Step-by-step Tutorial for Connecting Questa® VIP into the Processor Verification Flow

by Marcela Zachariasova, Tomas Vanak and Lubos Moravec–Codasip Ltd.

Verification Intellectual Properties (VIPs) play a very important role in the verification flow of modern SoCs. They can check the correctness of communication over system buses and provide master, slave, decoder, or arbiter components if these are missing in the verification set-up. This article describes verification of RISC-V processors, focusing on the combination of automatically generated UVM verification environments by QVIP Configurator and Questa® VIP (QVIP) components. The section with step-by-step instructions will demonstrate how to add QVIP components into processor verification environments.

RISC-V

RISC-V is a new processor architecture with a free to use, modern and open Instruction Set Architecture (ISA) definition. It was originally designed by the University of California, Berkeley, to support research and education. Eventually, RISC-V expanded and established itself in the industry, thanks to the RISC-V Foundation which is now covering most of the activities related to RISC-V.[1]

RISC-V standard includes the definition of base integer instruction set ("I" for 32 general-purpose registers or "E" for 16 general-purpose registers) and optional ISA extensions, such as multiplication and division extension ("M"), compressed instructions extension ("C"), atomic operations extension ("A"), floating-point extension ("F"), floating-point with double-precision extension ("D"), floating-point with quad-precision extension ("Q"), and other experimental and custom extensions. However, RISC-V does not include any hardware microarchitecture definitions, which means that IP vendors can implement and sell varied versions of RISC-V processor IPs with distinctive features and multiple bus interfaces.

This article describes verification of communication of the RISC-V core with its surrounding components using QVIP. RISC-V compliant processor cores and components referenced in this article were implemented by the Czech-based IP company Codasip.[2]

AUTOMATICALLY GENERATED UVM VERIFICATION ENVIRONMENTS

One of the main benefits of the Codasip solution is that apart from an off-the-shelf RISC-V IP series Codix Berkelium (Bk), the company also provides an EDA tool for processor development and customization. The tool describes processors at a higher abstraction level in Architecture Description Language called CodAL. From this description, many outputs are automatically generated, including the RTL representation of the processor, a complete set of software tools such as compilers, linkers, simulators, debuggers and profilers, and an UVM verification environment with a random assembler programs generator. This allows for very fast customization and optimization of the cores according to the user's individual needs.

From a verification perspective, the RTL generated for a specific RISC-V core serves as the Design Under Test (DUT). The Codasip automation flow can also generate functional instruction-accurate reference models. "Functional" means that the reference description does not contain micro-architectural details like timing, pipelining, or any kind of hardware blocks. For illustration, see Fig. 1.

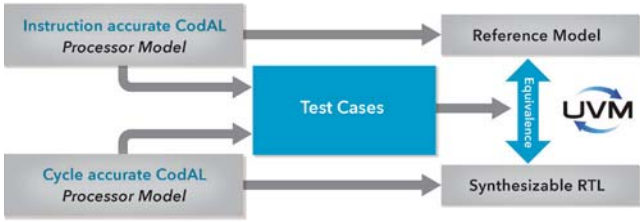


Figure 1: UVM Environment Generation

For each generated assembler program, DUT outputs are automatically compared to those of the reference model. Consequently, the generated UVM verification environment is processor-specific, and for the purposes of UVM generation, all necessary information is extracted from the high-level processor description in CodAL.

The generated verification environment has a standard UVM-style structure. For the purpose of this article, a detailed description is not needed, so only a simple block scheme is provided in Fig. 2. It illustrates the relation of the Berkelium core (DUT), connected main memory, and several UVM agents:

- ASIP agent – the main processor agent. It includes assertion checkers and coverage monitors for the processor ports and some internal signals. It also handles driving of input ports, mainly interrupt ports.
- Registers agent – monitors access to internal registers of the processor.
- Decoder agent – monitors coverage of executed instructions and combinations of instructions.
- Memory agent – handles loading of test programs to the program part of main memory, and monitors access to the memory while instructions are executed.

