Architecture Validation of VFP Control for the WiNC2R Platform

BY AKSHAY JOG

A Thesis submitted to the

Graduate School -- New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Prof. Predrag Spasojevic

and approved by

____

____

____

New Brunswick, New Jersey

October, 2010

ABSTRACT OF THE THESIS

Architecture Validation and Characterization of VFP for WiNC2R Platform

Akshay Jog

Thesis Director: Prof. Predrag Spasojevic

A Cognitive Radio processing requires intelligent transceiver which can be easily programmed and reconfigured dynamically to support multiple protocols. The Winlab Network Centric Cognitive Radio (WiNC2R) platform is based on the concept of Virtual Flow Pipelining Paradigm. WiNC2R can support per packet protocol adaption through the reconfiguration of function sequencing. Since WiNC2R platform can be programmed by adding additional functions in software, and flow sequencing reprogramming architecturally supported in hardware, it can easily support future protocols. The latest version of WiNC2R has advanced shared VFP control unit, cluster based SoC architecture with all the processing engines in an 802.11a like OFDM transmitter flow.

It is very important to characterize the VFP overhead with the realistic protocol processing examples to understand the performance and cost penalties of added flexibility, and establish the base for the comparison with Software Defined Radio approach. The performance analysis of the VFP will give detailed insight about the various latencies involved in the VFP processing. VFP Architecture is validated to see that the current implementation does meet the requirements of the

WiNC2R platform. This performance analysis will help in characterizing VFP overhead under varying throughput requirements. Architectural validation of VFP will characterize certain parameters of the system programming, like reschedule period, guard time, etc.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisors, Prof. Predrag Spasojevic and Prof. Zoran Milijanic for their constant support, guidance and concern. Despite all their commitments and responsibilities they always gave me time and guided me at every step of my thesis.

I would also like to thank my Parents for their constant support. I would like to thank my team comprising of Onkar Sarode, Madhura Joshi, Mohit Wani and Khanh Le for their support at every point of my thesis. Last but not the least, thanks to Mohnish Kulkarni, Akshay Shetty, Ronak Daya.

Table of Contents

Table of Figures

1. Introduction of WiNC2R Platform

The Winlab Network Centric Cognitive Radio (WiNC2R) is Cognitive radio platform (1). WiNC2R is also a software programmable platform which can provide necessary flexibility in hardware for various protocols. The WiNC2R platform is based on the concept of Virtual Flow Pipelining Paradigm (2). WiNC2R can support per packet protocol adaption through the reconfiguration of function sequencing. Since WiNC2R platform can be programmed by adding additional functions in software, and flow sequencing reprogramming architecturally supported in hardware, it can easily support future protocols. The latest version of WiNC2R has advanced shared VFP control unit, cluster based SoC architecture with all the processing engines in an 802.11a like OFDM transmitter flow.

The traditional hardware pipelined system has a fixed set of operation, fixed operations at each stage of the chosen operating mode and fixed timing of operation which is end to end processing latency. Multiplexing of functional units is also not possible in traditional pipeline based system. VFP processing allows flexibility with respect to each design dimension described above. VFP processing also allows software defined functions to be incorporated into the VFP based program control framework (2). Since the system is not hardwired system, where block1 gives output to block2 and block2 gives output to block3 and so on, WiNC2R provides flexibility in hardware so that processing engine1 can give its output to processing engine4 as per the requirement. This provides flexibility for transferring data from any processing engine to any other processing engine unlike hardwired system. This flexibility allows flow of traffic to be configured at run time.

Figure 1 shows the current WiNC2R platform implementation. As shown in the Figure 1, the WiNC2R platform is a cluster based SoC having AMBA AXI bus as main system interconnect (3). Every cluster is connected to main system interconnect via AXI bus. Each cluster consists of many functional units and a VFP controller. Every functional unit has a single processing engine. Current implementation of WiNC2R has processing engines (PEs) like Header, Modulator, IFFT, scrambler etc to support 802.11a like OFDM transmitter flow. All functional units communicate with each other, over the local AXI bus. All the data

