UVM *e* Reference Flow User Guide

Version 1.1
November 2012

Printed in the United States of America.

Cadence Design Systems, Inc., 2655 Seely Avenue, San Jose, CA 95134, USA

Table of Contents

Overview

Universal Verification Methodology (UVM) ***e*** is a complete reuse methodology that codifies the best practices for development of reusable verification components (UVCs) targeted at verifying large SoCs. The sample verification environments that are included with this reference flow (both module level and subsystem-level) contain UVCs built based on *e*RM as well as using UVM ***e***. Both *e*RM and UVM ***e***-compatible UVCs can be nicely integrated together and can work seamlessly. Thus, the UVM ***e*** Reference Flow ensures that all exiting *e*RM-compliant environments need not be re-coded to work with a UVM ***e***-compatible environment.

The UVM ***e*** Reference Flow applies the Universal Verification Methodology (UVM ***e***) to the Block and Cluster Verification in a System on Chip (SoC) Design using the ***e*** Language. It begins by showing aspects of the verification of a Universal Asynchronous Receiver Transmitter (UART) block. This Reference Flow document then shows how to verify a cluster design (an APB subsystem) into which the UART gets integrated along with other design components (namely, SPI, GPIO, and so on).

The UVM Reference Flow design is based on an Ethernet switch System-on-Chip (SoC). The SoC has the following key design components:

- An Opencores Open RISC Processor
- Opencores Ethernet Media Access controller (MAC)
- AMBA AHB network interconnect
- Address look-up table (ALUT)
- Support and control functions. For instance, power management and peripherals like UART, SPI, GPO, timer, and so on.
- On-chip memories and memory controller

The UVM ***e*** Reference Flow also includes the following key verification components designed using the ***e*** language:

- AHB UVC
- APB UVC
- UART UVC
- GPIO UVC
- SPI UVC

For more information, please refer to the user documentation, available at these locations:

UVM ***e*** Reference : `<INCISIV Install Area>/kits/VerificationKit/doc/uvm_flow_topics/uvm_e/uvm_e_ref_flow_ug.pdf`

Release Version: 1.1

The UVM ***e*** Reference Flow release (UVM Reference Flow Version 1.1) is tested with Incisive Enterprise Simulator (IES). It should be possible to run the UVM ***e*** Reference Flow on any IEEE 1647 Compliant Simulator that supports UVM.

For more information about using the UVM Reference Flow, contact `uvm_ref@cadence.com`.

Setup and Installation Instructions

Licenses, Terms, and Conditions

Refer to the `README_terms_and_conditions.txt` file located at the installation directory.

Dependencies

For Cadence customers, IES 12.2 is required to run the UVM ***e*** flow.

Setup Instructions

1. Set up the UVM Reference Flow using one of the following methods:

   - In csh

     ```
     % setenv SOCV_KIT_HOME <INCISIV Installation Area>/kits/VerificationKit
     % source $SOCV_KIT_HOME/env.csh
     ```

   - In bash

     ```
     % SOCV_KIT_HOME=<INCISIV Installation Area>/kits/verificationKit
     % export SOCV_KIT_HOME
     % source $SOCV_KIT_HOME/env.sh
     ```

   Note: `uvm-1.1` is selected for this release of UVM ***e*** Reference Flow

2. Ensure that you have a simulation tool installed and properly set up.

Running a Simulation Using Incisive Enterprise Simulator

When the installation and setup of the UVM ***e*** Reference Flow is complete and the Incisive Enterprise Simulator (IES) is available, try a quick simulation to ensure everything is set up:

Module-level simulation

```
% $SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl/demo.csh
```

Cluster/subsystem-level simulation

```
% $SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/demo.csh
```

Further Information

Refer to the `README.txt` file in the installation area for information about:

- Available documentation for the UVM ***e*** Reference Flow and other related items
- Ethernet Switch SoC Design

- Opencores IP
- The Reference Design hierarchy and directory structure
- The Reference Flow verification environment and directory structure

UVM *e* Module-Level Verification

This section describes details of UVM ***e*** standalone environments contained in the UVM Reference Flow for verifying the functionality of both module and simple subsystem level environments using the *e* Language. The Reference Flow starts with a basic module, the UART, and shows how an environment can be built around the UART module by using the Universal Verification Methodology (UVM ***e***).

This section illustrates an implementation based on UVM ***e***.