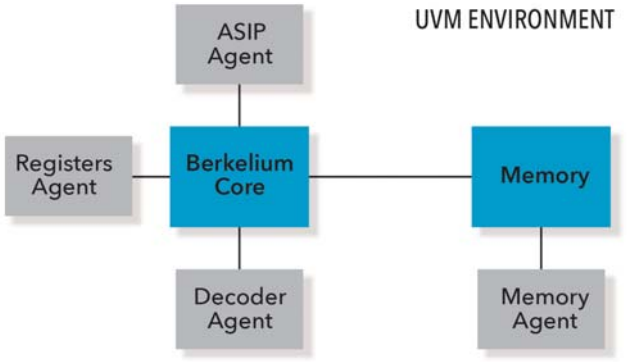


Figure 2: Simplified Block Scheme of the Generated UVM Environment

For further reference, it is important to note that the Berkelium core interface (for connecting the memory and optional peripherals, not depicted in Fig. 2) is an AMBA-like interface.

QUESTA® VIP CONFIGURATOR

For easier integration of QVIP, there is a tool called QVIP Configurator. The tool is provided in the installation package along with supported protocol VIPs, and it can be used to create QVIP test benches based on UVM, which include a basic set of test bench building blocks. The steps to generate the test-bench are simple: After opening QVIP Configurator, required testbench components can be added and configured, such as instances of DUTs, modules, VIP protocols, and memory modules in the testbench project. It is also possible to set the clock and reset modules, define address mapping, and select available sequences.