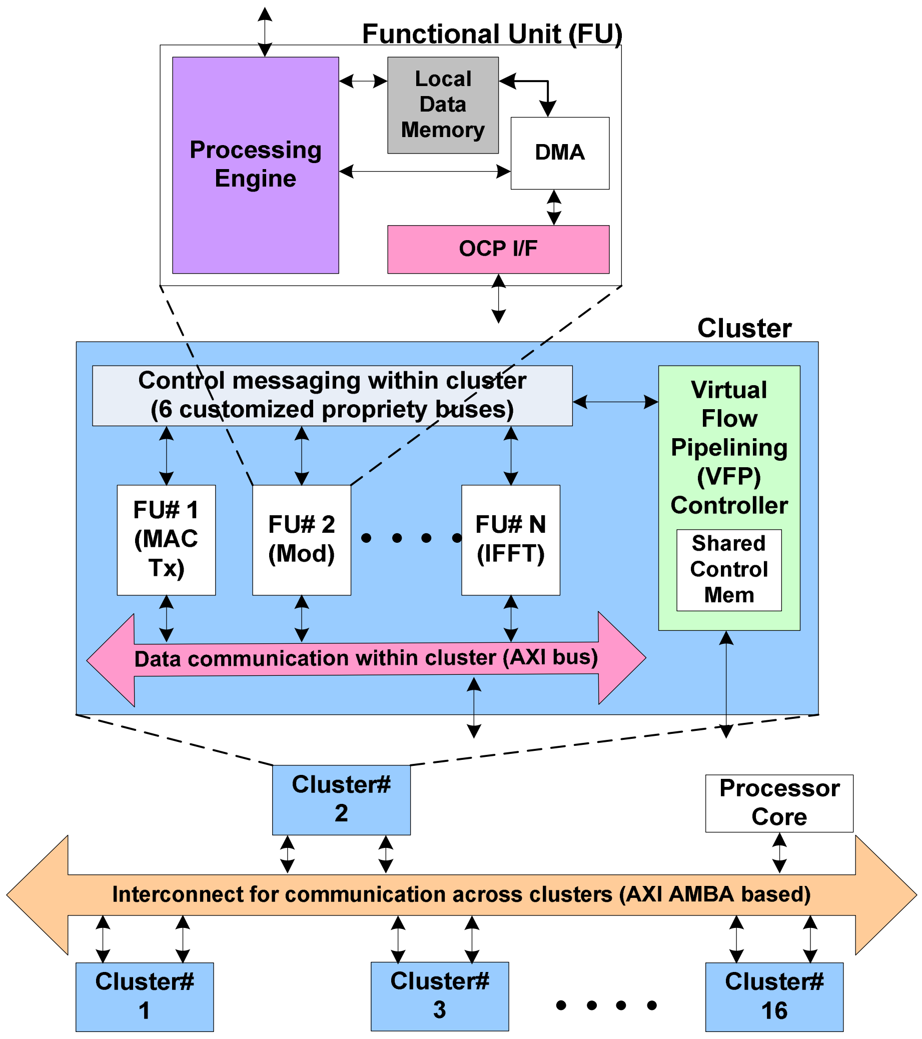
Functional Unit (FU)
Processing Engine
Local Data Memory
DMA
OCP I/F
Cluster
Control messaging within cluster (6 customized propriety buses)
Virtual Flow Pipelining (VFP) Controller
Shared Control Mem
FU# 1 (MAC Tx)
FU# 2 (Mod)
FU# N (IFFT)
Data communication within cluster (AXI bus)
Cluster# 2
Processor Core
Interconnect for communication across clusters (AXI AMBA based)
Cluster# 1
Cluster# 3
Cluster# 16

Figure 1: WiNC2R Platform (4)

transfers between functional units occur on AXI bus. VFP controller is also connected to local AXI bus. All the communication between VPF and functional units (FUs) for control messaging occurs on local bus as shown in Figure 1. As shown in Figure 1, functional unit consists of PE, local data memory, Direct Memory Access (DMA) engine and interface to access the local and

AXI bus. The local data memory present is FU is used by PE for processing the data and for storing the processed data. The data to be processed by PE is placed in Input Buffer, and processed data is stored in Output Buffer. DMA engine is responsible for transferring the data from Producer Output Buffer to consumer Input Buffer.

