Highly Configurable UVM Environment for Parameterized IP Verification

Liu HongLiang
AMD Inc.
Andy.liu@amd.com

Whiting Karl
AMD Inc.
Karl.whiting@amd.com

ABSTRACT

A parameterized IP is configurable, which means the IP design can has different design parameters in different SoC, the design parameters can be port protocol, port number, port name and internal logic. Various IP design parameters significant affect verification environment infrastructure including test bench connections, drivers, monitors, stimulus sequences and functional coverage.

This paper presents a highly configurable advanced microcontroller bus architecture (AMBA) fabric IP verification environment with universal verification methodology (UVM), Synopsys AMBA VIP, and Ruby script. The verification environment can update itself automatically according to AMBA design parameters; it includes `IP_PREPROCESS`, `IP_TEST_BENCH` and `IP_MODULE_UVC.`

This paper also introduce how to do vertical reuse and horizontal reuse of test bench bind connection , verification IP(VIP) , sequence reuse, UVM configuration, scoreboard reuse, and coverage.

Our project results indicate the approach can reduce demand for additional resources by 30% when parameterized IP changes its design parameters and integrate to different SoC.

Keywords: **Configurable, Horizontal reuse, Parameterized IP, Vertical reuse, universal verification methodology(UVM)**.

1. Introduction

Parameterized IP design is an accelerating industry trend. Likewise, using one IP design suit different SoC projects with different parameters is an accelerating trend. These trends improve the efficiency of RTL development; however, they also lead new requirements for verification.

Janick Bergeron wrote “in the era of multi-million gate ASICs and FPGAs, reusable IP, and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort.”[1]. Verification usually rests squarely in the critical path of project schedule, it is essential to understand how to let one verification environment suit different IP parameters automatically, and require the least amount of effort when IP changes design parameters.

To reduce the verification effort, the verification environment must be highly configurable and easily reusable as well. This paper describes how to develop configurable and reusable

verification test bench components, and how to let them be configured by design parameters automatically. The approach is called a self-adapting IP verification environment in this paper.

2. Challenge to Verify Parameterized IP

Sharing parameters is a challenge when the design needs to be tested in multiple parameterized configurations[2]. Manually updating verification environment to use new design parameters is a complex work. It includes modifying test bench connections; increasing or decreasing the number of drivers, monitors, and scoreboards; updating configuration objects; rewriting stimulus sequences; and, adjusting functional coverage and assertions, all of these consume significant verification bandwidth.

A commonly used method is using macros (`ifdef) to distinguish different design parameters. Typically, one project would own a group of macros, test bench debug and maintain become complicated when the number of projects grows.

The next section introduces a self-adapting IP verification environment that deals with this challenge.

3. Self-adapting Parameterized IP Verification Environment Architecture

In this paper, the parameterized IP design is an advanced microcontroller bus architecture (AMBA) fabric; it connects a group of AXI, AHB, APB masters and slaves. The port width, port protocol, port number, internal register base address, and counter initial value can be different according to design parameters.

As shown in Figure 1, the self-adapting IP verification environment includes three parts- `IP_PREPROCESS`, `IP_TEST_BENCH` and `IP_MODULE_UVC`.