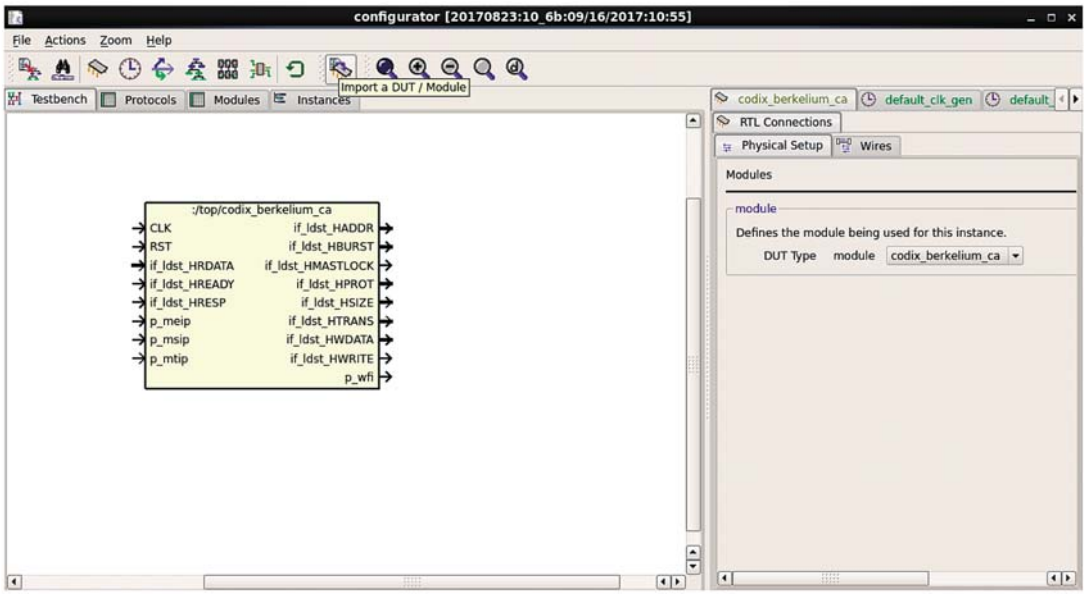


Figure 3: DUT Module Added into QVIP Configurator Environment

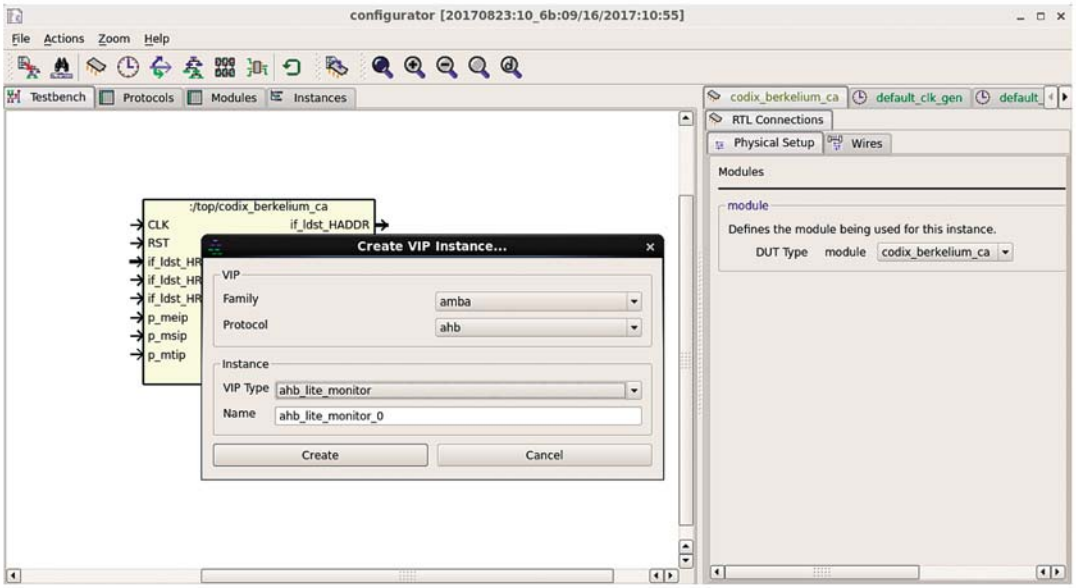


Figure 4: Create VIP Instance Windows in QVIP Configurator Environment

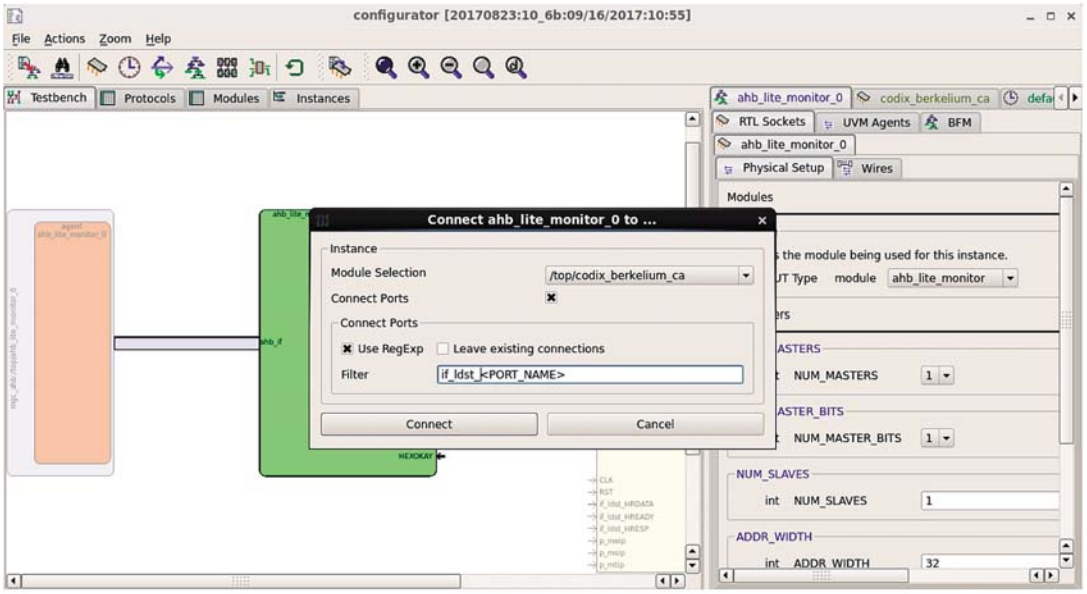


Figure 5: Connection Window for Interconnects in QVIP Configurator Environment

Step 1

First of all, the DUT component must be added. Let's use a Codix Berkelium IP core in this test case. To add the DUT module, navigate to the menu "File > Import a DUT / Module", or use the icon shown in Fig. 3.