VFP controller is responsible for following tasks

1. Dynamic Scheduling of Tasks
2. Task Activation
3. Next Task Processing
   I. Consumer Identification
   II. Data transfer initiation between producer and consumer

Consider one producer PE has finished its processing and it has stored the processed data in its Output Buffer. This producer PE then sends a message to VFP controller indicating he has finished working on the data and based on the flow, the VFP should initiate a data transfer between producer and consumer. In order to determine which consumer to activate the VFP accesses Next Task (NT) table. When producer PE sends a message to VFP to initiate a data transfer, producer PE sends an NT table pointer. This NT table pointer decides the particular consumer which can change as per the flow. By accessing NT table VFP sends a control message to consumer DMA engine to initiate data transfer between producer and consumer. After DMA transfer is done, VFP does flow graph dependency checks by accessing Global Task Table (GTT). GTT has all the necessary information related to a particular task for a particular consumer. Once the dependency checks are done, VPF activates a task for consumer. Every consumer has a Task Descriptor (TD) table. TD table has all the necessary information related to the activated task. Essentially TD table has information related to the data size to be processed by

PE, the starting address of the Data stored in Input Buffer for processing and the Output buffer pointers, to store the processed data. GTT and TD table has descriptors related to all tasks in the system. NT table has descriptors related to the next task processing. GTT, TD and NT table are backbone of the WiNC2R platform since they have necessary information about the flow graph dependency, next task processing and about the current task to be processed by PE.

Considering the complexity of the system, it is very important to characterize the VFP overhead with the realistic protocol processing examples to understand the performance and cost penalties of added flexibility, and establish the base for the comparison with Software Defined Radio approach. The performance analysis of the VFP will give detailed insight about the various latencies involved in the VFP processing. VFP Architecture is validated to see that the current implementation does meet the requirements of the WiNC2R platform. In order to validate and extract performance from the WiNC2R platform, very powerful verification/ validation environment is needed. In the later chapters, it will be shown that how Open Verification Methodology (OVM) based verification environment can be used to validate the architecture and to extract performance from WiNC2R (5).

2. Overview of Testbench

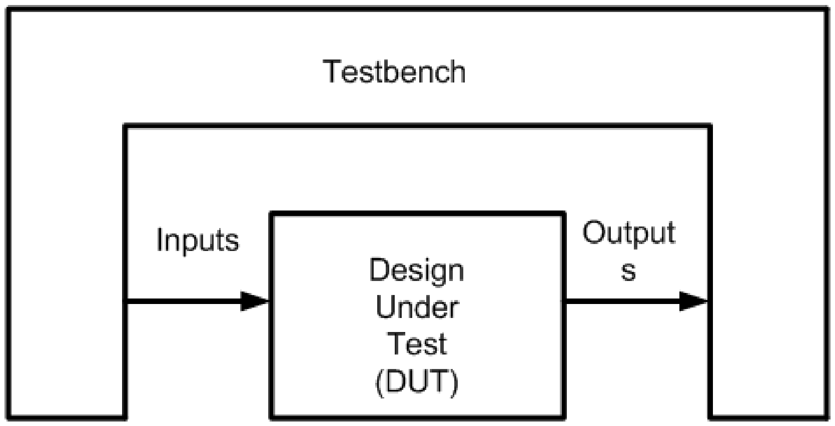

Figure 2: Generic Testbench (6)

Generic testbench wraps around the Design Under Test (DUT) as shown in the Figure 1. The testbench has to work over wide range of the levels of abstraction, sequences and transaction to verify the DUT under various scenarios. The basic testbench functionality is as follows

1. Generate Stimulus
2. Apply stimulus to the DUT
3. Capture the response
4. Check for correctness
5. Measure progress against the overall verification goals

Testbench consists of various Bus Functional Models (BFM). For DUT, BFMs are real component, but these BFMs are part of the testbench. Consider the Advance Microcontroller Bus Architecture (AMBA) bus as DUT (3). To verify the functionality of the AMBA bus, real components connected to bus are not required. BFMs will be used in place of the real components which will comply with AMBA protocol. BFMs will be designed to meet the functionality of the real component. BFMs are not required to be synthesizable unless prototyping is done on FPGA or emulations (6).