UART DUT

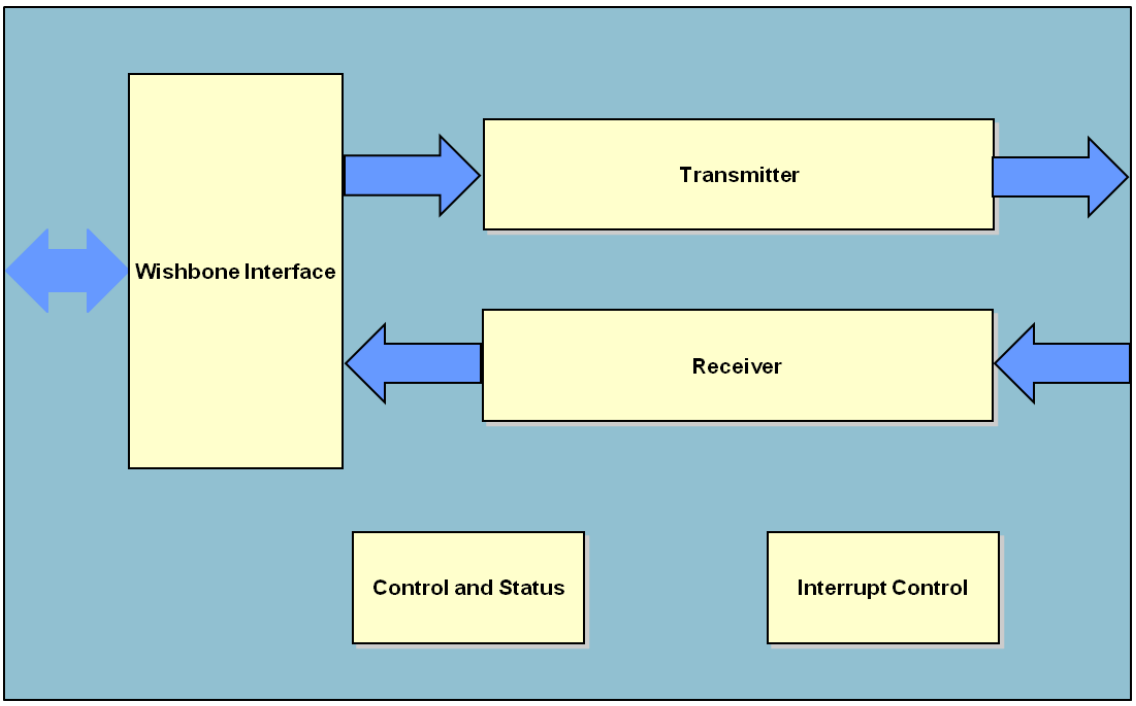
The UART module is a pre-verified soft IP from Opencores and is written in synthesizable Verilog RTL.

The DUT has the following interfaces, which needs to be driven by the UVC:

- WISHBONE interface
- UART receiver interface
- UART transmitter interface
- UART interrupt interface
- Clocks and Resets

The DUT is shown in the following Figure 1.

Figure 1. UART DUT



The following registers are available within UART DUT (WISHBONE Interface of the UART0):

