   `uvm_info("####", "flag is asserted", 0); // inserted
   do_flag_action();
end
```

Verbosity and Messages

- **Establish verbosity/message practices from the start**
  - Exported IP should generate few messages with UVM_MEDIUM
  - Qualify debug messages with UVM_HIGH or higher
  - Only a very few critical, one-line messages should qualify with UVM_LOW
  - Temporary messages are okay during development
    - Adopt a visible standard, easily spotted and removed:

```
`uvm_info(“####”, “message”, UVM_LOW);
`uvm_info(“####”, $sformatf(“value = %0d”, value), UVM_LOW);
```

    - Consider “####”, “BOZO” or “TODO” in development messages and comments.

Verbosity and Messages (cont.)

- **Utilize command line switches**
  - Make use of enabling of verbosity by component from the command line:

```
vcs +uvm_set_verbosity=*.my_component,_ALL_,UVM_HIGH,run
 vcs +UVM_VERBOSITY=UVM_NONE
   +uvm_set_verbosity=*.my_component,_ALL_,UVM_HIGH,run
```

- **Manage local Verbosity with functions**
  - Some functions can be included that are sensitive to local verbosity.
    - Derive components from uvm_report_object rather than uvm_object
    - Local verbosity can then be sensed with a local variable:

```
full_verbosity = uvm_report_enabled(UVM_FULL);

if(full_verbosity) begin // special printing or
```

Message ID Uniqueness (cont.)

- **Use unique IDs and include hierarchy**
  - Bad example:

    ```
    `uvm_info($sformatf("ID:%0d", my_variable), "message", 0);
    ```

    - This can result in the same message having multiple IDs making debug more difficult
  - Good example, with hierarchy:

    ```
    `uvm_info(sig_id("SIGNATURE_TAG"), "message", 0);
    ```

    - sig_id might be defined locally with something like:

    ```
    virtual function string sig_id(string sig_tag);
        return {"MY_COMP/NEXT_COMP_DOWN/", sig_tag};
    Endfunction
    ```

    - The client may customize sig_id to make the message unique across instances of the same IP

Quality Control/Delivery

Code Review

- **Enforce a common coding style for your team**
  - There should be lots of coding examples per the style
  - Enable others to readily inspect, comprehend and modify
  - Enforce for pseudo code and actual code
- **Periodically audit created UVM component instances**
  - Check that generated instances are actually used/needed
  - Delete unused interfaces, model instances, queues and so forth
- **Build exportable IP early and often**
  - With all available testbenches and all levels of hierarchy
  - Catch changes that affect other testbenches as soon as possible

Code Delivery

- **Encapsulate your IP with a package file**
  - Include a package file that brings in the exported IP
  - Documentation should include a guide for use
  - Include all sources for client to peruse as desired
- **Include needed imported IP with the exported IP**
  - Include hooks to enable connection to alternative imported IP
  - Include an example testbench with both exported and imported IP
- **Allow the client access to IP source**
  - As a rule, the client should be able to regard the IP as "black box."
  - But, the client should be free to make temporary, local modifications pending the next IP release

Code Delivery (cont.)

- **Minimize compile warnings, none if possible**
  - Clients should be free to promote warnings to errors in exported IP.
  - Only a handful of well-documented warnings should be allowed, never to be promoted to errors
- **Include a UVM configuration database object**
  - It must be well-documented with clear examples of usage
  - Client should readily be able to:
    - Disable checkers
    - Disable scoreboards
    - Control component verbosity
    - Access other useful configuration options
- **Try to export only pure UVM**
  - Conversion may be necessary for imported IP (such as legacy AXI OVM)
  - Client may mix later (including documentation)
  - Sync documentation with specifications with each release

Code Delivery (cont.)

- **Document and support important internal variables**
  - Though contrary to object-oriented theory, clients often want this
  - Document which variables are accessible, usually via function calls
  - Limit to a small set of variables of interest to the user
- **Document all IP analysis ports**
  - Include descriptions of, for example, specific analysis port "write_*" functions created by UVM
  - Include descriptions of expected port sources ("*_ap") and destinations ("*_xp")
  - Without these steps the client may struggle to figure out the IP analysis port connections

Conclusion

Conclusion

- **UVM is a powerful verification methodology**
  - It encourages flexibility, reuse and scalability
- **Exportable IP Has Extra Challenges**
  - Multi-tiered, Multi Client SOC verification
  - Chip size/complexity increasing
  - Managing the iterative delivery of IP is getting harder
- **This paper discusses guidelines for these challenges**
  - We hope you find some of these guidelines useful in managing this complexity

  – **THANK YOU!**