2.1 Flat Testbench

Consider DUT is a generic bus. A basic testbench to verify the generic bus will look like following

```
module test_AMBA (addr, write, read, data, rst, clk, grant, req);

//port declarations and wiring ……..

initial begin

rst = 0; // initializing the reset to zero

clk=0; // initializing the clock to zero

#40 rst=1; // assigning reset to 1 , this will pull out the DUT from reset

#20;  // wait period of 20 time units

//To write on Bus, address , data, control bits needs to be enabled or driven

addr=32'h40;

write=1;

data=32'h100;

req=1; // driving request to one

wait(grant==1)  // waiting on grant from bus

………
```

```
end  // end of the initial begin

always

#10 clk= ~ clk;  // clock generation

endmodule
```

The Advantages of Flat Testbenches are

1. Easy to write
2. Rapid Development of basic testbench

Disadvantages of Flat Testbench

1. Without grouping of similar transaction into task and functions, the Testbench is not reusable
2. All the testcases are written manually implying more probability of error
3. Only limited testcases can be executed which will not cover wide range of the input combinations
4. Constrained-random stimuli are not generated in the flat Testbenches. Constrained-random stimuli help in finding the bugs at a faster rate
5. Process of verification using flat Testbenches, converges slowly in terms of meeting verification goals. This can affect the product launch, implying loses to the company, considering the competition in the market
6. No automation is provided to check the correctness of the results and to generate stimuli

The disadvantages of the flat testbench, dictates the need of the layered testbenches.

2.2 Layered Testbench

Layered testbench reduces the complexity of the verification process to the manageable pieces. Transactors provide a useful pattern for building these pieces. With appropriate planning, testbench infrastructure can be built which will be shared by all tests and does not have to be continually modified as per the tests. Building the layered testbench will take longer time than flat testbenches, but the paybacks are very high in long run. Due to layered approach, the designing of testbench can be broken down to different layers which can be designed by different teams simultaneously. Different layers are also broken down to different blocks (Transactors) to make reusable, self contained blocks. Figure 3 shows the layered testbench.

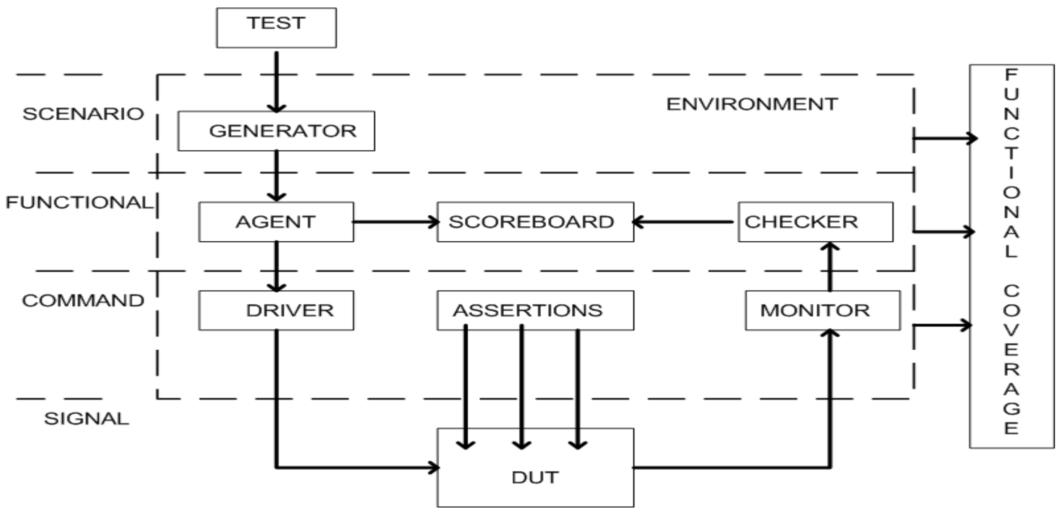

Figure 3: Layered Tesetbnech (6)