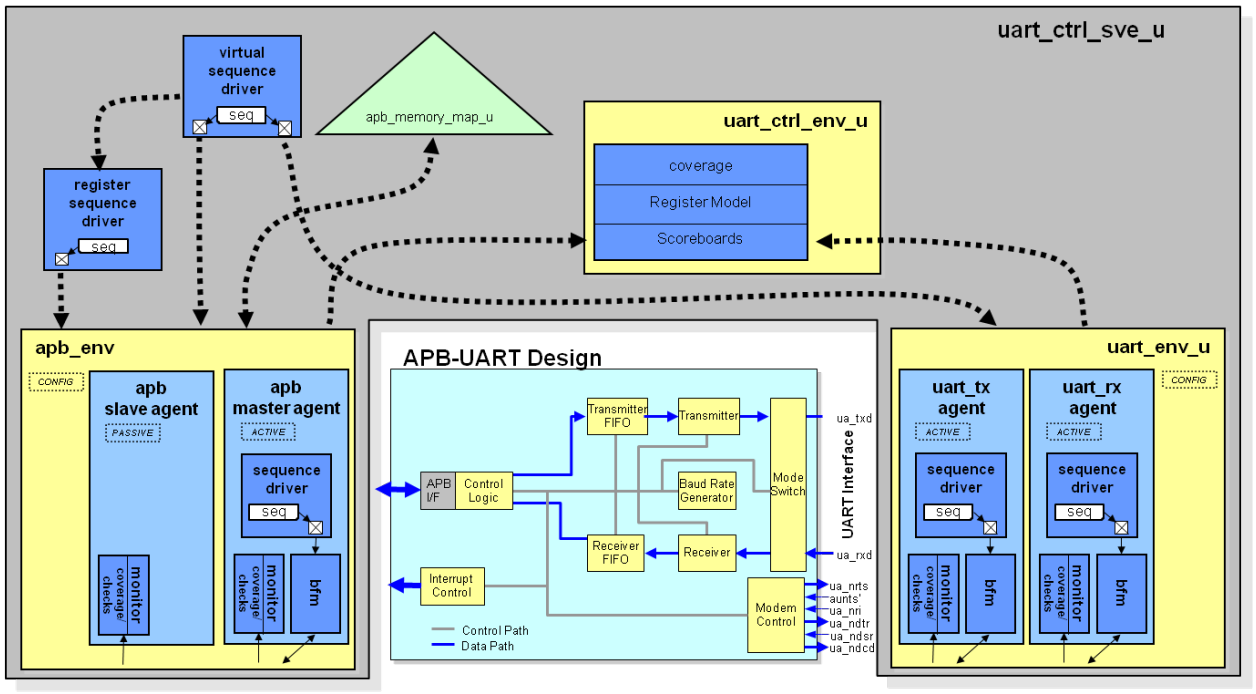
Table 1 WISHBONE Interface of the UART0

Offset	Function	Name	R/W	Reset Value
0x00	Receiver Buffer	UA_REC	R	0x00
0x00	Transmit Holding Register (THR)	UA_TRA	W	0x00
0x01	Interrupt Enable	UA_IER	R/W	0x00
0x02	Interrupt Identification	UA_IDR	R	0xC1
0x02	FIFO Control	UA_FCR	WO	0xC0
0x3	Line Control Register	UA_LCR	RW	0x03
0x4	Modem Control	UA_MCR	WO	0x00
0x5	Line Status	UA_LSTS	RO	-
0x6	Modem Status	UA_MSTS	RO	-
0x0	Divisor Latch Byte 1	UA_LDIV0	R/W	-
0x1	Divisor Latch Byte 2	UA_LDIV1	R/W	-

UART Module Verification – UVM *e* Reference Flow Methodology

The UART module environment is constructed as shown in the UART Module Verification Environment figure.

Figure 2 - UART Module Verification Environment



Universal Verification Components (UVC)

The UART module environment uses two UVCs: APB UVC (directly driving the WISHBONE interface) and a UART UVC constructed for the UVM Reference Flow according to the UVM ***e***. These UVCs are used to provide stimulus and monitoring functionality for the DUT interfaces. The simplest transactions are used in the APB interface. The UVCs are provided at the following locations:

APB: `$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/apb/e`

UART: `$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/uart/e`

Configuration Library

The top-level configuration file for the UART block ***e*** environment is available at the following location:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl/sve/e/uart_ctrl_sve.e
```

In this file, the top-level sys is extended to instantiate the UART Verification Environment (VE).

Test Sequence Library

The UART_CONFIG sequence library has been created to demonstrate how more complex sequences can be built up from existing sequences. In this example, the UART_CONFIG sequence performs the following tasks:

- Enables TX and RX operation of UART0
- Enables automatic flow control operation
- Sets the divider to 1
- Sets the baud rate divider to 15 (sets overall BDIV to 16)

The sequence library also ensures that the mode of operation randomly selected by the UART is also applied to the DUT.

The uart_config field of the UART_CONFIG'kind vr_ad_sequence generates some random values. Data length, number of stop bits, and parity values are captured from these randomly generated values, and passed as argument to the DUT register so that the values match between the UVC and the DUT. This is necessary; otherwise, the devices would not interoperate correctly.

```
extend UART_CONFIG'kind vr_ad_sequence {

   uart_config : uart_env_config;

   body() @driver.clock is only {

          var stop_bits := uart_config.stopbit_type;
          var parity    := uart_config.parity_type;
          var data_len  := uart_config.databit_type;

          parity_val,parity_sel,stopbit_val and datalen_val are derived from
          stop_bits,parity and data_len

          line_ctrl_val = pack(packing.high, 3'b0, parity_val , parity_sel,
          stopbit_val, datalen_val );

          Writing into the Line Control Register of the DUT
```

The randomly chosen values captured from the configuration are passed as an argument to the register write to the DUT.

```
      write_reg {.static_item == reg_file} line_ctrl_reg  value line_ctrl_val;};
```

The uart_config binding in the MAIN uart_ctrl sequence is as shown below:

```
extend MAIN uart_ctrl_sequence {
     !uart_config  : UART_CONFIG vr_ad_sequence;
     keep uart_config.driver == read_only(driver.vr_ad_seq_drv);
     keep uart_config.uart_config ==
     get_enclosing_unit(uart_ctrl_sve_u).uart_if.config;
};
```

Scoreboards

A scoreboard is used to verify end-to-end data transformation through the DUT. The scoreboard collects data from the UVC monitors at the interfaces of the DUT and compares the results against the expected transformation. This transformation can be very simple or can be a complex model, depending on the type of DUT.

The UVM scoreboard is a built-in scoreboard infrastructure implemented in ***e***. The UVM ***e*** scoreboard provides a default search and matching algorithm, and can be used to verify various kinds of systems with various kinds of requirements. UVM ***e*** scoreboard usage is very simple, and can be very easily integrated with *e*RM or UVM ***e*** or mixture of ***e***RM-UVM ***e*** compatible verification environment.

We recommend using the UVM ***e*** scoreboard, rather than implementing one from scratch.

The UART module-level environment uses the UVM ***e*** scoreboard. The user-defined UVM scoreboard unit inherits from UVM base class uvm_scoreboard.

The UVM scoreboard code snippet is as shown below:

```
unit uart_ctrl_scoreboard like uvm_scoreboard {
  // ports related to uart frame - to - apb transfer
      scbd_port uart_frame_add  : add uart_frame_s;
      scbd_port apb_trans_match : match apb_trans_s;

      // ports related to apb transfer - to - uart frame
      scbd_port apb_trans_add    : add apb_trans_s;
      scbd_port uart_frame_match : match uart_frame_s;
};
```

The scoreboard unit has add and match TLM analysis ports for communication with the VE.

Note: *port_name_predict*(item:port-declared-type) is a hook method to transform an added data item as required to ensure that it can be matched correctly.
port_name_reconstruct(item:*port-declared-type*) is a hook method to transform a match data item as required to ensure that it can be matched correctly.

add_to_scbd(item:any_struct) is a pre-defined UVM ***e*** scoreboard data transformation method, which will add an item into the scoreboard database for future matching. Similarly, match_in_scbd(item:any_struct) is used for matching an item with scoreboard database.

compute_key(keyed_struct:any_struct) is another pre-defined UVM scoreboard match process customization method , which can be used in an UVM e scoreboard, where some of the fields need to be added or omitted.

The UVM e scoreboard developed for the module-level environment is reused for subsystem level verification of UART. The UART control module level UVM ***e*** scoreboard code is available at:

```
uart_ctrl/e/checker/uart_ctrl_scoreboard.e
```

Coverage Module

The APB UVC contains a comprehensive coverage model. The UART, used as the starting point for the UART UVC, contains coverage items around the ports of the UART. The coverage code for the UART module level environment is located in uart_ctrl/e/cover/uart_ctrl_cover.e.

FIFO levels for TX FIFO, RX FIFO and Interrupt details are covered in the module. The example code is as given below.

```
tx_fifo_level_p: in simple_port of uint(bits:5) is instance;
   keep bind(tx_fifo_level_p, external);
   keep soft tx_fifo_level_p.hdl_path() == "regs.transmitter.tf_count";

   event cov_tx_fifo_level_e is change(tx_fifo_level_p$) @sim;
cover cov_tx_fifo_level_e using per_unit_instance is {
     item tx_fifo_level : uint(bits:7) = tx_fifo_level_p$ using
     ranges = {
       range([0],       "Empty");
       range([1..31],   "1 to 31");
       range([32],      "FIFO full");
     };
```

Getting Started with the Verification Environment Flow

Ensure that you have a simulation tool installed and properly set up.

The rest of this section discusses:

- Package Directory Structure and Contents
- IntelliGen
- Run Scripts
- UVM ***e*** UVC
- Test Cases
- Running a Simulation

Package Directory Structure and Contents

The Package Contents table below explains the UART ***e*** Module level environment directory structure and contents. This package is located at the following location:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl
```

Table 1. Package Contents and Description

Directory	Filename	Description
`uart_ctrl`	`PACKAGE_README.txt` `demo.csh`	Describes UART environnent package usage instructions. Demo sim for the module-level environnent.
`uart_ctrl/e`	`*.e`	The ***e*** files for the UART environment are located in this directory.

uart_ctrl/e/checker	uart_ctrl_scoreboard.e	Data checkers implemented using UVM ***e*** Scoreboard Package. UART to APB UVC data checking and APB UVC to UART data checking are performed.
uart_ctrl/e/cover/	uart_ctrl_cover.e	Functional coverage file for DUT-specific features - paths point to signals in RTL.
uart_ctrl/e	uart_ctrl_apb_config.e uart_ctrl_uart_config.e uart_ctrl_define.e	Configuration details of APB UVC for use in the module level environment. Configuration file for UART. Define used in the UART module level environment
uart_ctrl/sve/simvision	simvision.svcf	SimVision Command script
uart_ctrl/sve/testbench	tb_uart.v	Test Bench top for UART module level environment.
uart_ctrl/sve/scripts/	run_sim.sh covfile.cf irun_batch.tcl nc_waves.tcl	Simulation run script. ./run_sim.sh –h[elp] will provide all command line options Configuration file to set code coverage options. TCL file used for batch mode of simulation. TCL file used for waveform dumping.
uart_ctrl/sve/tests/	data_poll.e test_uart.e data_poll_virtual.e	Test APB and UART traffic, implemented using MAIN sequences. Negative test case added to show parity error. Test APB and UART traffic, implemented using virtual sequence.

