

Using PSL and FoCs for Functional Coverage Verification

January 2007

Chuck Alley

Introduction

IBM has recently been recognized as the company that developed the CPU's for each of the 3 major video game consoles on the market today. These processors are quite complex “Server Class” processors and were global efforts for IBM. In RTP we developed the Vector Scalar Unit (VSU), a multi-pipe vector and floating point processor unit, for for the Microsoft Xbox 360.

I used this project, code named “Waternoose”, to develop a strategy for functional coverage using a mixture of both internal (IBM Proprietary) and externally available tools.

- PSL Property Specification Language (IEEE 1850)
- FoCs Formal Checkers (<http://www.alphaworks.ibm.com/tech/FoCs>)
- Bugspray (IBM Proprietary Coverage Collection Tool)

Introduction

The Waternoose VSU team inherited a coverage strategy, and bugspray code, from a previous project. This existing bugspray code may have provided a useful report card but was not an adequate for functional coverage:

- The designer developed Bugspray code was almost entirely structural or code coverage
- Did not focus on areas of maximum complexity or areas of the logic prone to bugs
- The designer driven Bugspray coverage points were not documented in any test plan or workbook
- The VHDL of the Bugspray was hard to maintain as the design diverged from the antecedent design

Our Method

- We were unhappy with this method so we made the following changes to fully implement functional coverage
- The verification team took over the coverage plan
 - This included maintenance of the old bugspray and development of new coverage points
- We adopted PSL as the coverage language
- We translated the PSL code into VHDL bugspray code with FoCs
- We documented the coverage points in the test plan, and reviewed them as part of our verification of the unit

Core Verification Concerns

Several concerns came to my attention from the Austin core verification team, as we revealed this plan to them.

- Generated VHDL is bloated and inefficient and would seriously degrade the simulation speed at the core and chip levels
- Generated VHDL is hard to read and debug inhibiting anyone else from debugging problems that may come up or be revealed by our code
- Only designers know enough about the logic to write effective monitors, so how are you going to get your information and how will you maintain the bugspray

Bugspray Organization

The Waternoose project leadership had organized the bugspray into the following priority scheme:

- P1 Required for Unit Gold criteria
- P2 Required for Unit RIT criteria
- P3 Required for Core Gold criteria
- P4 Required for Core RIT criteria
- ...

Unit P1 and P2 monitors were instantiated in the local shell files and not passed up to the core team. The P3 and P4 monitors were in `unit_top` to be passed to the core.

Bugspray File Organization

For instance, the file *fw_p3_vmx_load_forwarding_bugspray.vhdl* needs to be included and covered at the core level. However, the file *fw_p2_vector_float_bugspray.vhdl* needs only to be run and covered in a unit simulation environment, and will not be promoted to the core level for simulation. Other features:

- Proper grouping of coverage events allows verification engineers to turn off coverage checking when coverage levels are achieved
- All coverage events were included in the unit simulation environment. This was because we wanted to develop def files capable of achieving all coverage points to save time and effort at higher levels of simulation.

Identifying Coverage Events

- Meetings are required with the design team to help identify coverage points.
- We went from there to the available project documentation (specs and workbooks) to find the specific details that we needed to code the monitors.
- We covered IO cases, areas of complexity, areas very likely to cause problems, complex control scenarios that were handled by multiple units/designers

Writing Monitors

A simple example:

- We are interested in monitoring for several interesting forwarding scenarios. We determine that the criteria are as follows:

Monitor that data is forwarded to a valid instruction from every possible source. Make sure that the instruction that uses this data is not canceled in rf1, so that the data is checked when it writes back to the VRF by IBI.

