Vertical Reuse of functional verification from subsystem to SoC level (with seamless SoC emulation)

Pranav Kumar, Staff Engineer
Digvijaya Pratap SINGH, Sr. Staff Engineer
STMicroelectronics, Greater NOIDA, INDIA
Ankur Jain, Verification Technologist
Mentor Graphics, Bangalore, INDIA
Introduction

• Ever increasing SoC complexity
  – Verification challenge

• Two levels of verification
  – IP/Subsystem Level
  – SoC Level (integration aspects)

• Full application scenarios
  – FPGA prototyping
  – Emulation

• Verification methodology standardization
  – UVM
Typical verification Environments

Subsystem Level

SoC Level

© Accellera Systems Initiative 3
Standardizing subsystem level verification Env
LVP – Lightweight Virtual Platform

• Processor subsystem in SystemC/TLM
  – TLM ARM ISS
  – Abstracted TLM interconnect
  – C++ Memory models integrated in SystemC
  – VIC and DMAC TLM models
  – Bus interface bridges
    ▪ AMBA and STBus
Emulation Env

- External interface now driven by real hardware or external device models
  - Addressing the simulation speed bottleneck
  - Useful for concurrent/full application verification scenarios
  - Early software development, e.g. boot rom code, etc.
Reuse – simulation to emulation
Reuse – major challenges

• Easy portability of VIP
  – Consider VIP as another IP with
    ▪ Custom registers set
    ▪ VIP driver in ‘C’
    ▪ Standard bus interface

• C & SV/UVM Synchronization
  – VIP env in SV/UVM
  – VIP driver in ‘C’
  
  Develop custom VIPs
  OR
  Interface VIPs with intermediary layer mentioned here as VAL (Verification Abstraction Layer)
VAL – Central Idea

• Control on complete verification environment
  – VIP reset and config
  – IP config
  – VIP and IP activities synchronization

• Allow SW-driven constrained test randomization
  – Configuration parameters
  – Data

• Control score-boarding
  – Enable / disable
  – Reference data and configuration values transfer

• Initiate backdoor operations
  – Accesses to LVP and VIP memories
VAL - Implementation

• A generic configurable component with two sub-layers:
  a) VAL_FE (VAL FrontEnd)
     ▪ Interface to
       — any standard (AMBA, STBus, etc.) bus, or
       — proprietary bus or
       — TLM model
  b) The VAL_BE (VAL BackEnd)
     ▪ Interface between VAL_FE and VIP
     ▪ converts instructions from VAL_FE to
       — configuration or data packets and
       — sequence generation or control of existing sequences
VAL – In verification environment
Val Flow - Steps

- Understanding VAL flow
  1. VAL trigger from C
  2. Cfg randomization (constrained, ....)
  3. Cfg updated in Memory model
  4. C test reads Cfg information from Memory
  5. C test programs Subsystem/IP in according configurations
Testbench Architecture

- **LVP (Lightweight Virtual Platform)**
  - A processor subsystem (SystemC/TLM) with multiple AMBA interfaces to control the DUT & VIP.
- **VAL (Verification Abstraction Layer)**
  - C to SV/UVM interface
  - Provides control, triggering and synchronization of UVM VIP - configuration, randomization and sequences from a C testcase running on the LVP.
  - Provides control of other testbench components e.g. scoreboard.
  - TLM ports to trigger and synchronize
- **VIP**
  - Connected to LVP through VAL via AMBA interface
  - Where AMBA i/f can be replaced with any other bus i/f
  - Providing APIs for controlling QVIP configuration and sequences
  - Triggering above QVIP APIs from C, and synchronizing using TLM
Example Testbench

Lightweight Virtual Platform (LVP)

Testframe.sv

© Accellera Systems Initiative
VIP as an IP

• Identify configuration parameters of the VIP

• Define C data-structure for them
  – custom registers set

• Develop VIP driver using above data structures
  – configuration/control functions
  – read/write transactions functions
  – Interrupt handlers
typedef union {
    struct {
        uint32_t tlp_type : 8; // [7:0]
        uint32_t reserved : 7; // [14:8]
        uint32_t bar_size : 1; // [15]
        uint32_t unused_0 : 16; // [31:16]
    } Bit1;
    uint32_t Reg1;
} t_PCIE_QVIP_TLP_ID_REG;

typedef union {
    struct {
        uint32_t seq_id : 32; // [31:0]
    } Bit1;
    uint32_t Reg1;
} t_PCIE_QVIP_SEQ_ID_REG;