Step 2

Next, VIP components for AHB-lite protocol are added. This can be done via the menu "Actions > VIP Instance", or by right-clicking in the workspace and selecting "Create VIP Instance". The *Create* window is shown in Fig. 4. Two newly added components (QVIP Monitor and UVM Agent) are generated in one step as the Monitor component is configured and driven by its UVM agent.

Step 3

Now we need to establish connection between the DUT and the QVIP monitor component. For this step, it is recommended to use the Connect Instance wizard with module selection for components that shall be connected. To launch the wizard, right-click the QVIP component in the workspace and select "Connect Instance".

Step 4

When steps 1–3 are completed, the testbench is ready to be generated. To start, navigate to the

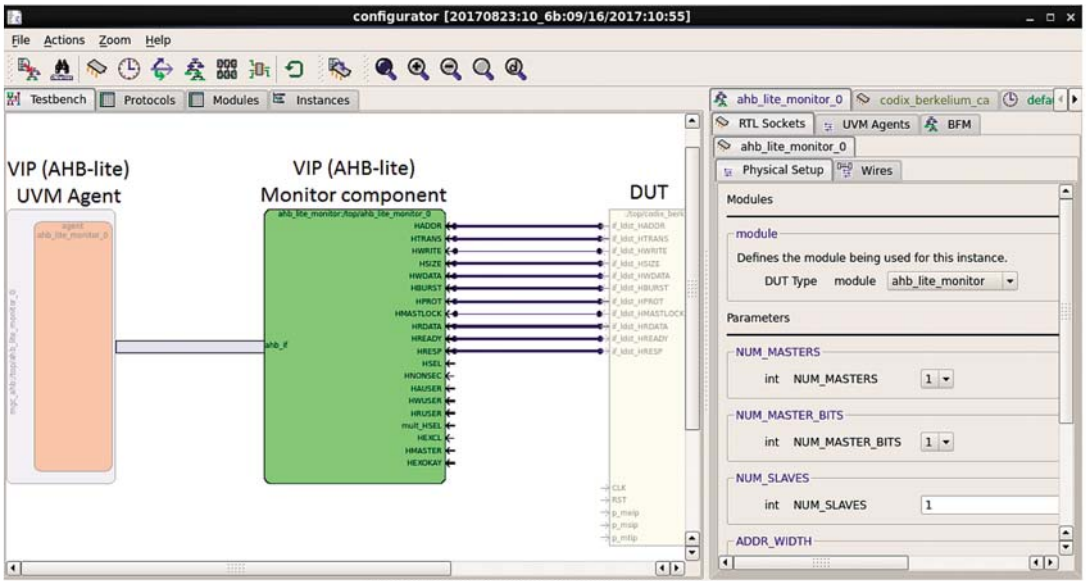


Figure 6: Fully Connected QVIP Configurator Environment

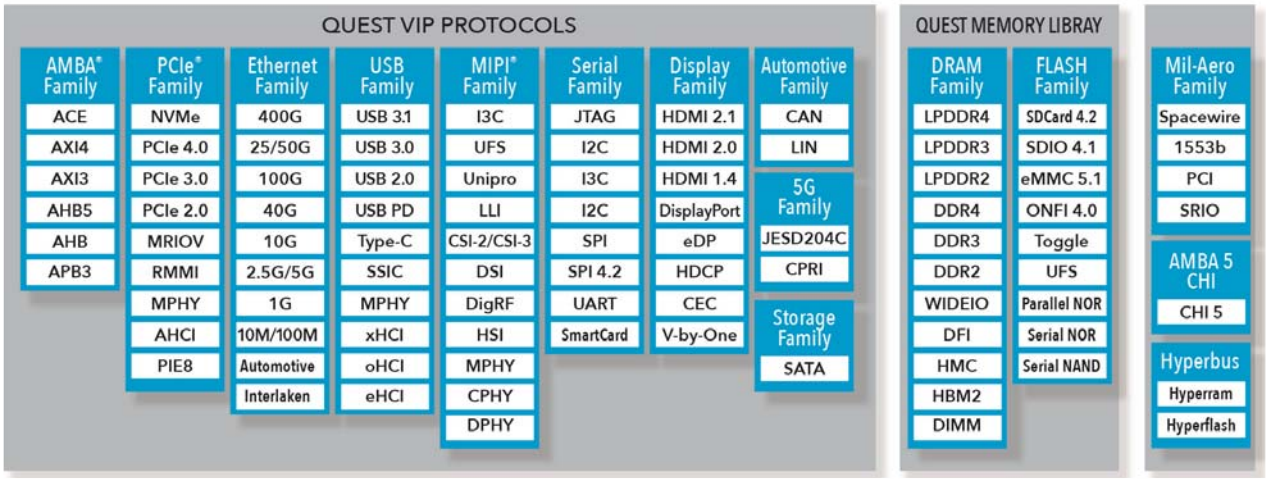


Figure 7: Questa VIP Protocols Support

menu "Actions" and select "Generate Testbench". An example of a fully connected test environment is shown in Fig. 6.

QVIP AS MONITOR (PROTOCOL CHECKER)

