Faster, Better, Cheaper Verification Using Internal DUT Stimulus

Elihai Maicas

Intel Corporation
Haifa, Israel

elihai.maicas@intel.com

ABSTRACT

An important phase of the ASIC verification flow is verification at the unit level. The main goal of this phase is to isolate the unit and thoroughly verify its correctness prior to the integration into the subsystem. However, there are cases where creating a fully isolated testbench for the unit cannot be justified. For example, when creating a proper BFM to model the unit's stimuli requires too great an effort or where the unit contains a new and relatively minor feature to be added to an existing legacy subsystem. This paper will demonstrate a novel way of using the UVM and bind techniques to handle such cases, by manipulating the internal DUT signals relating to the feature and thus achieving a unit-level verification while running in a subsystem level environment.

Table of Contents

Table of Figures

1. Introduction

As the number of target chips increases at ASIC companies, reuse becomes ever more important. A single IP might have a number of targets each requiring a small feature to be added and which will have to be thoroughly verified. We have a solution that validates these new features faster and cheaper than current methods using a combination of interface binds and Universal Verification Methodology (UVM) techniques.

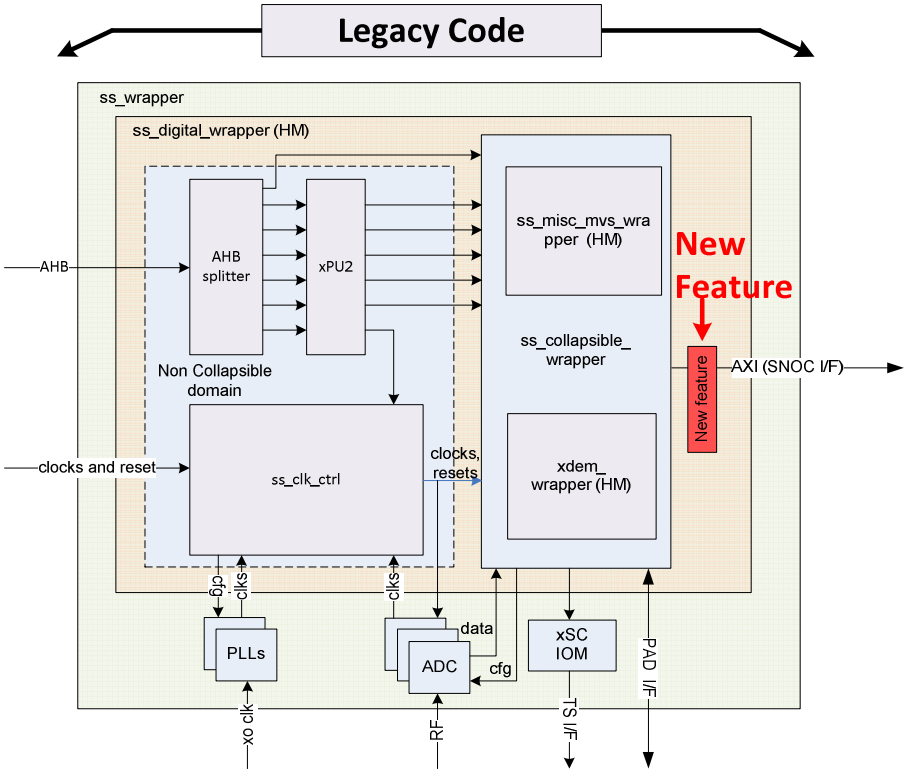

Figure 1: Legacy subsystem with new feature

2. Background

Recently, our previously verified subsystem IP was given a new target. This target required a small new feature to be added. This feature gave the IP the ability to snoop on an AXI bus going to the NOC. Upper address bits were encoded as specific flags that indicated if the transaction was an event and to which of the three interrupters to notify. This notification was done by inserting an additional AXI doorbell write. Additionally, an arbiter was added to arbitrate between the requests for the different interrupters.