Signal layer contains the DUT and the signals connecting it to the testbench. The command layer is next layer. The DUT’s inputs are driven by driver which is a command and the responses or the output is captured by monitor. Assertions also cross the command/signal layer as they look at individual signals but look for changes across an entire command. The command layer is specific to the protocol. For example if the DUT is AMBA bus or System connected to AMBA bus, then the driver need to comply AMBA protocol and monitor halso has to follow AMBA protocol in order to monitor the correct transactions. Considering the AMBA based system, the command in this case is bus read or bus write. The functional layer feeds the command layer. The agent block receives the higher level transaction such as DMA read or DMA write and breaks them into individual commands like bus read or bus write. Agent also gives these commands to scoreboard that predicts the results of the transaction. The checker then compares the commands from monitor with those in scoreboard. The Scenario layer feeds the functional layer. As the name suggests, the function of the scenario layer is to create the scenario for DUT so as to stress the DUT to the boundaries. Consider MP3 player as a DUT which can concurrently play music stored into memory, download new songs, respond to user input like play next song, increase/ decrease volume etc. Consider downloading a music file from internet, involves reading control registers, writing to memory for setting up the download, multiple DMA write transfers for song etc. This is the scenario while downloading a song. Like this various scenarios needs to be executed to verify the DUT. The scenario layer uses constrained-random values for scenario generation. The blocks in each layer are written once during the development for verification infrastructure. During verification process, the functionality is added to these blocks as per the requirement, but the blocks do not change as per the test. The hooks are written in the code so as to change the behavior of the block as per the test without rewriting the block again. The test layer contains the constraints to create the stimulus. The test orchestrates the various scenarios. Functional coverage measures the progress of all the tests in accordance with the verification plan. As per the project requirements, functional coverage criteria changes. Because of varying nature of functional coverage criteria, the functional coverage is not part of layered testbench. Even to run direct tests, the layered testbench or the verification environment does not have to change. Direct tests are written to explore the bugs which were not activated when constrained random stimulus were used. Due to constrained random stimulus functional coverage may saturate to a fixed value. After this point direct testing is required to achieve 100% functional coverage. Functional coverage is a measure of which design features have been exercised by the tests. 100% functional coverage implies all the features of the DUT are verified correctly (6).

If constrained-random stimuli are used, fewer tests need to be written. But with direct testing, verification engineer has to write thousands and thousands of tests. Due to layered testbench, the lower layers that is command, signal, functional remains generic implying reuse of the infrastructure. All the testcases are generated automatically unlike flat testbench. Testcases can be executed automatically for all days of the week and all months of the year. Also due to automatic checking capability of the layered testbench, testcases can be executed for a longer time without human intervention. The complexity of the modern devices dictates the need for an automated, systematic, efficient testbench environment to fix the bugs as fast as possible. To make efficient use of the layered testbench, methodology is required. Selection of appropriate methodology is crucial for the success of the product.

To verify earlier implementation of WiNC2R, Xilinx Bus Functional Model was used to simulate the system. Earlier WiNC2R platform was based on Processor Local Bus (PLB) bus which required using BFMs supporting PLB protocol (7). The Xilinx BFM is like a flat testbench and posses all the disadvantages of the flat testbench. There is need of an automated testbench to verify WiNC2R which can gather data from hardware component and use this feedback to modify the tests being executed (8). BFMs are part of the layered testbench, but they cannot serve the purpose of the entire layered testbench. As shown in figure 2, the driver and monitor are BFMs. But just the driver and monitor does not suffice the purpose of the complex requirements of the verification process. BFMs are in command layer as shown in figure 2. These BFMs are protocol specific as mentioned earlier. So Xilinx BFMs were just following the PLB protocol. Current implementation of WiNC2R is based on AMBA AXI bus requiring new BFMs supporting AMBA AXI protocol (3). AMBA AXI bus was chosen for the current WiNC2R architecture, because of the burst constraint of PLB bus. PLB bus allowed only 64 bytes of data transfer in one burst transaction. In case of bytes more than 64, several burst transactions were required which was creating a bottleneck for system performance. In AXI burst length can go up to 1024 byte

(9). Hence to generate constrained-random stimulus, to gather functional coverage information, to have one common testbench for all tests and to keep test code specific separate from testbench, Layered testbench with appropriate methodology is required.

2.3 Verification Techniques

Verification is a process used to demonstrate the functional correctness of a design (10). Following are three categories in functional verification (11).