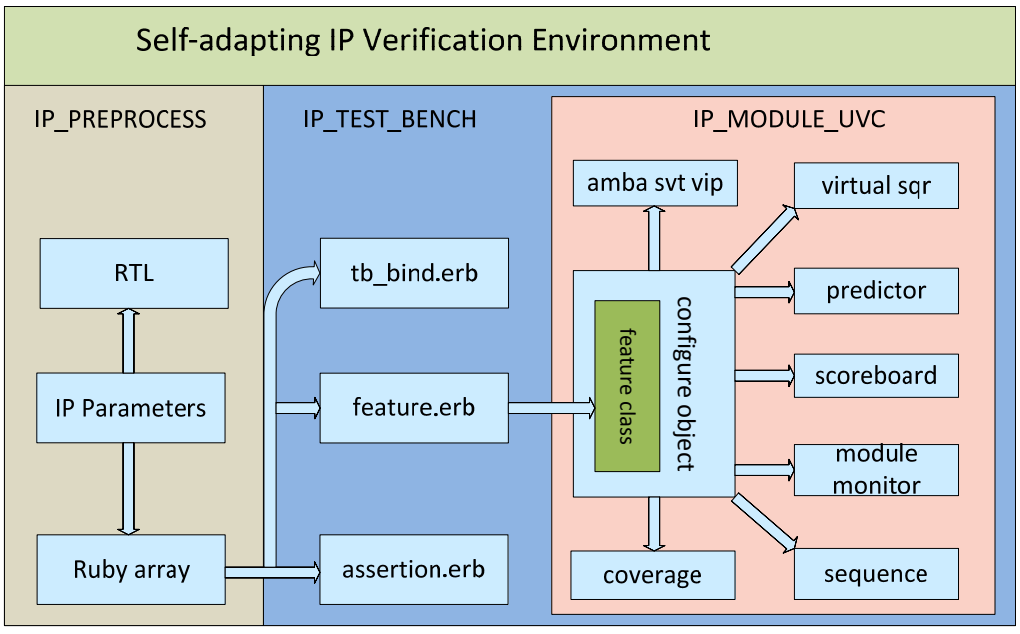


Figure 1 - self-adapting IP verification environment

3.1 IP_PREPROCESS

The IP preprocess mechanism requires putting all IP design parameters in one group, after the group is pre-processed, a parameter file is generated for design usage, and a global ruby array, @@FEATURES is generated for verification usage. This paper focuses on the verification side:

```
@@FEATURES = YAML::load('---
ip_component.axi_slaves: N
ip_component.axi_slv0_port: AXI0
ip_component.axi_slv0_addr_width: 32
ip_component.axi_slave_timeout: M
ip_component.axi_slv0_outstanding_wr:8
...
')
```

The global Ruby array @@FEATURES stores all IP design parameters. Whenever IP parameters change, the global Ruby array is updated automatically before VCS simulation runs.

3.2 IP_TEST_BENCH

IP test bench comprises an IP module UVM Component (UVC), a feature Ruby program, a module bind Ruby program and an assertion Ruby program. The three Ruby programs execute before VCS simulation, and capture the IP design's parameters automatically.

a) The IP module UVC's main function is to model behaviour by generating constrained random traffic, monitoring DUT responses, guaranteeing DUT's function works correctly, checking the validity of the protocol activity, and collecting coverage. Chapter five elaborates it more.

b) The feature Ruby program traverses the global Ruby array @@FEATURES. It converts all IP design parameters to properties of a System Verilog feature class - `ip_component_features` is instantiated as `ip_features` in this paper, whose properties store all IP design's parameters as following, the UVC is configured by this `ip_features`.

```
class ip_component_features extends uvm_object;
function new();
    axi_slaves = N;
    axi_slv0_port = "AXI0";
    axi_slv0_addr_width = 32;
    axi_slave_timeout = M;
    ...
endfunction : new
endclass
```

c) The module bind Ruby program connects the DUT module and test bench.

Below code segment describes that the number of Synopsys SVT AXI slave interface instance is controlled by IP design parameter "`axi_slaves`", the connection of AXI slave interface signal `wdata` and `awaddr` is controlled by IP design parameters:

```
bind `IP_COMPONENT_MODNAME svt_axi_slv_if
svt_axi_slave_if<%=i%> (
.awaddr(<%=features("axi_slv#{i}_port")%>_awaddr),
  );
```

This paper uses uvm_resouce_db to set bind interface instance in a top module:

```
uvm_pkg::uvm_resource_db#( virtual svt_axi_slave_if)::set
("uvm_interface_registry",top.svt_axi_slv_if0, top.
svt_axi_slv_if0)
```

Then the UVM test bench utilizes virtual interfaces at dynamic driver, monitor class to access static interfaces [3], it retrieves the bind interface instance handle by reading uvm_resouce_db:

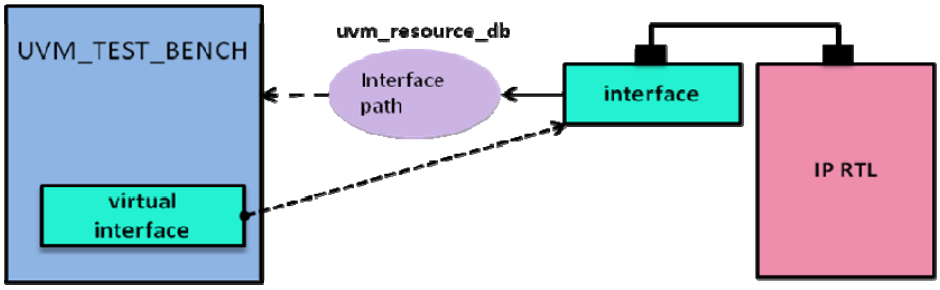


Figure 2 – connect RTL and UVM by interface bind

```
uvm_pkg::uvm_resource_db#( virtual
svt_axi_slave_if)::read_by_name ("uvm_interface_registry",
top.svt_axi_slv_if0, virtual_axi_if_variable).
```

This bind interface can be directly reused to chip level, because IP design module name stays the same in IP standalone environment and SoC full chip environment.

d) The assertion ruby program passes the IP design parameters to one type assertions related to IP design block function. These assertions are created according to IP design specification, and the assertion property uses local variables in its timing expressions, so it is self-adaptive.