For this problem the PSL code would look like this:

```
cover {[*]; i1_vq10_vsu_instr_v; i1_vq11_vmx_b_mux_cntl(0..5) = %bypass_source; true[*2];  
i1_rf1_cancel=0};
```

Writing Monitors

- There are however, many sources from which the forwarding can occur So this code can be very quickly extended:

```
vunit vector_bypass_i1Bmux{  
  
  %for bypass_source in { i0_by1, i0_by2, i0_vwb, i0_wb2, i0_wb3, i1_by1, i1_by2, i1_vwb, i1_wb2,  
  i1_wb3, load_ex2, load_ex3, VRF } do  
  
    assert "--!!count -t nclk -cn bypass_mux_monitors_p3 -vn i1_Bmux_%{bypass_source};"  
  
    cover {[*]; i1_vq10_vsu_instr_v; i1_vq11_vmx_b_mux_cntl(0..5) = %{bypass_source}; true[*2];  
    i1_rf1_cancel=0};  
  
  %end  
  
}
```

Converting to VHDL with FoCs

- Neglecting the for loop, FoCs shows one of the monitors as:

Assertion 1 of vunit vector_bypass_i1bmux:

```
--!!count -t nclk -cn bypass_mux_monitors_p3 -vn i1_Bmux_1;
```

Never,

For zero or more cycles (1)

and then

For one cycle (i1_vq10_vsu_instr_v)

and then

For one cycle ((i1_vq11_vmx_b_mux_cntl(0..5)) = (1))

and then

For 2 cycles (1)

and then

For one cycle ((i1_rf1_cancel) = (0))

- The FoCs gui provides a very useful debug aid by simply converting your PSL into english
- We required little if any further debugging for our monitors

Converting to VHDL with FoCs

- FoCs converted the previous example to about 78 lines of VHDL, including the process statements, variables, and etc.
- FoCs converted the same example into 58 lines of verilog
- Handwritten verilog for this example would start at about 34 lines of verilog depending user choices

So addressing the issue of code bloat, while this example would indicate that there is not a significant code bloat issue with FoCs, especially given the work savings for development and maintenance of the code. However, we have seen several cases in more complex state machines where the FoCs solution is less optimal.

So yes generated code is somewhat more verbose than handwritten code. How does that affect simulation speed? It will clearly reduce it. But it can be addressed, and we claim that the given the current state of the tool, that the value to verification far out weighs the loss of simulation speed.

Test Planning

- We found that the PSL/Sugar code could be added directly to the test plan
- Given very simple supporting documentation and sensible naming conventions, the code can truly document itself.
- We found that one did not have to have much PSL expertise to review the code for completeness and accuracy.

```
%for bypass_source in { i0_by1, i0_by2, i0_vwb, i0_wb2, i0_wb3, i1_by1, i1_by2, i1_vwb, i1_wb2, i1_wb3, load_ex2, load_ex3, VRF } do
```

```
cover {[*]; i1_vq10_vsu_instr_v; i1_vq11_vmx_b_mux_cntl(0..5) = %{bypass_source}; true[*2]; i1_rf1_cancel=0};
```

Problems

- Drawing the line how much coverage is enough?
- Over constraining random constraint files
- Completion, when are we done?
 - Is 100% coverage done?
 - It is all about risk management
- PSL is simple to learn to be productive in 1 day, but it does require some expertise. For instance: If Rst is asserted it must remain active for a minimum of 8 cycles and may remain active for upto and including 25 cycles.

assert always {[*];Rst[*8 to 25];not reset} is incorrect the proper solution:

assert always {rose(Rst)} |-> {Rst[*8 to 25]; not Rst};

Results and Conclusion

- Closing the coverage loop improved our verification performance and quality
- Given that our functional monitors were conceived of, and implemented by both the verification team and the design team, we claim that we had a well thought out and very evenly applied coverage methodology
- The combination of these strategies allowed us to complete our portion of the design ahead of schedule.
 - We then reduced our simulation throughput, allowing units that still needed simulation cycles to use “our” cycles
- We then spent the extra time reviewing our coverage plan, RTX and design.

We had no post RIT bugs in any of the logic that was covered by this plan.

We did however, have post RIT bugs in sections of the design that relied on the more traditional system of custom environment hooks and random scenario generation to find bugs.