IntelliGen

IntelliGen is constrained-random generation technology in Specman and is the default in 12.2. It has improved functionality and performance compared to Pgen. The block UART environment is compatible with IntelliGen, which will be used for all the simulations.

Run Scripts

You can find the run scripts developed for the UART environment at the following location:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl/sve/scripts
```

A description of these scripts is given below.

- **run_sim.sh** - This script compiles UART RTL, Verilog test bench, and *e* code.
  - Constructs the irun command line including sourcing covfile.cf for coverage options.
  - It can run in batch, interactive, interactive debug mode.

  Usage:

  ```
  run_sim.sh -test <test_name> -run_mode
     <batch|interactive_debug|interactive|batch_debug>
  ```

- **covfile.cf** - Coverage configuration file. This file is used to inform irun how to elaborate the design so that coverage can be collected if desired. It is sourced during irun compilation using inclusion in the following file: `uart_ctrl/sve/uart_ctrl.irunargs`

UVM *e* UVC

The APB interface UVC is UVM ***e*** compliant and is available at:

```
$SOCV_KIT_HOME/uvm_e_ex_lib/interface_uvc_lib/apb/
```

The APB UVC is reusable and can work flawlessly with any *e*RM-compliant verification environment.

The UVM-compliant APB UVC has testflow phases incorporated in agent and BFM.

A new struct-member construct is provided with the following single keyword: `tf_testflow_unit.` This declares a unit's participation in a UVM testflow scheme. This construct is implemented as a macro that expands the declaration of phase TCMs, the clocking event, and auxiliary fields and methods.

`tf_env_setup(),tf_hard_reset(),tf_reset(),tf_init_dut(),tf_init_link(),tf_main_test(), tf_finish_test()`and `tf_post_test()` are the TCMs introduced in the unit scope, one for each of the testflow phases.

Each unit that participates in the testflow uses the clocking event `tf_phase_clock`. For each unit, this clock can be linked to different events (external or internal), according to the phase the unit is going through. Linking the clock to different events in different phases is called clock switching.

To automate clock switching, use the macro `CLOCK_SWITCH_SCHEME`.

Usage example:

```
extend apb_master_driver_u {
    CLOCK_SWITCH_SCHEME {ENV_SETUP;MAIN_TEST}
        {p_env.unqualified_clock_rise;p_env.clock_rise};
};
```

The APB UVC uses testflow sequence that participates in the testflow scheme. Testflow in sequence declaration is as follows:

```
sequence name [using sequence_option,...]
sequence_option:
..
testflow = TRUE
```

`MAIN MAIN_TEST` sequence should replace the base MAIN sequence in the testflow model. The predefined behavior of `MAIN MAIN_TEST` is identical to that of the MAIN sequence.

Usage of the MAIN MAIN_TEST sequence is as follows:

```
extend MAIN MAIN_TEST sequence-name {
   count: uint;
   !sequence: sequence-name;
   keep soft count == 10;
   body() @driver.clock is only {
     for i from 1 to count do {
       do sequence;
     };
    };
};
```

In a UVM ***e***-compatible UVC (APB UVC in our case), the basic verification units should be inherited from UVM base types like `uvm_bfm, uvm_agent,uvm_signal_map, uvm_env, uvm_monitor,` and so on.

`uvm_active_passive_t` should be constrained to either ACTIVE or PASSIVE depending upon the use.

All ports used in the UVC should be of TLM types. To add a scoreboard, we recommend using the UVM ***e*** scoreboard instead of implementing a scoreboard from scratch.

The module as well as subsystem-level UART environment uses the UVM ***e***-compliant APB UVC.

Test Cases

The UART environment contains test cases, which are described in the UART Environment Test Cases table.

Table 2. UART Environment Test Cases

Test Case	Description
`data_poll.e`	This is a basic sanity test case, which: • Configures the UART DUT to the same chosen mode as the UART UVC by programming the configuration registers. • Commands the UART UVC to transmit randomized frames to the UART DUT receiver input. • Commands the UART DUT to transmit randomized frames. These frames are queued sequentially based on the UART DUT TX FIFO threshold flag to prevent overflow of the UART DUT FIFO. • Polls the UART DUT RX FIFO threshold flag to respond to received frames and to read them from the FIFO via the APB. • For frames transmitted by the DUT, the UVM Scoreboard monitors that the frame data written to the DUT TX FIFO is successfully received by the UVC receiver monitor. • For frames received by the DUT, the UVM ***e*** Scoreboard monitors that the frame data transmitted by the UVC transmitter BFM is successfully read from the DUT RX FIFO. The test case is run using UVM ***e*** complaint APB UVC and MAIN_TEST testflow phase is used in it. The test is located in the `uart_ctrl/sve/tests` directory.
`test_uart.e`	This is a negative test case where parity error is introduced. The UVM ***e*** scoreboard is supposed to throw Packet Mismatch Error.
`data_poll_virtual.e`	Basic UART datapath test case implemented using virtual sequence

Running Module-Level Simulation

To run a simulation:

1. Make a work directory in a user-chosen area.
2. Compile the DUT, test environment, and chosen test case using the following command:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl/sve/scripts/run_sim.sh
    -test <test_name> -run_mode <batch|interactive_debug|interactive|batch_debug>
```

Note: Choose a test from the `uart_ctrl/sve/tests` directory.