These assertions are placed in an interface, then are bound to DUT module. For DUT module could be reused to SoC level and different projects, thus they accommodate in both vertical reuse and horizontal reuse:

```
bind `IP_COMPONENT_MODNAME ip_checker ip_checker(
   .axi_timeout_in(<%=features("axi_slave_timeout")%>),
 );
```

4. Self-adapting Parameterized IP Module UVC Structure

`IP_MODULE_UVC` is extended from uvm_env, which contains AMBA SVT VIPs, predictors, scoreboards, module monitor, IP module UVC configuration object, basic sequence and global virtual sequencer. Figure 2 illustrates `IP_MODULE_UVC`.

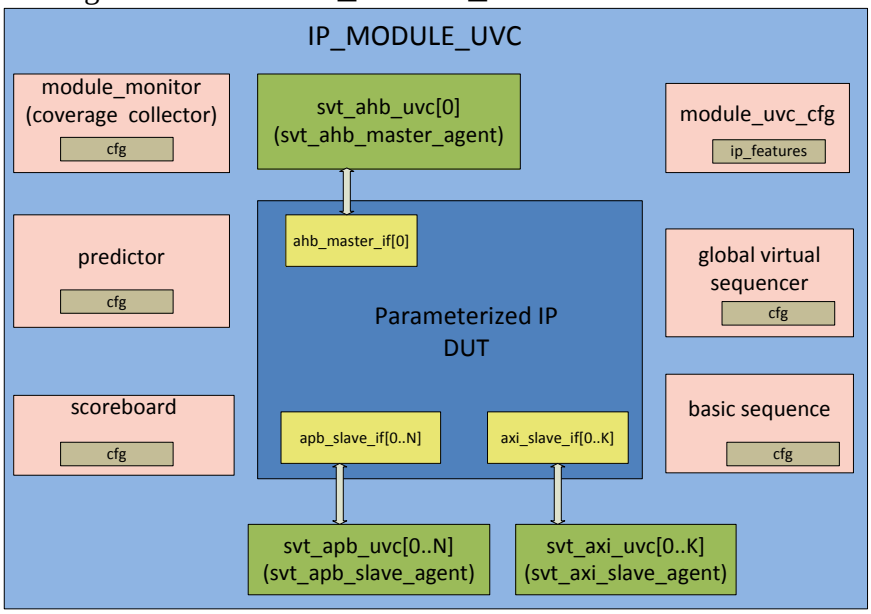


Figure 3 - IP_MODULE_UVC structure

The System Verilog feature class-`ip_component_features` generated at chapter 4.2 is instantiated in IP module UVC configuration object with name `ip_features`. Each component of the `IP_MODULE_UVC` can be configured and controlled by IP design parameters.

4.1 AMBA SVT VIPs

Synopsys AMBA SVT VIP is a UVC which contains master and slave agents, drivers, sequencers, monitors, configuration objects, protocol checkers, functional covergroups, constrained random sequences and transaction packets.

a) Instantiate AXI, AHB and APB SVT UVC
   As depicted in Figure 2, in order to let the agents numbers automatically get IP design parameters, this paper creates AXI, AHB and APB SVT UVC dynamical array in IP_MODULE_UVC, for instance AXI SVT UVC:

```
svt_axi_uvc  = new[cfg.ip_features.axi_slaves];
```

b) Configure AXI, AHB and APB SVT UVC
   The AMBA SVT UVC configuration classes[4] are instantiated in module UVC configuration object and assigned at the same place to capture IP design parameters. In UVM configuration mechanism[5], AXI SVT UVC, AHB SVT UVC and APB SVT UVC are configured by IP design parameters during the UVM build phase.

```
for (int i=0; i<ip_features.axi_slaves; i++)
axi_env_cfg[i].addr_width = ip_features.axi_slv_addr_width[i];
```

4.2 Self-adapting predictor and scoreboard.

The predictor takes in the real transaction packet from AMBA UVC TLM ports, puts out predicted transaction to scoreboard through TLM ports. The scoreboards check transaction

correctness.