One of the main purposes of QVIP is to verify that communication over a system bus complies to the communication protocol. This task is performed by assertion checkers and protocol coverage ensured by comprehensive QVIP inbuilt sequences. There are versions of QVIP that support various industry standard interfaces, see the overview in Fig. 7.

A step-by-step tutorial follows on how to connect the QVIP protocol checker for AHB-Lite bus connection between the Berkelium core and the main memory. All steps are illustrated by the block scheme in Fig. 8 on the following page. The example is based on the testbench generated from QVIP Configurator as described in the previous chapter. The method can be used also for very complicated SoCs.

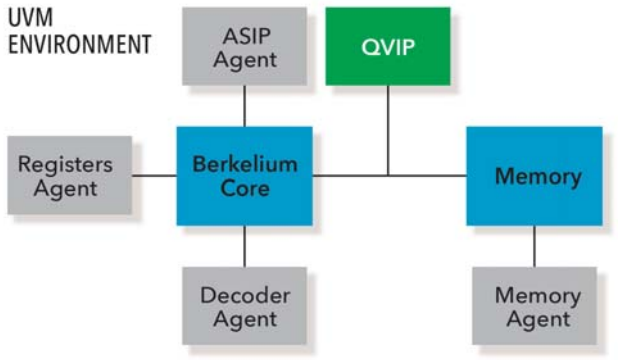


Figure 8: Inserting QVIP As a Monitor (Protocol Checker)

1. First, a path to Questa® VIP package must be set as an environment variable.

   export QUESTA_MVC_HOME=<PATH_TO_VIP_INSTALLATION>/Questa_VIP_10_6_b

2. In our example, we use Questa® as the main RTL simulator. When running Questa® with QVIP, parameter –mvchome must be used with the vsim command.

   vsim -gui **-mvchome** <PATH_TO_VIP_INSTALLATION>/Questa_VIP_10_6_b -do "do questa/start_gui.tcl -codix_berkelium xexes/crc.xexe"

