A pragmatic approach leveraging portable stimulus from subsystem to SoC level and SoC emulation

Karandeep Singh, (karandeep.singh@nxp.com)¹
Aditya Chopra, (aditya.chopra@nxp.com)¹
Joachim Geishauser, (joachim.geishauser@nxp.com)²
Nitin Verma (nitin.verma@nxp.com)¹
NXP Semiconductors, ¹Noida, India, ²Munich, Germany
Introduction

• SoC complexity
  – Verification Challenge

• Verification Stages
  – IP/Subsystem
  – SoC

• Key of testbench architecture
  – Seamless usability for native UVM Verification (sequences based approach)
  – C based test verification at subsystem
  – Stimulus portability to SoC simulation and SoC emulation
Execution at Subsystem Level
Subsystem Testbench architecture

- DPI calls from C side
- TLM 2.0 GP based packets and execution layering (SV)
- UVM TLM GP driver
- Cross bar
- Subsystem (RTL/GLS)
- Protocol Specific VIP's
- ISR Agent
- DMA Agent
- System RAM

## C Code
```
WR(REG2, 0xABCDABCD);
RD(REG2, data_value);
```

## C Code
```
dma_configure(S_addr, num_of_bytes,D_addr);
```

## DMA agent implemented through same DPI layer, which further triggers events for DMA sequence.
Stimulus execution at Subsystem Level

<test>.c
Uses standard C-API functions for bus transactions
RD32 (for word read)
WD16 (for half word writes)
Call_blk_A()
Call_blk_B()

C-BFM
- C domain (test.c)
  - Communication done through DPI calls
  - System Verilog domain (UVM lb)
    - Communication done using TLM2 GP (Generic Payload)
    - UVM VIP (converts GP to AHB)

Light weight core
- C-BFM can be replaced with a light weight core if it is desired to port the same testbench to Emulator and run the same testcase

Cross bar
- Testbench
  - RTL
  - Bus
  - sub-system A
  - Sub-system (DUT)
  - sub-system A
Stimulus Execution at SoC Level

```c
<test>.c

Uses standard C-API functions for bus transactions
RD32 (for word read)
WD16 (for half word writes)
Call_blk_A()
Call_blk_B()
```
Major Requirements

- C and SV communication
- Test execution flow
- Reset Handling
- Interrupt execution
- Reusable-Interoperable-Protocol Independent test sequence
C and SV communication through DPI calls.

SV WORLD

- task run_phase(uvm_phase phase);
- fork begin
  - run_startup;
  - run_test;
  - run_shutdown();
- end

- Call to C test
  - run_startup
  - run_test
  - run_shutdown

C WORLD

- STARTUP_STIM();
- TESTCASE();
- SHUTDOWN_STIM();

DPI Layer

- import “DPI-c” context task run_startup();
- import “DPI-c” context task run_test();
- import “DPI-c” context task run_shutdown();

1. Clock Configuration
2. SW reset to RTL

1. Status registers match
2. No error check
3. No pending interrupt
4. No faults

1. No pending transactions
2. No protocol violations
extern WR(unsigned int addr, usinged int data)

int testcase(){
    W8(REG_NAME,0xABCDABCD);
    task WR8;
    sequence.W8(address,data);
    sequencer.start (GP)
    GP to data in case of read
    endtask;
    RD(REG_NAME,rd_value);
}

import "DPI-C" task testcase;
export "DPI-C" task WR;

task sequence::body()
    fork
        testcase();
        execute_seq();
        join
    endtask

task sequence:: execute_seq();
    while(1)begin
        task WR8;
        sequence.W8(address,data);
        sequencer.start (GP)
        wait_for_nb_transport_bw_call;
    endtask;
Reset Handling

**Simulation**
- C Test trigger interrupt
- DPI CALL
- Reset Env
  - mon
- Extended Reset ENV
  - Functional reset
- C-BFM agent (UVM + C calls)

**Emulation**
- TCL Driver
  - resets

**Analysis port** (event)
- POR
- Functional reset

**Synthesizable Reset DRIVER**
- Parametrized Reset assertion-desertion driver
- BFM which is working as CORE, on receiving reset event
  - Stop the current seq
  - Start test execution flow again

**DUT**
- POR
  - Functional reset

**Base layer implementation and connections**
- DUT Specific reset overrides, SoC specific reset overrides
Interrupt execution diagram

Subsystem  ---  ISR agent

1. Lock the execution current layer and add UVM objection for this layer
2. Move to next layer

<table>
<thead>
<tr>
<th>Subsystem Execution Semaphore: Application layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 – base code execution</td>
</tr>
<tr>
<td>L2 – Intr level1</td>
</tr>
<tr>
<td>L3 – Intr level2</td>
</tr>
<tr>
<td>L255</td>
</tr>
</tbody>
</table>

Whenever there is interrupt:
1. Grab the handle for sequencer
2. Execute service routine
3. Move to L-1 layer
4. Drop UVM objection raised for this interrupt.

## C world
`INSTALL_ISR(INT_NUMBER, intr_routine)`

```c
void intr_routine{
  intr++
  CLR(INTR_REG,INTR_FILED);
}
```

## SV world for interrupt execution
- Lock the current execution layer
- Add execution layer semaphore for +1
- Execute the interrupt routine

TLM GP based C-agent(DPI calls based from C)

Grab the sequencer and fire intr_routine
Direct re-use of testbench and testcases in emulator

- Frontend/RTL Changes for emulation
- Testbench Changes for emulation
- Execution Flow
Application

• Stimulus creation that can be re-used in SoC level testbench.
• L-BIST and M-BIST verification.
• Gate level verification.
• Special purpose RTL simulations (Ex: To generate power analysis VCD).
• Direct re-use of testbench and testcases in emulator.
• DFT verification.
Summary

• Seamless reusability for native UVM verification.
• Faster RTL compilation/simulations
• Faster UVM-SV based driver leads early verification closure.
• Synthesizable Verilog based slave memories eased portability to SoC emulation.
• Interoperable and protocol independent implementation using TLM 2.0 GP compliance.
• SoC Independent tests development (i.e. even when the sub-system has not been integrated into the SoC).
• TLM 2.0 compliance testbench can easily plugged with SYSTEMC hosts for early low-level driver development.
• GLS faster run time and can be run independent of the SoC.
Questions

Finalize slide set with questions slide