As Figure 3 depicts, each slave has a scoreboard, both the predictor TLM port number and the scoreboard number are self-adapting to the IP design parameters - `ip_features`, they are all dynamic array data structure. When device number reduces, scoreboard number and TLM port number reduce automatically; when the device number increases, the scoreboard number and TLM port number increase automatically.

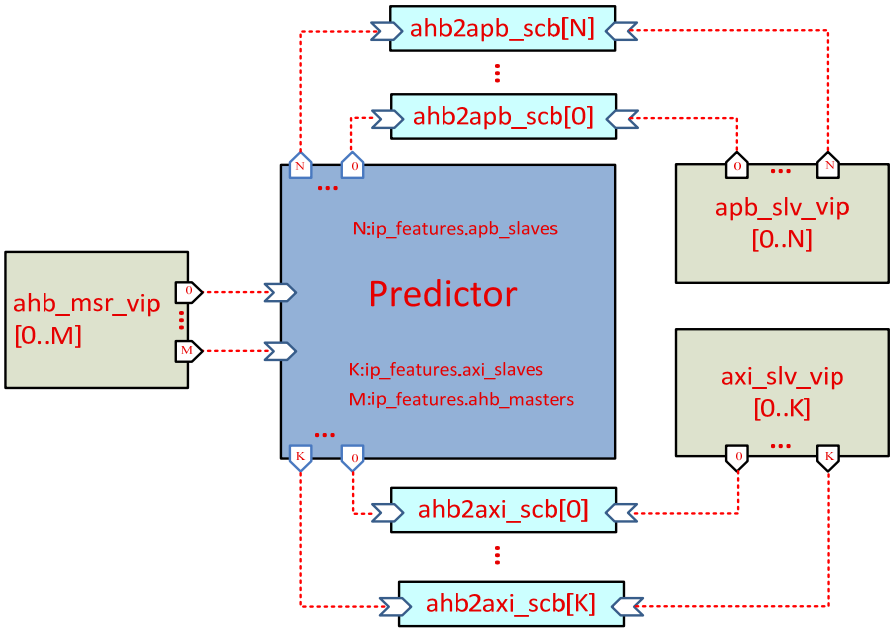


Figure 4 – predictor and scoreboard structure

4.3 Self-adapting basic sequence

The basic sequence provides stimulus. This paper's proposed solution uses IP design parameters to control sequence creation and start.

For instance, when IP design changes the base address of an AXI slave device, the AXI address stimulus is updated automatically.

```
for(k=0;k<cfg.ip_features.axi_slaves;k++) begin
`uvm_rand_send_with (read_tran,
{ read_tran.xact_type    == svt_axi_transaction::READ;
  read_tran.addr == cfg.ip_features.axi_slv_addr[k];}
```

4.4 Self-adapting functional coverage

Synopsys AMBA SVT VIP provides internal AXI, AHB and APB functional coverage; user just need enable them in the VIP configuration class.

Synopsys AMBA SVT VIP also provides coverage callback for IP design self coverage. In order to use IP design parameters to control functional coverage, this paper extends the Synopsys coverage callback class, instantiate `ip_component_features` in Synopsys coverage callback class, and uses IP features to configure the cover point, so the cover point is self-adapting:

```
wr_max = ip_features.axi_slv0_outstanding_wr;
covergroup cg_outstanding @(cov_outstanding_event);
cp_wr_num : coverpoint num_write_outstanding_xact
              { bins nums[] = {[0:wr_max]};}
endgroup
```

Be careful to use `ip_features` in functional coverage, since functional coverage should always be from the design specification, but not from IP design parameters code directly.