3. Compilations of the QVIP package and the AHB package are part of a compilation script that needs to be created. In our example, the script is called *start_gui.tcl*. When another protocol is used, then another QVIP package is compiled.

```
# Compiling infrastructure files
vlog -sv $QUESTA_MVC_HOME/include/questa_mvc_svapi.svh
vlog +define+MAP_PROT_ATTR \
    +incdir+$QUESTA_MVC_HOME/questa_mvc_src/sv \
    $QUESTA_MVC_HOME/questa_mvc_src/sv/mvc_pkg.sv
# Compiling protocol package
vlog +define+MAP_PROT_ATTR \
    +incdir+$QUESTA_MVC_HOME/questa_mvc_src/sv \
    +incdir+$QUESTA_MVC_HOME/questa_mvc_src/sv/ahb \
    $QUESTA_MVC_HOME/questa_mvc_src/sv/ahb/mgc_ahb_v2_0_pkg.sv
# Compiling module
vlog +define+MAP_PROT_ATTR \
    $QUESTA_MVC_HOME/questa_mvc_src/sv/ahb/modules/ahb_lite_monitor.sv
```

4. As indicated by the generated testbench code, components participating in the connection of QVIP instantiate QVIP interface and QVIP monitor module. Then the QVIP monitor signals are connected with the DUT.

```
mgc_ahb
#(
    .AHB_NUM_MASTERS(ahb_lite_monitor_0_params::AHB_NUM_MASTERS),
    .AHB_NUM_MASTER_BITS(ahb_lite_monitor_0_params::AHB_NUM_MASTER_BITS),
    .AHB_NUM_SLAVES(ahb_lite_monitor_0_params::AHB_NUM_SLAVES),
    .AHB_ADDRESS_WIDTH(ahb_lite_monitor_0_params::AHB_ADDRESS_WIDTH),
    .AHB_WDATA_WIDTH(ahb_lite_monitor_0_params::AHB_WDATA_WIDTH),
    .AHB_RDATA_WIDTH(ahb_lite_monitor_0_params::AHB_RDATA_WIDTH)
)
ahb_lite_monitor_0_bfm
(
    .iHCLK(codix_berkelium_ca_CLK),
    .iHRESETn(codix_berkelium_ca_RST)
);

ahb_lite_monitor·
#(
    .NUM_MASTERS(ahb_lite_monitor_0_params::AHB_NUM_MASTERS),
    .NUM_MASTER_BITS(ahb_lite_monitor_0_params::AHB_NUM_MASTER_BITS),
    .NUM_SLAVES(ahb_lite_monitor_0_params::AHB_NUM_SLAVES),
    .ADDR_WIDTH(ahb_lite_monitor_0_params::AHB_ADDRESS_WIDTH),
    .WDATA_WIDTH(ahb_lite_monitor_0_params::AHB_WDATA_WIDTH),
    .RDATA_WIDTH(ahb_lite_monitor_0_params::AHB_RDATA_WIDTH)
)
ahb_lite_monitor_0
(
    .ahb_if(ahb_lite_monitor_0_bfm),
    .HADDR(codix_berkelium_ca_if_ldst_HADDR),
    .HTRANS(codix_berkelium_ca_if_ldst_HTRANS),
    .HWRITE(codix_berkelium_ca_if_ldst_HWRITE),
    .HSIZE(codix_berkelium_ca_if_ldst_HSIZE),
    .HWDATA(codix_berkelium_ca_if_ldst_HWDATA),
    .HBURST(codix_berkelium_ca_if_ldst_HBURST),
    .HPROT(codix_berkelium_ca_if_ldst_HPROT),
    .HMASTLOCK(codix_berkelium_ca_if_ldst_HMASTLOCK),
    .HRDATA(codix_berkelium_ca_if_ldst_HRDATA),
    .HREADY(codix_berkelium_ca_if_ldst_HREADY),
    .HRESP(codix_berkelium_ca_if_ldst_HRESP),
    .HSEL(ahb_lite_monitor_0_HSEL),
    .HNONSCE(ahb_lite_monitor_0_HNONSCE),
    .HAUSER(ahb_lite_monitor_0_HAUSER),
    .HWUSER(ahb_lite_monitor_0_HWUSER),
    .HRUSER(ahb_lite_monitor_0_HRUSER),
    .mult_HSEL(ahb_lite_monitor_0_mult_HSEL),
    .HEXCL(ahb_lite_monitor_0_HEXCL),
    .HMASTER(ahb_lite_monitor_0_HMASTER),
    .HEXOKAY(ahb_lite_monitor_0_HEXOKAY)
);
```