Example:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl/sve/scripts/run_sim.sh
    –test data_poll.e -seed 1 –run_mode batch
```

3. Depending on the run mode you chose, do one of the following:

   - When using batch command line mode, the simulation starts and terminates automatically. Ensure there are zero DUT and zero DUT warnings reported in the transcript output.
   - When using the interactive GUI mode, SimVision starts. In the SimVision console command line, enter run to start the simulation.

   When the simulation finishes, make sure that there are zero DUT and zero DUT warnings reported in the transcript output.

4. If running in interactive GUI mode, enter `exit` in the SimVision console command line to finish the simulation and close the simulator.

UVM *e* Subsystem-Level Verification

This section describes how a verification environment can be constructed to verify the functionality at the cluster or subsystem level with UVM ***e***, which reuses the existing module level components from the UART environment. For this demonstration, the chosen subsystem contains AHB Bus Matrix, two UARTs, SPI, and GPIO that are included within the UVM ***e*** Reference Flow. A UVM ***e*** environment is built around the APB Subsystem by reusing the existing module/block level UART verification environment components.

This section illustrates an implementation based on UVM ***e***.

The APB Subsystem DUT

The APB subsystem contains UART, SPI, GPIO, and other blocks as shown in the figure APB Subsystem Verification. These blocks are written in synthesizable Verilog RTL. The Segment Representative Design (SRD) contains two instantiations of the UART IP module one to show the low power features.

The DUT has the following interface which needs to be driven or monitored by the UVCs:

- AHB interface
- APB(WISHBONE) interface
- UART interface
- GPIO interface
- SPI interface
- Clocks and Resets

APB Subsystem Verification – UVM *e* Reference Flow Methodology

The APB subsystem environment is implemented in ***e*** mostly with reference to UVM.

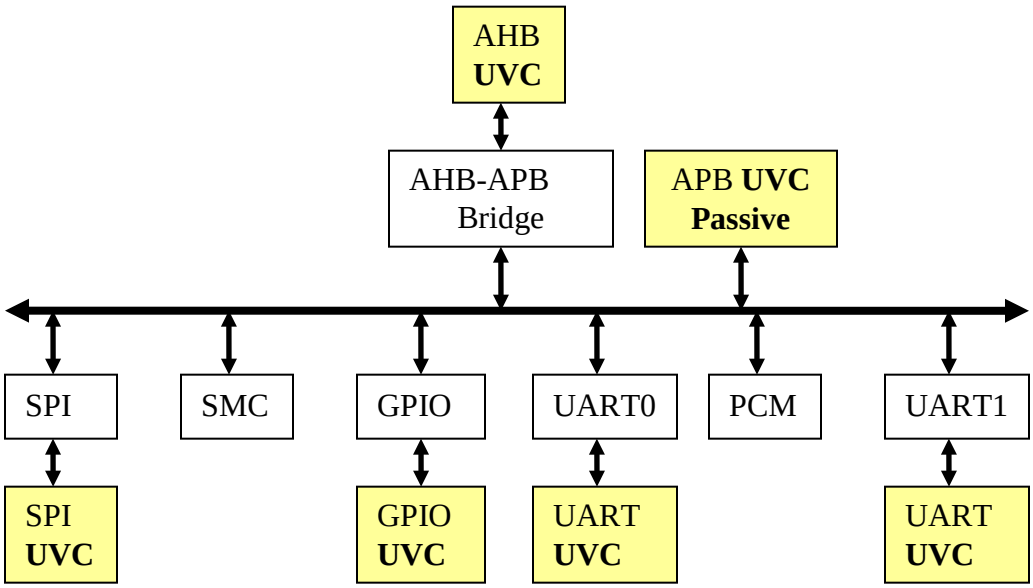
The environment consists of several UVCs, like SPI, GPIO, AHB, and UART that are mainly *e*RM-compliant but incorporate very minimal adoption of UVM ***e*** features (like usage of UVM base types, UVM active passive fields, and so on).On the other side, the APB UVC is UVM ***e*** complaint. This ensures that *e*RM and UVM ***e***-compatible environments can be used together flawlessly.

The scoreboards used in AHB-SPI and AHB-UART interfaces are implemented using UVM scoreboards. The UART module-level scoreboard is also re-used in the subsystem level.

The UVM ***e*** scoreboard is built as a package that can be used very easily in developing the scoreboard module.

A simplified version of the APB Subsystem environment is constructed as shown in the APB Subsystem Verification figure.

Figure 2: APB Subsystem Verification



Top-Level Environment: apb_subsystem

The top-level verification environment instantiates all the individual UVCs, and extends them. The files that perform this are in `$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/e`.

The top-level verification environment uses Boolean fields (`has_gpio, has_spi,` etc.) to build a highly configurable environment that supports different flows for cluster and module level verification in the APB Subsystem.

The APB subsystem provides all verification tests to be run in the top-level verification environment.

GPIO UVC: gpio

The GPIO UVC is a simple UVC that stimulates and checks the general purpose IO interface of the DUT. The files that implement this are in:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/gpio
```