1. Formal Verification
2. Simulation-based Verification
3. Acceleration/ Emulation-based Verification

2.3.1 Formal Verification

Formal Verification uses logical and mathematical formulas and approaches to prove or disprove a given property of a hardware implementation. Formal verification operates on equations describing the system and not on test vectors. Any property proved by a formal verification tool holds for all possible test vectors applied to that behavior. Formal verification does not require test vectors to be applied. Also formal verification techniques are able to make universal statements about a property of a design implementation holding for all possible inputs. This technique is useful when the testbench and test vectors are not yet available.

2.3.2 Simulation-Based Verification

As the name suggests, simulations are done for functional verification. The verification environment consists of a testbench and a design. The DUT is put into known current state and based on the output of the current state, DUT is put into next state. The output of current state is checked with the expected outputs. This technique involves the process of consecutively taking the design through different states where the sequence of observed design states corresponds to a verification scenario listed in verification plan.

2.3.3 Acceleration/Emulation-based Verification

Formal and simulation-based verification techniques provide many benefits in the early to middle stages of the design flow. The speed of those techniques falls short, when the entire system along with its application software needs to be verified. In this scenario, the simulation needs to run for millions and billions of instructions so that the macro behavior of the application software can be verified. In this technique design is mapped to configurable platform like FPGA. Since the design is mapped to FPGA, the instructions can run at the desired clock rate of the design, making the execution time shorter. In hardware acceleration, the testbench program is running on host computer. In hardware emulation, stimuli are applied via real world interfaces and verification in general is restricted to monitoring the input and outputs of the DUT. In the hardware acceleration the acceleration is limited by runtime of the testbench on the host computer and the speed of the communication channel between host computer and acceleration platform.

To verify current implementation of the WiNC2R, simulation-based verification technique has been chosen. Most of the processing engines in the current WiNC2R are written in C/ C++ code. These processing engines are not synthesizable. The reason for choosing most of the processing engines in C/ C++ is to evaluate the current architecture for performance, assuming the standard delays involved with the processing engines, so as to check the compliance with 802.11a standard.

2.4 Verification Methodology Selection

Recent statistics show that 60-70 % of the entire product cycle for a complex logic chips is dedicated to the verification tasks. Verification of complex functions that can be built using new tools poses a challenge to reduce the total product time. Designing at a higher abstraction level allows the designers to build highly complex functions with ease. This increase in the design complexity doubles the verification effort. The increasing size and complexity of designs and shortened time to market window means verification engineers need to verify the more complex and larger designs in a shorter time frame than previous projects (12) (11). Due to such stringent requirement, challenges in verification process increases. Hence selection of verification methodology is very crucial in the success of the product. Efficiency, reusability, and productivity are of at most importance in the verification process. There are many verification methodologies available in the market. Verification Methodology is categorized in to Assertion- Based Verification, Coverage Driven Verification and Metric-Driven Verification. Out these three options, coverage driven verification is chosen. Coverage Driven verification methodology brings the following concepts and approaches.

1. Transaction Driven Verification
2. Constrained random Stimulus generation
3. Automatic Result Checking
4. Coverage Collection
5. Directed-test- based verification

Layered testbench will provide the necessary infrastructure for the coverage driven methodology. Also for the verification of WiNC2R coverage driven verification methodology

Following are different criteria to choose verification methodology (12) (13).

1. Identifying verification goals which will be catered by the Verification methodology
2. Evaluate the effect of adapting new verification methodology in terms of tools and learning curve of the team
3. Direct and indirect effect of new verification methodology on cost and time to market

4. Available support for new verification methodology
5. Interoperability with existing in-house tools and methodology and Interoperability with existing Verification IPs (VIP) and new methodology
6. Effectiveness of the new methodology in case of various products
7. Reusability of infrastructure across different projects
8. Ability to quickly provide necessary infrastructure for the verification, considering the verification goals and time to market
9. Modularity of the Verification Methodology
10. Scalability of the Verification Methodology
11. Flexibility of the Verification Methodology
12. Predictability of the Verification Methodology

Advance Verification Methodology (AVM) by Mentor Graphics, Universal Reuse Methodology (URM) by Cadence, Verification Methodology Manual (VMM) (14) by Synopsys and Open Verification Methodology (OVM) (15)by Cadence and Mentor Graphics were available methodologies in the market. OVM was available for download from January 2008. Since OVM was joint development between Cadence and Mentor Graphics, options to choose methodology, narrowed down to VMM and OVM. Among OVM and VMM, OVM was chosen as verification methodology to verify WiNC2R. OVM is coverage driven verification methodology which will help in building the layered testbench. Following are the key aspects of OVM (16).