5. Reuse Self-adapting Parameterized IP Verification Environment

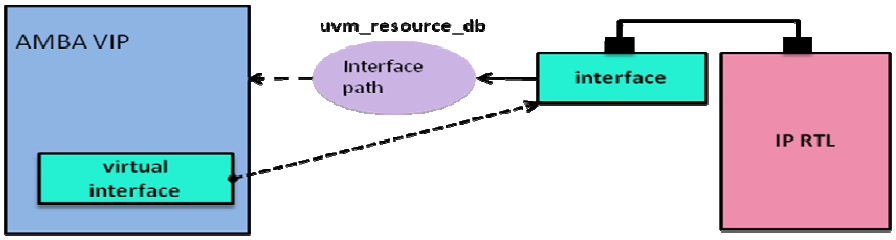
5.1 Vertical reuse

Vertical reuse means reuse from IP level to SoC level.

a) AMBA SVT VIP reuse.
AMBA SVT VIP consists of driver agent, monitor and protocol checker; at IP level, all the three works in active mode. When AMBA SVT VIP is reused to SoC level, driver agent is configured to work in passive mode, it doesn't generate master request or slave response stimulus to drive RTL signals anymore; monitor and protocol checker works in active mode.

b) UVM sequence reuse.
Regarding sequence, the normal sequence is hard to reuse, for it is adhere to interface protocol; Register model sequence can be directly reuse from IP level to SoC level, the key point is that implement register adapters for different protocols.

c) Connection reuse.
The Self-adapting parameterized IP verification environment deal with connection in three aspects:
First, how to connect the static RTL port signal to the dynamic AMBA VIP?
We instantiate real AMBA interface in static RTL world, declare virtual AMBA interface in AMBA VIP, and use global uvm_resource_db to connect the real AMBA interface to virtual AMBA interface like below.



Second, how to directly reuse the connection from IP level to SoC level?
We bind the real interface instance to RTL module, when IP RTL is reused to SoC level, module usually remains the same at IP and SoC level, so the connection bind can be directly reused from IP level to SoC level; Furthermore, bind method separate verification code from RTL code, we get clean RTL code with any EDA tool check.

Third, how to let the connection self-adapting to port signal changes?
As section 3 description, the port signals can be changed when design parameters changes.

5.2 Horizontal reuse

Horizontal reuse means reuse from one project to another project.
The whole IP Module UVC can be directly reused to different projects since it is self-adapting to IP design parameters.

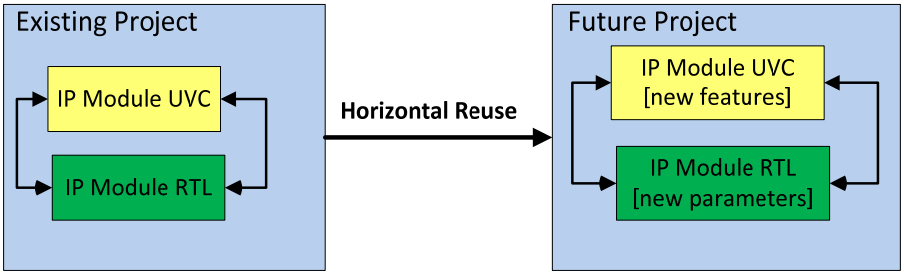


Figure 5 – IP Module UVC horizontal reuse

Moreover, in a new project, user can extend the AMBA SVT VIP build-in packet, modify some fields according to new project requirement, and employ UVM factory API set_type_override_by_type() or set_inst_override_by_type () to replace the old packet during simulation time. It is a convenient method for data packet horizontal reuse.

6. Results

This paper describes a highly configurable UVM environment with AMBA VIP. In this approach, the device number, memory range, address, data bus width and port names are all IP design parameters, which can be changed in different projects. The whole UVM verification environment is self-adapting to those changes.

Thus, when the IP design parameters change, it requires only minimal effort to update the verification environment. Sequences and drivers issue proper stimulus automatically. Module UVC creates correct components according to the new parameters directive automatically. Predictor accommodates its functions to the new parameters. Test bench connections automatically add or remove ports. Functional coverage and assertions also are self-adapting to new IP design parameters.

The whole UVM verification environment supports vertical reuse and horizontal reuse well.

7. Conclusions

The approach improves parameterized IP verification efficiency, and improves flexibility in UVM component creation, UVM configuration, stimulus sequence constrains, scoreboard, predictors, assertions, cover groups and test bench connections. The project practice shows this approach reduces additional resource demand by 30% when the IP design parameters change.

8. References

[1] J. Bergeron. "Writing Testbenches using System Verilog."24-25.

[2] B. Ramirez, M. Horn."Parameters and OVM — Can’t They Just Get Along?"2,DvCon2011

[3] W. Yun, S. Zhang." Deploying Parameterized Interface with UVM"1,DvCon2013

[4] Synopsys. "Synopsys Discovery Verification IP for AXI UVM User Manual."pp.29-30.

[5] Accellera. "Universal Verification Methodology (UVM) 1.1 User’s Guide." pp.72-73.