AHB UVC: ahb

The APB UVC can support multiple slaves and can drive as well as monitor transaction at AHB interface. It is implemented in Specman ***e*** using basic UVM ***e*** constructs.

The files that implement this are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/ahb
```

APB UVC: apb

The APB UVC has been included in configurations for verifying APB peripherals as blocks within the DUT. The files that implement this are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/apb
```

UART UVC: uart

The UART UVC forms the basis of the UART implementation. The reference flow package uses it extensively to show reuse from module to cluster to system level. The files that implement this are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/uart
```

There are two instances of the UART interface, so two UART UVC environments are used.

APB/UART Module UVC: uart_ctrl

If the configuration of the DUT requires both a UART and the APB, then the subsystem module UVC combining the two is included. If the configuration requires both UARTs and the APB, then this module is instantiated twice. The files that implement this are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/uart_ctrl
```

SPI UVC: spi

The SPI UVC implements the basic SPI protocol using ***e***. The files that implement this are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/interface_uvc_lib/spi
```

Verilog Verification Components

The Verilog components are at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/testbench
```

The top-level Verilog module tb_apb_subsystem() comprises the DUT testbench.

Configuration Library

The top-level configuration file for the APB subsystem *e* environment is found at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/e/apb_subsystem_sve
.e
```

In this file, the top-level `sys` is extended to instantiate the individual VE. This is the only place that sys is referenced as `apb_subsystem_sve_u` is normally used through the VE so that it can be more readily reused in other VEs. However, in this configuration file, we instantiate `apb_subsystem_sve_u` under `sys`.

Test Cases

There are three test cases that are included in the APB verification environment.

Test cases include basic data path poll access. Test cases are available at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/tests/
```

APB Subsystem Peripheral Test

Tests are targeted from AHB to subsystem peripherals GPIO, SPI & UART.

Scoreboards

The UVM ***e*** scoreboard infrastructure implements the scoreboard logic.

AHB - UART Scoreboard

This scoreboard checks data transmission from UART to AHB and AHB to UART path. The UVM ***e*** scoreboard package is used to develop the same. The scoreboard code is available at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/e/apb_subsystem_checker
    /apb_subsystem_ahb_uart_uvm_scoreboard.e
```

AHB - SPI Scoreboard

This scoreboard checks data transmission from SPI to AHB & AHB to SPI path. UVM ***e*** scoreboard package is used to develop the same. The scoreboard code is available at:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/e/apb_subsystem_checker
    /apb_subsystem_ahb_spi_uvm_scoreboard.e
```

Getting Started with the *e* Verification Environment Flow

Ensure you have a simulation tool installed and properly set up.

The rest of this section discusses:

- Package Directory Structure and Contents
- Test Cases
- Run Scripts
- Running a Simulation