5. It is important to set the agent configuration in the UVM environment configuration object. Below, there is an example of the generated configuration *ahb_lite_monitor_0_cfg* integrated to our UVM environment.

```
// ahb agent configuration
ahb_lite_monitor_0_cfg_t ahb_lite_monitor_0_cfg;

// Constructor - creates new instance of this class
function new( string name = "m_codix_berkelium_ca_env_config_h" );
    super.new( name );
    ahb_lite_monitor_0_cfg = new();
endfunction: new
```

6. In our UVM environment, an AHB QVIP agent must be created. The package is *mgc_ahb_v2_0_pkg*.

```
// QVIP agent
ahb_lite_monitor_0_agent_t ahb_lite_monitor_0;

// create the agent
ahb_lite_monitor_0 =
    ahb_lite_monitor_0_agent_t::type_id::create( "ahb_lite_monitor_0", this );
// set the agent configuration
ahb_lite_monitor_0.set_mvc_config( m_cfg_h.ahb_lite_monitor_0_cfg );
```

7. Now we need to configure the AHB QVIP agent. All parameters can be set directly in the QVIP Configurator (see Fig. 9), and after testbench generation, they are located in the files *top_params_pkg.sv* and *<VIP_instance_name>_config_policy.svh*. The *is_active* parameter is set to 0 when just the protocol checker is in use. If QVIP supplies a master, a slave, a decoder or an arbiter, the parameter is set to 1. In such case, the *agent_type* parameter should be set either to AHB_MASTER, AHB_SLAVE, or AHB_ARBITER. The parameter *if_type* represents the type of the AHB interface, and it can be set to AHB, AHB_LITE, or ARM11_AHB. The parameter *en_cvg* enables coverage collection, the parameter *en_logger* enables creation of a log with transactions, the parameter *en_sb* enables checking that the written data are read correctly, and the parameter *en_txn_ltnr* enables printing transactions into the simulation transcript.

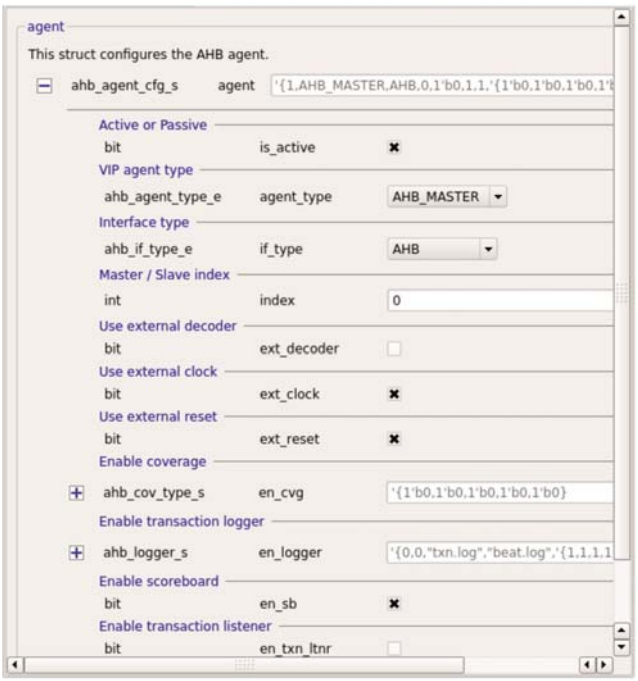


Figure 9: Agent Configuration Window from QVIP Configurator

QVIP AS MASTER/SLAVE

The previous chapter explained how to connect QVIP as a Monitor (Protocol Checker). There are more options to connect QVIP, for example as a Master or Slave component, if such components exist in the verified environment.

