Advanced Scoreboard Techniques using UVM

François Cerisier
Test and Verification Solutions

DVClub – 9th Sept 2013
**Abstract**

This presentation describes scoreboard techniques using UVM. It reviews the scoreboard principles and UVM features for scoreboard and extends to more advanced techniques to verify full transaction contents, data, attributes and responses for data flow designs, bridges and interconnects.

The presentation will go through the analysis_port requirements, search in lists, queues and pools and using the UVM factory to architect a generic scoreboard for complex multi-protocol interconnects.

**Biography**

François Cerisier has an Engineering Diploma in Digital Signal Processing from Polytech’Sophia, University of Nice-Sophia-Antipolis and over 13 years of experience in verification of IPs, CPUs and System-On-Chips and in hardware/software co-verification. François gained verification methodology expertise from industrial projects of major semiconductor companies (including Infineon, Broadcom, ST-Microelectronics, ST-Ericsson, NXP) and EDA start-ups. He is now leading Test and Verification Solutions subsidiary in France to provide verification services and consulting.
Agenda

- Introduction
- Scoreboard Principles
  - Case 1: Bridge / Data flow design
  - Case 2: More complex data flow design
  - Case 3: Memory controllers
- Connecting Scoreboard to Agent
  - analysis_port
- Storing and searching for referenced data
  - Pool
  - Queues
- Scoreboards for complex designs
  - Requirements for predictor
  - Divide and Conquer
  - Using the UVM Factory
- Interconnect Scoreboard Architecture Example
- Conclusion
Scoreboard Tutorials

• **UVM User Guide**
  – Quick explanation how to connect a scoreboard

• **UVM Cookbook, Verification Academy**
  – Straight to the code of a out of order comparator/predictor

• **Books, Online Materials, UVM Trainings**
  – A lot about UVM
Random Verification Aspects

• Verification using Constrained Random Generation:
  – Generation of random test scenarios
    • Automate tests
  – Functional Coverage
    • Know what has been automatically covered
  – Checks
    • Know that the design complies to a protocol
      → assertions
    • Know that the design does what it should do
      → transaction checks
      → scoreboards
Scoreboard Roles

- **Check the design is doing what we expect**
  - transaction content
    - Data, address, attributes, opcode, response code
  - transaction ordering
    - FIFO
    - OOO
    - Precedence relationship
Data flow design example

DUT
Typical UVM testbench

- Test
  - sequences
  - sequencer
  - driver
  - monitor
  - assertions

- Bus A VIP
  - Master Agent

- Bus B VIP
  - Slave Agent

- DUT
  - vif
  - driver
  - monitor
  - assertions
Scoreboard principles – data flow design

Test

sequences

sequencer

driver

monitor

assertions

Bus A VIP
Master Agent

DUT

sequences

Bus B VIP
Slave Agent

driver

monitor

assertions

trans

Ref trans

Match ?

Transaction Predictor

Storage

Compare / Search

Scoreboard

Response Scoreboard
Scoreboard principles – complex data flow design

Test

sequences

sequencer

Bus A VIP
Master Agent

driver

monitor

assertions

trans

Bus B VIP
Slave Agent

sequences

DUT

vif

vif

DUT

driver

monitor

assertions

trans

Ref

trans

TLM Reference Model

Match ?

Storage

Compare / Search

Request Scoreboard

Response Scoreboard

DVClub Sept 9th 2013 - Advanced Scoreboard Techniques using UVM – François Cerisier – page 10
Scoreboard principles – memory controller

Test

sequences

sequencer

driver

monitor

assertions

Bus A VIP Master Agent

DUT

Scoreboard

Write trans

Transaction Predictor

Ref trans

Storage

Match ?

Compare / Search

Read trans

Reads are checked against previous Writes
Connecting the scoreboard – Analysis Ports

- **UVM 1.1 Class Reference Manual**

### 13.5 Analysis Ports

This section defines the port, export, and imp classes used for transaction analysis.

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis Ports</strong></td>
</tr>
<tr>
<td>This section defines the port, export, and imp classes used for transaction analysis.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>uvm_analysis_port</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcasts a value to all subscribers implementing a uvm_analysis_imp.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>uvm_analysis_imp</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Receives all transactions broadcasted by a uvm_analysis_port.</td>
</tr>
</tbody>
</table>
Connecting the scoreboard – UVM Example

```verilog
// MyBus Monitor: monitor and collect transaction.
class mybus_monitor extends uvm_monitor;
    uvm_analysis_port #(mybus_transfer) item_collected_port;

    task monitor_transaction;
        mytrans.address = vif.addr;
        /* ... */
        item_collected_port.write(mytrans);
    endtask
endclass

// My DUT Scoreboard:
// port_in: connects to master monitor (DUT slave interface)
// port_out: connects to slave monitor (DUT master interface)
`uvm_analysis_imp_decl(_in)
`uvm_analysis_imp_decl(_out)
class mydut_scoreboard extends uvm_scoreboard;
    uvm_analysis_imp_in #(mybus_transfer, mydut_scoreboard) port_in;
    uvm_analysis_imp_out #(mybus_transfer, mydut_scoreboard) port_out;
    /* ... */
    extern function void write_in(mybus_transfer trans);
    extern function void write_out(mybus_transfer trans);
endclass

// My main DUT verification environment
class mydut_env;
    mybus_env mybus_env;
    mydut_scoreboard mybus_scbd;
    /*...*/
    virtual function void connect_phase(uvm_phase phase);
        mybus_env.agents[0].monitor.item_collected_port.connect (my_scoreboard.port_in);
        mybus_env.agents[1].monitor.item_collected_port.connect (my_scoreboard.port_out);
    endfunction
endclass
```
Storing Data / Transactions