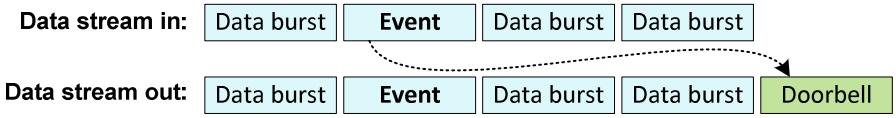

Figure 2: New feature flow

3. The Problems

Initial testing of the feature was completed at the subsystem level. Immediately two problems presented themselves.

Corner Cases

The first problem as usual appeared as we started trying to cover corner cases. Since we had a new arbiter we wanted to generate more than one active request at the same clock cycle. Additionally, although possible, the subsystem very rarely generated multiple event writes using an AXI burst transaction which was an interesting corner case as well.

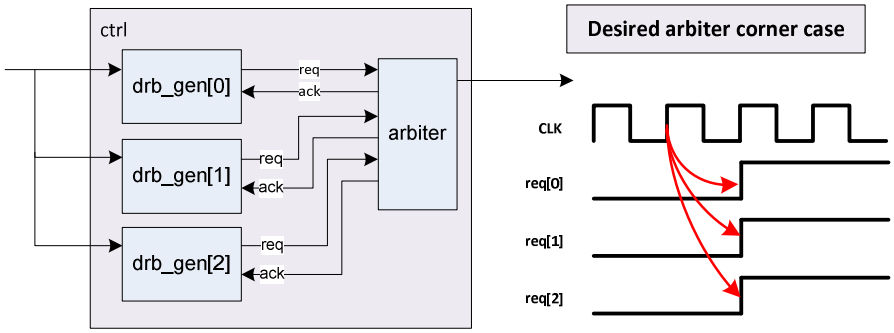

Figure 3: Desired arbiter corner case

Our first attempt at covering these cases was to use the large regression technique i.e. run enough tests and hope that it will eventually reach it. However, most of the time this "technique" doesn't succeed and so we were left with the choice of either creating a unit level testbench or writing a directed test. A unit level testbench was not optimal since it required a new Bus Functional Model (BFM), in addition to the rest of the supporting infrastructure such as a new environment, clock generators etc. Furthermore, a directed test was no guarantee of success as there needed to be too many variables to line up exactly and the number of transactions needed to reach the corner case was prohibitive.

Error Injection

The second problem was validating that the added checkers were working properly. As per a standard verification checklist it is required that errors are injected to ensure that checkers will fail and thus prevent false positives.

For the negative testing usually the two main approaches are to use an external tool to inject errors or to use a rather cumbersome set of force-release blocks from the testbench. However, external tools usually have a high ramp-up cost. As for using Verilog's force-release syntax, this technique affects simulator performance, and in general follows neither the spirit nor elegance one expects from a well-crafted UVM testbench.

4. The Solution

We determined that the best way to solve the two problems outlined above without having to resort to new testbenches, complicated directed tests, or outside tools was to give the user the ability to access, monitor, and modify internal Device Under Test (DUT) signals during runtime. In order to perform the above actions we gathered the signals into a system-verilog interface and bound it to a set of internal signals inside the DUT. The interface was then registered in the main UVM database, and was then fetched by a designated environment agent.