1. Open
    - 1.1 Written in IEEE 1800 SystemVerilog
    - 1.2 Runs on any simulator supporting the IEEE 1800 standard
    - 1.3 Verified on Cadence's Incisive and Mentor Graphics' Questa Verification Platform
    - 1.4 True open-source license agreement (Apache 2.0)

2. Interoperable

    2.1 Ensures VIP interoperability across ecosystem & simulators

    2.2 Enables VIP ‘plug and play’ functionality for designers

    2.3 Ensures interoperability with other high level languages

3. Proven

    3.1 Based on Cadence’s Incisive Plan-to-Closure Methodology - URM Component and Mentor’s Advanced Verification Methodology (AVM)

    3.2 Incorporates Best Practices from >10 years of experiences

VMM Initially was not open. VMM has many flaws compared to OVM (13). VMM is based on old technology. OVM takes full advantage of SystemVerilog and Object Oriented Programming (OOP). VMM 1.1 included many features borrowed from OVM. Many important features of OVM like Transaction Level Modeling (TLM), Factory, set/get_config methods, automated phasing are not there in VMM. In terms of building and configuring the verification environment, VMM is not as Flexible as OVM. OVM environments are scalable whereas in VMM only one environment can be built. More detailed difference between OVM and VMM are given by Tom Fitzpatrick (13). Hence OVM is the best choice for the validation/verification methodology.

3. Overview of OVM

OVM is an open source, SystemVerilog based class library developed to quickly build object-oriented verification environment. Due to availability of the predefined classes for building verification environment and writing tests allows verification engineers to meet their verification goals sooner with high confidence. The OVM class library objects and classes are defined to implement multi-layered verification environment based on coverage driven methodology. OVM class library features can be categorized as follows

1. Creating and managing class objects in the verification environment
2. Building and configuring the verification environment hierarchy and managing the simulation runtime phases
3. Use of Transaction Level Modeling (TLM) for connecting verification environment blocks
4. Generating transaction sequence for verification scenarios
5. Provide built in checking support
6. Provide facility for reporting and messaging (11) (17)

Following are various verification components used in building verification environment using OVM (11) (17)

Driver: Driver is a verification component which connects to Design Under Test (DUT) via interface. It has transaction level interface to communicate with other transaction level component in verification environment. Driver is just responsible for driving the transactions to DUT. Sequence generation is not done in driver. Driver has to follow a particular protocol to drive the transactions to DUT. Driver receives the transaction on transaction level interface from sequence generator and then by following the protocol driver drives the transaction to DUT. This makes the driver reusable for later projects provided it has to follow same protocol.

Monitor: Monitor is a verification component responsible for extracting signal information at the interface level and translating it into events, data and status information. Coverage and basic protocol checking is also done in Monitor. Monitor can broadcast the information received from DUT to other verification components using TLM which can also act as a feedback in sequence generation.

Sequence: Sequences generate the data items and other sequences (subsequences) which are sent via Driver to DUT. Constrained random stimulus generation is done in sequences. Various complex scenarios can be generated using the sequences.

Sequence Library: A collection of sequences used by sequencer

Virtual Sequence: Any sequence that co-ordinates the activities of other sequences in one or more sequencers is called Virtual Sequence. Virtual Sequences enable centralized data flow control on multiple interfaces.

Sequencer: Sequencer is a verification component that mediates the generation and flow of data between sequences and driver.

Virtual Sequencer: Virtual Sequencer allows a single sequence to interact with multiple sequencers and hence interact with multiple drivers.

Sequence Item: Sequence Item is the transaction generated by sequence, based on the constraints given to sequence item.

OVC: OVM Verification Component is an encapsulated, reusable and configurable verification component for an interface protocol, a design sub-module or a full system.