• **Storage structure strongly depends on ordering model**
  – FIFO → queues
  – OOO → dynamic arrays, queues
  – Memory → associative array (indexed by address)
  – Others:
    • Associative array of queues

• **Data type:**
  – Memory → scalar, bytes, integers,
  – Data path:
    • Scalar
    • Transaction Item Class
Storage of transactions

class mytrans extends uvm_transaction;
    bit [31:0] address;
    bit [31:0] data;
    /* .... do_compare() , do_copy() , uvm_object_utils */
endclass

class mytrans_queue extends uvm_object;
    mytrans queue[$];
endclass

class myscoreboard extends uvm_scoreboard;
    mytrans_queue recorded_trans[bit [31:0]];

    // Queue up transactions in per address queues
    function void write_in(mytrans tr);
        mytrans stored_tr = new;
        stored_tr.copy(tr);
        recorded_trans[stored_tr.address] = new;
        recorded_trans[stored_tr.address].queue.push_back(stored_tr);
    endfunction
// Search for a matching transaction.
// Stop at the first matching transaction at a specified address
function void write_out(mytrans tr);
    bit found = 0;
    int ii = 0;
    if ( recorded_trans.exists(tr.address) && recorded_trans[tr.address].queue.size() )
        begin
            // search for the first matching transaction.
            do begin
                found = recorded_trans[tr.address].queue[ii].compare(tr);
                ii = ii + 1;
            end while (!found && ii < recorded_trans[tr.address].queue.size() );

            // remove matching transactions from the queue
            if ( found )
                begin
                    recorded_trans[tr.address].queue.delete(ii-1);
                    if ( recorded_trans[tr.address].queue.size() == 0 )
                        recorded_trans.delete(tr.address);
                end
            else
                `uvm_error(get_type_name(), $psprintf("Unable to find matching transaction at
end

else
    `uvm_error(get_type_name(), $psprintf("Unable to find any transaction at address ( endfunction
Predictor / Reference Model

• **Predictor required when:**
  – Compared transaction has not the same format as the input
    
    *(protocol bridges)*
  – Design is transforming data
    
    *(encryption, filter, encoder, …)*

• **Re-Use concern:**
  – Encapsulation: implemented as a separated class
  – uvm_analysis_port / imp to connect
  – UVM Factory to extend, replace existing objects
Divide and Conquer

• What if the design has
  – Different transaction paths/routes
  – Different behaviour depending on
    • Address segments
    • Opcodes or other transaction attributes

• Option 1: TLM Reference Model
• Option 2: Divide and Conquer
  – Replace complex predictor with several simpler scoreboards
Divide and Conquer – AXI read/write separation

```verilog
class axi_bridge_scoreboard extends uvm_scoreboard;
  axi_scoreboard write_channel_scbd;
  axi_scoreboard read_channel_scbd;

  // AXI channel separation
  virtual function void write_in(mytrans tr);
    if ( tr.direction == WRITE )
      write_channel_scbd.write_in(tr);
    else
      read_channel_scbd.write_in(tr);
  endfunction
```
Improving reuse

- **UVM Factory**
  - Same architecture
  - Different implementation, different behavior

```c
factory.set_inst_override_by_type(
    ooo_scoreboard::get_type(),
    fifo_scoreboard::get_type(),
    "dut_env0.scbd0");

dut_env0 = dut_env::type_id::create("dut_env0",null);
```
Complex NoC / Fabrics scoreboard example
Conclusion

• **Scoreboards verify transaction functional correctness**

• **SystemVerilog provides aggregate types for transaction storage & search**
  – Dynamic Arrays, Associative Arrays, Queues, Classes

• **UVM ease scoreboard development, providing:**
  – UVM analysis ports (easier than call backs)
  – uvm_transaction compare() / do_compare() methods
  – UVM factory for extension, replacement & reuse.

• **Divide and Conquer:**
  – keep simple things simple.
  – Compose simple blocks to build complex behavior
Thank you

• Questions?