The scenario with QVIP as a Master component (shown in Fig. 10) is suitable for verification of existing Slave DUT components, for example AHB memory. The QVIP Master is equipped with test sequences, cover-points and assertions, like in the Monitor scenario. After running the testbench generated from QVIP Configurator, it is therefore possible to verify that the Slave DUT component is compliant with the tested communication protocol.

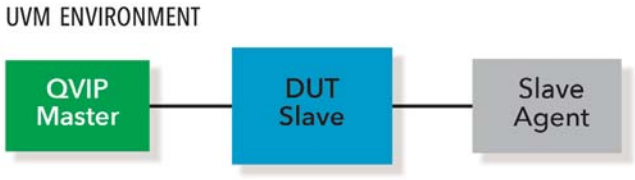


Figure 10: QVIP Connected as a Master Component

In the second scenario, QVIP is connected as a Slave component. In the default mode, it will simulate the behavior of a memory – the behavior can be set in Agent configuration. Successful testing guarantees that the Master DUT's communication protocol is compliant.

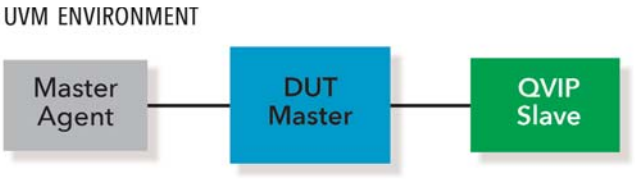


Figure 11: QVIP Connected as a Slave Component

CONCLUSION

This article provided a step-by-step tutorial for connecting QVIP into the processor verification flow. The tutorial explained the use of QVIP Configurator and described the crucial parts of the generated test bench, essential for debugging when misbehavior is reported. The main benefits of using QVIP and its Configurator are as follows:

- Efficiency: Replaces implementing complex UVM agents for each new bus protocol.
- Automation and rapidity: QVIP Configurator speeds up assembly of the verification environment as it generates the testbench automatically.

Codasip connected AHB QVIP to their RISC-V compliant Berkelium processors and, similarly to QVIP Configurator, automatically initiated and connected the QVIP agents.

REFERENCES

[1] RISC-V Foundation (2017, April) https://riscv.org/
[2] Codasip website (2017, April) https://www.codasip.com/

Mentor®
A Siemens Business
www.mentor.com

Editor:
Tom Fitzpatrick

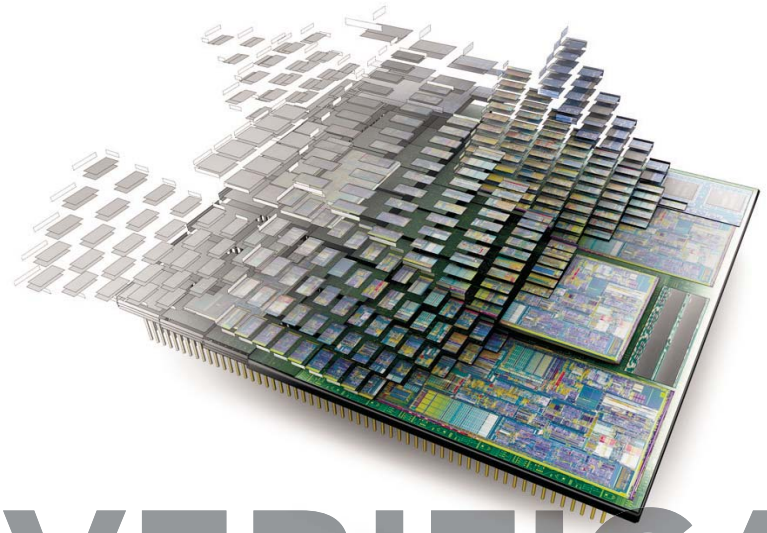
Program Manager:
Rebecca Granquist

Mentor, A Siemens Business
Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM



VERIFICATION HORIZONS

VH

VERIFICATION HORIZONS
CELEBRATING TEN YEARS