Package Directory Structure and Contents

The table below explains the APB Subsystem directory structure and contents. This package is located at the following location:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem
```

Package Contents and Description

Directory	Filename	Description
apb_subsystem	PACKAGE_README.txt	Describes APB environment package usage instructions.
	demo.csh	Demo sim for the subsystem level environment.
apb_subsystem/e	*.e	The *e* files for the UART-APB subsystem environment are located in this directory.
apb_subsystem/e/apb_subsystem_checker	apb_subsystem_ahb_spi_uvm_scoreboard.e apb_subsystem_ahb_uart_uvm_scoreboard.e	This contains the scoreboard units for data streaming from UART, SPI to AHB. Scoreboard is developed using UVM ***e*** package
apb_subsystem/e/apb_subsystem_ahb_config	apb_subsystem_ahb_config_top.e apb_subsystem_ahb_config.e	This contains AHB bus configurations.
apb_subsystem/e/apb_subsystem_apb_config	apb_subsystem_apb_config.e	This contains APB configuration & signal mapping for APB UVC.
apb_subsystem/e/apb_subsystem_uart_config	apb_subsystem_uart_config.e	This contains UART configuration & signal mapping for UART UVC.
apb_subsystem/e/apb_subsystem_spi_config	apb_subsystem_spi_config.e	This contains SPI configuration & signal mapping for SPI UVC.
apb_subsystem/e/apb_subsystem_gpio_config	apb_subsystem_gpio_config.e	This contains GPIO configuration & signal mapping for GPIO UVC.
apb_subsystem/e/apb_subsystem_cover	apb_subsystem_cover.e	This contains APB subsystem coverage.
apb_subsystem/e	apb_subsystem_env_h.e	Contains environment unit declaration.
	apb_subsystem_env.e	Contains environment variable declarations & scoreboard connections.
	apb_subsystem_ahb_seq_lib.e apb_subsystem_gpio_seq_lib.e apb_subsystem_spi_seq_lib.e apb_subsystem_uart_seq_lib.e	Contains reusable sequences.
	apb_subsystem_monitor.e	Contains scoreboard instances.

	apb_subsystem_vir_seq.e	Contains virtual sequence.
	apb_subsystem_reg.e apb_subsystem_reg_config.e apb_subsystem_reg_seq_lib.e	Contains register definitions, extensions & sequences.
	apb_subsystem_top.e	Contains all imports.
apb_subsystem/sve/scripts/	run_sim.sh	Run a stand-alone sim. ./run_sim.sh –h[elp] will provide all command line options.
	run_com.sh	Only RTL and Verilog testbench compilation script.
	covfile.cf	Coverfile to set code coverage options.
	nc_waves.tcl	Contains common declarations for signal probing
	irun_batch.tcl	TCL file to invoke simulator run.
apb_subsystem/sve/tests	apb_subsystem_data_poll.e	Test from AHB to UART, SPI & GPIO implemented using MAIN sequence
	apb_subsystem_data_poll_virtual.e	Test from AHB to UART, SPI & GPIO implemented using virtual sequence
	apb_subsystem_smc_uart_pd_pu.e	Test from AHB to UART, SPI and GPIO implemented using MAIN sequence. It also performs writes into the Power Control Register.
apb_subsystem/sve/e/	apb_subsystem_sve.e	Top integration file
apb_subsystem/sve/testbench	tb_apb_subsystem.v	Top-level Verilog testbench file

Test Cases

The APB *e* environment contains test cases in the `apb_subsystem/sve/tests` directory, which are described in the following table.

Sl. No.	Test Case	Description
1.	`apb_subsystem_data_poll.e`	AHB transactions to UART, SPI & GPIO using MAIN sequences. This test is the basic data path test case implemented using MAIN Sequence.
2.	`apb_subsystem_data_poll_virtual.e`	AHB transactions to UART, SPI & GPIO using virtual sequence. This test is the basic data path test case implemented using Virtual Sequence.
3.	`apb_subsystem_smc_uart_pd_pu.e`	Test from AHB to UART, SPI and GPIO using MAIN sequence. It also performs writes into the Power Control Register.

Run Scripts

You can find the run scripts developed for the APB environment at the following location:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/scripts
```

A description of these scripts is given below.

- **run_sim.sh** – This script compiles APB Subsystem RTL, Verilog test bench, and *e* code.
  - Constructs the irun command line including sourcing `covfile.cf` for coverage options.
  - It can run in batch , interactive, interactive debug mode.
  - If a seed is not specified, random seed is selected.

  To get all the run_sim.sh command line options, execute `./run_sim.sh –h[elp]`

  Usage :

  ```
  run_sim.sh -test <test_name> -run_mode
  <batch|interactive_debug|interactive|batch_debug> –seed <seed_num>
  ```

- **covfile.cf** – This is the code coverage configuration file. This file is used to inform irun how to elaborate the design so that coverage can be collected if desired. It is sourced during irun compilation via inclusion in the following file: `apb_subsystem/sve/apb_subsystem.irunargs`

Running Subsystem-Level Simulation

To run a simulation:

1. Make a work directory in a user-chosen area.
2. Compile the DUT, test environment and chosen test case using the following command:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/scripts/run_sim.sh
    -test <test_name>  -run_mode <batch|interactive_debug|interactive|batch_debug>
    –seed <seed_num>
```

   Note: Choose a test from the `apb_subsystem/sve/tests` directory.

Example:

```
$SOCV_KIT_HOME/soc_verification_lib/uvm_e_ex_lib/apb_subsystem/sve/scripts/run_sim.
    sh -test apb_subsystem_data_poll.e -seed 1 –run_mode batch
```

3. Depending on the run mode you choose, do one of the following:
   - When using batch command-line mode, the simulation starts and terminates automatically. Ensure there are zero DUT and zero DUT warnings reported in the transcript output.
   - When using the interactive GUI mode, SimVision starts. In the SimVision console command line, enter `run` to start the simulation.

   When the simulation finishes, make sure that there are zero DUT and zero DUT warnings reported in the transcript output.
4. If running in interactive GUI mode, enter `exit` in the SimVision console command line to finish the simulation and close the simulator.