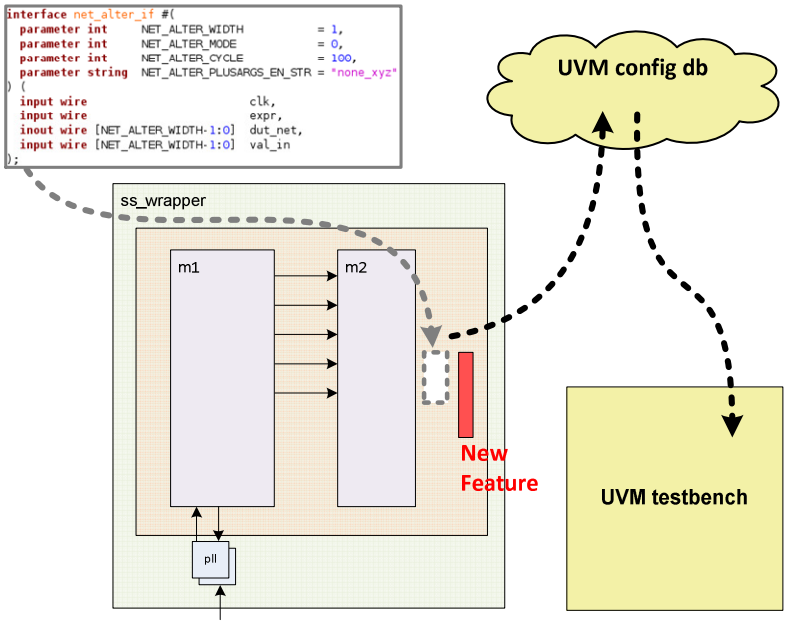

Figure 4: Bind interface and registration to the db

There are two methods for registering the bound interface to the UVM configuration database. The first one is to wrap the interface within a module, bind the module, and have the module register the interface instance. The second method is to use the `::self` syntax (Synopsys only), analogous to `this` in OOP, and have the interface register itself. In both cases, the name id in the database will represent the actual hierarchical path of the interface. This can be retrieved by using `%m` syntax.

Set the interface using wrapper module

```
module net_alter_if_wrapper #(
  //parameters...
) (
  input wire      clk,
  // ports...
);

  // interface instance
  net_alter_if #(/*parameters...*/)) i_net_alter_if (
    .clk(clk),
    /*ports...*/);

  initial begin
      string loc_hier = $psprintf("%m.net_alter_if");
      uvm_config_db#(virtual net_alter_if#(/*...*/))::set(/*root*/,\
      "*", loc_hier, i_net_alter_if);
  end
```

Set the interface using ::self

```
interface net_alter_if #(
  //parameters...
) (
  input wire      clk,
   // ports...
);

  initial begin
    loc_hier = $psprintf("%m");
    uvm_config_db#(virtual net_alter_if#(/*...*/))::set(/*root*/,\
    "*",loc_hier, interface::self());
  end
```

Figure 5: Methods to set the interface to the UVM config db

Monitoring was straightforward but modification requires a deeper understanding of the Verilog wire assignment rules. In general, a Verilog net can have multiple drivers, but they should not be driving at the same time. If one driver drives '1' and other driver drives '0', the net will be x. Thus during the monitoring when we identified the correct moment for intervention we assigned a net from the bound interface with a higher strength logic value than the internal net.

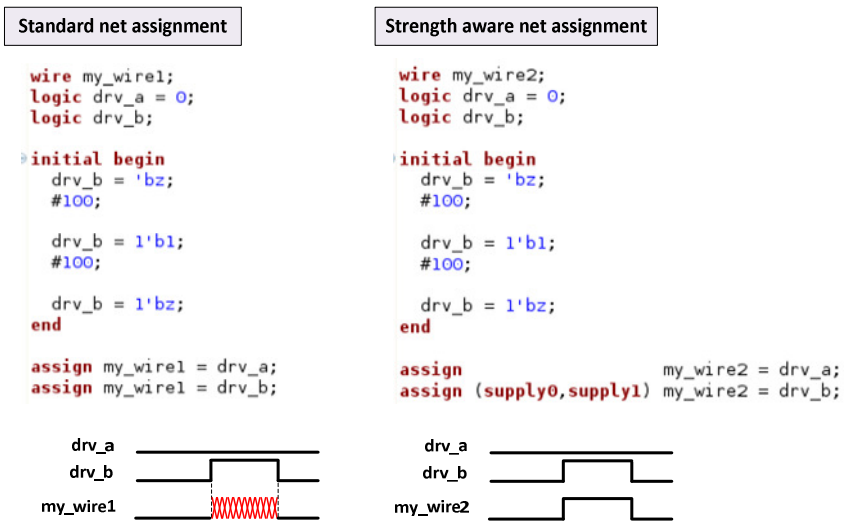

Figure 6: Regular net assignment vs. strength aware assignment

Now we will explain how this was used in practice.