typedef struct {
    #ifdef BLOCK_LEVEL
        uint32_t* VAL_TRIG;  // @0
    #else
        uint32_t VAL_TRIG;   // @0
    #endif
    t_PCIE_QVIP_SEQ_ID_REG PCIe_QVIP_SEQ_ID_REG;        // @4
    t_PCIE_QVIP_SEQ_ADDRL_REG PCIe_QVIP_SEQ_ADDRL_REG;  // @8
    t_PCIE_QVIP_SEQ_ADDRH_REG PCIe_QVIP_SEQ_ADDRH_REG;  // @12
    t_PCIE_QVIP_SEQ_DATA_LEN_REG PCIe_QVIP_SEQ_DATA_LEN_REG;  // @16
    t_PCIE_QVIP_SEQ_DATA_REF_REG PCIe_QVIP_SEQ_DATA_REF_REG;  // @20
    t_PCIE_QVIP_PLP_REG PCIe_QVIP_PLP_REG;           // @24
    t_PCIE_QVIP_END_SEQ_REG PCIe_QVIP_END_SEQ_REG;       // @28
    t_PCIE_QVIP_SYNC_FLAG PCIe_QVIP_SYNC_FLAG;        // @32
    t_PCIE_QVIP_MSI_OFF PCIe_QVIP_MSI_OFF;           // @36
    t_PCIE_QVIP_DLLP_REG PCIe_QVIP_DLLP_REG;          // @40
} PCIe_QVIP_registers;

The Verification component configuration has been arranged in register set and made word addressable ‘C’ declaration for registers.

The register set for Verification IP are mapped to consecutive addresses in memory, reserving 0th for VAL triggers.
C & SV/UVM Synchronization

• Synchronization needed so that VIP and IP are configured before test is initiated (SV env)

• VIP’s sequencer should wait and not flow-in
  – Blocking port used to block sequencer

• VIP configuration based on SV/UVM config classes
  – VAL provides trigger for the configuration

• Sequences to be run on the VIP sequencer also controlled from VAL
Example - Synchronization

typedef struct {
  uint32_t* VAL_TRIG; // @0
  t_PCIE_QVIP_SEQ_ID_REG PCIe_QVIP_SEQ_ID_REG; // @4
  t_PCIE_QVIP_SEQ_ADDRL_REG PCIe_QVIP_SEQ_ADDRL_REG; // @8
  t_PCIE_QVIP_DLLP_REG PCIe_QVIP_DLLP_REG; // @40
} PCIe_QVIP_registers;

void PCIe_QVIP_init(PCie_QVIP_desc* p, uint32_t base_addr) {
  PCIe_QVIP_REGMAP = (PCie_QVIP_registers*) base_addr;
  #ifdef BLOCK_LEVEL
  PCIe_QVIP_REGMAP->VAL_TRIG = (uint32_t*) PCIE_QVIP_START;
  #endif
  PCIe_QVIP_START;
  #endif
}

class pcie_qvip_sequencer extends uvm_sequencer;
`uvm_sequencer_utils(pcie_qvip_sequencer)
uvm_blocking_get_port #(pcie_qvip_data) val_port;
....
vlc_sequencer pcie_vip_sqr;
function new (string name = "pcie__sequencer", ....);
  super.new(name,parent);
  val_port = new("val_port", this);
endfunction: new
endclass: pcie_qvip_sequencer

Sequence:
virtual task body();
  forever begin
    p_sequencer.val_port.get(d);
    ....
  endtask: body

Sequencer:

called from sequence to un-block

Trigger from “C” to unblock

“C” header for VIP [TRIG]

Blocking port to stop VIP
Example - Configuration and Data Flow Control via VAL
Configuration and Data Flow Control via VAL (cont ...)

• Two VAL_BE interfaces –
  a) QVIP_VAL_BE
     Two ports
     i. Config
        – configuration/control packets
     ii. Data
        – data transactions along with required control of sequence
  b) TEST_VAL_BE
     Two ports
     i. test_config
        – randomization of the test/config parameters
        – transfer to main memory and coverage-collector
     ii. test_data
        – providing reference and test data to scoreboard
S/W Framework layers for Reuse

- **Test impl. SS level To SoC level**
- **Test definition layer**
- **Low Level Drivers for subsystem, interrupt handlers, etc.**
- **HAL (Hardware Abstraction Layer) operating directly on IP Registers**

**SoC APIs For clock/reset**

**SoC APIs for IP/Subsystem level signals control and status**

**Call Test*()**

**Low Level Driver (API)**

**SS Header Files**

**Interrupt Handler (API)**

**IP-Xact Description of IP**

© Accellera Systems Initiative
Conclusion

• Testplan and coverage shared with SoC team
  – Reused for SoC verification plan.
• Subsystem LLD, Testcases successfully re-used from sub-system level to SoC level
  – Re-used further in emulation (thanks to VAL)
• Subsystem LLD re-used for silicon validation
Questions