For the first problem described above, the requirement to cover multiple simultaneous requests to the arbiter, we bound the interface to the request signals. From the environment agent we fetched the handle to the interface that was previously registered in the UVM configuration database. We then monitored the signals and waited for the first request. Once the internal DUT net became active it was immediately overridden by our bound interface to an inactive state (this was acceptable since the module driving the request is unaware of the change and still "believes" it is driving the request with an active value). Then the interface waited for a second active request and at that moment we reactivated the first request thus simulating a scenario of two requests being active at the same clock cycle.

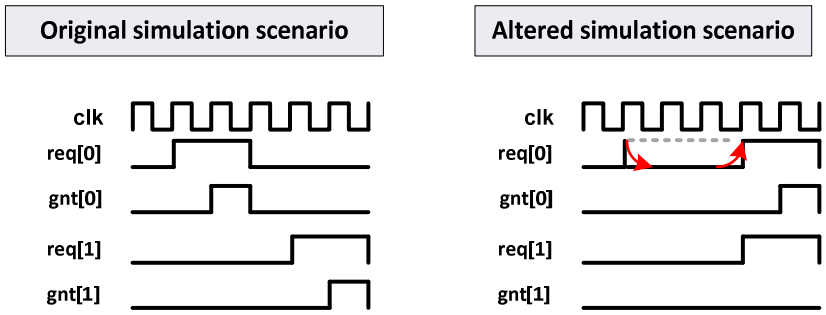

Figure 7: Reaching arbiter corner case with the bound interface

For the second problem of simulating the very rare scenario of multiple event writes using an AXI burst transaction we bound our interface to the AXI lines. We then waited until there was one of the very frequent data burst transactions, and through our interface dynamically reconfigured the upper bits to convert it to an event transaction. This allowed us to test the feature quickly and thoroughly.

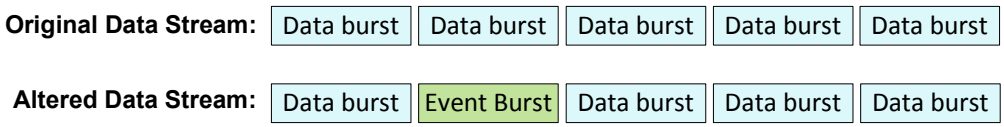

Figure 8: Modifying burst type with the bound interface

Finally we added a feature into the interface that we call "autonomous signal injection" which allows signal injection to occur periodically without user intervention. The interface was bound to the AXI master and the data out was periodically corrupted when a doorbell transaction was identified. Therefore, the test failed as expected since it no longer matched the expected data.

Autonomous Signal Injection

```
// autonomously alter net
initial begin

  // initially, do not drive
  loc_net = {NET_ALTER_WIDTH{1'bz}};

  if($test$plusargs(NET_ALTER_PLUSARGS_EN_STR)) begin

    forever begin

      repeat (NET_ALTER_CYCLE) @(posedge expr);
      #1;

      $display("[%0t] altering dut net at %m", $time);
      case (NET_ALTER_MODE)
        0:  loc_net = ~dut_net;
        1:  loc_net = {NET_ALTER_WIDTH{1'bx}};
        2:  loc_net = val_in;
        3:  loc_net = $urandom();
      endcase

      @(negedge expr);
      loc_net = {NET_ALTER_WIDTH{1'bz}};

    end
  end
end
```

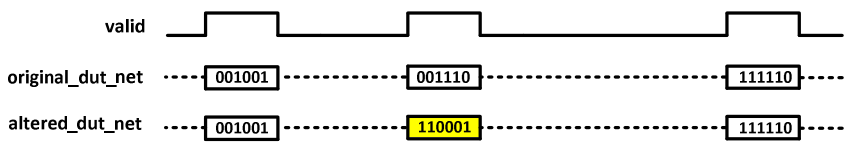

Figure 9: Autonomous error injection

This proved the scoreboard was functioning properly and fulfilled the requirement for negative checking.

5. Conclusions

We have demonstrated through bound interfaces and UVM techniques a simple, quick, and effective way of testing new features, by manipulating internal DUT signals. This technique enables us to cover hard to reach scenarios and also to easily inject error scenarios for negative testing. It allows common IPs that are to be integrated into multiple targets and requiring small feature changes to be verified quickly and exhaustively with a single proven testbench.