Bus monitor: Bus monitor is verification component responsible for extracting signal information at bus level and translating it into events, data and status information

Agent: Agent encapsulates driver, monitor, and sequencer. An agent is capable of independent operation. In a verification environment there can be many agents with different behavior. Behavior of each agent is configurable. Consider a case of multiport router verification. In this case there can be single agent per port of the router. Since the behavior of each agent is configurable, this gives the opportunity to generate real world scenario.

Environment: Environment is top level component of the OVC. Environment can contain one or more agents and top level component such as bus monitor. The environment is also configurable. In a verification environment many environments can exist at the same time each with configurable behavior. Environment is useful in reuse of the verification environment. Environment can configure underlying agent for particular test scenario.

Scoreboard: Scoreboard is a verification component responsible for checking the correctness of the transactions received from DUT. Scoreboard has transaction level interface to communicate with the monitor.

Testbench: Testbench can contain various environments, scoreboards. Testbench can configure the each environment.

Test: Test encapsulates the test specific instructions from the test writer. Test can configure testbench as per the test scenario.

TLM: Transaction Level Modeling interfaces provide a standard method for components to exchange transactions instead of signals. TLM focuses on the transaction and not on the implementation of the components using it.

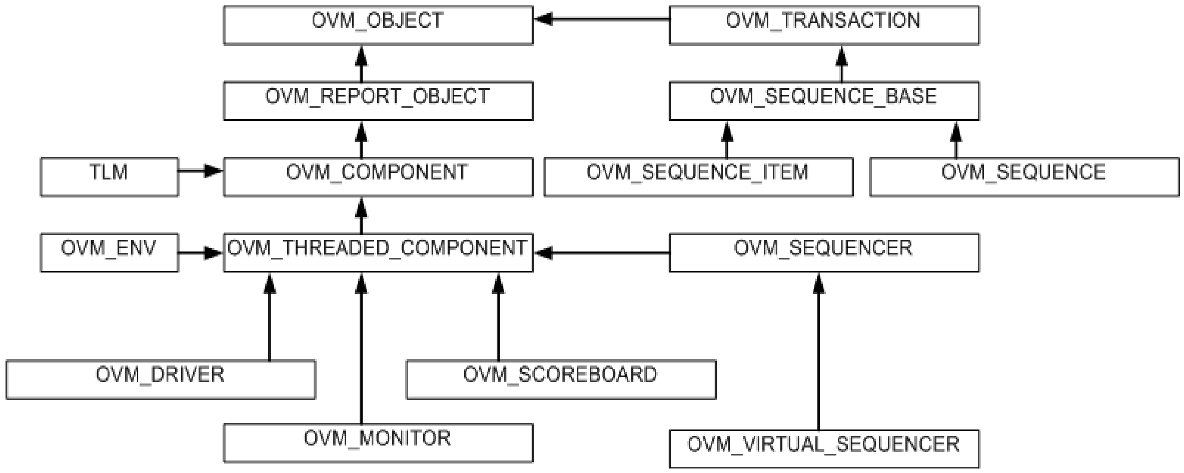

Figure 4: OVM Class Hierarchy (5)

Figure 3 shows the Unified Modeling Language (UML) diagram of OVM class library. Using these predefined classes, OVM based verification environment is built. OVM classes help to build hierarchical class based verification environment which makes it layered testbench. OVM gives the facility to generate constrained random stimulus, to apply the stimulus via driver, to capture the responses via monitor, to check the correctness via scoreboard and to measure the overall progress against the overall verification goals via coverage. The OVM class library and methodology provides all the technology need to implement the reusable constrained random, coverage driven layered testbench. Use of TLM communication as the underlying foundation for connecting verification components facilitates reusability and modularity. OVM based verification environment can be modified on the fly and multiple tests are written from the same base environment with minimal code changes. OVM provides common configuration interface so that all components can be customized on per_type or per_instance basis without changing the underlying code. OVM also provides common message reporting and formatting interface (5). In SystemVerilog simulation, time is advanced without any consideration of abstract phases that may exist in verification flow. Progression of time in verification environment is however, managed in phases where different sets of activities take place in each phase. OVM defines such simulation phases. OVM class library provides following built in simulation phase methods (17).

1. build ( )
2. connect ( )
3. end_of_elaboration ( )
4. start_of_simulation ( )
5. run ( )
6. extract ( )
7. check ( )
8. report ( )

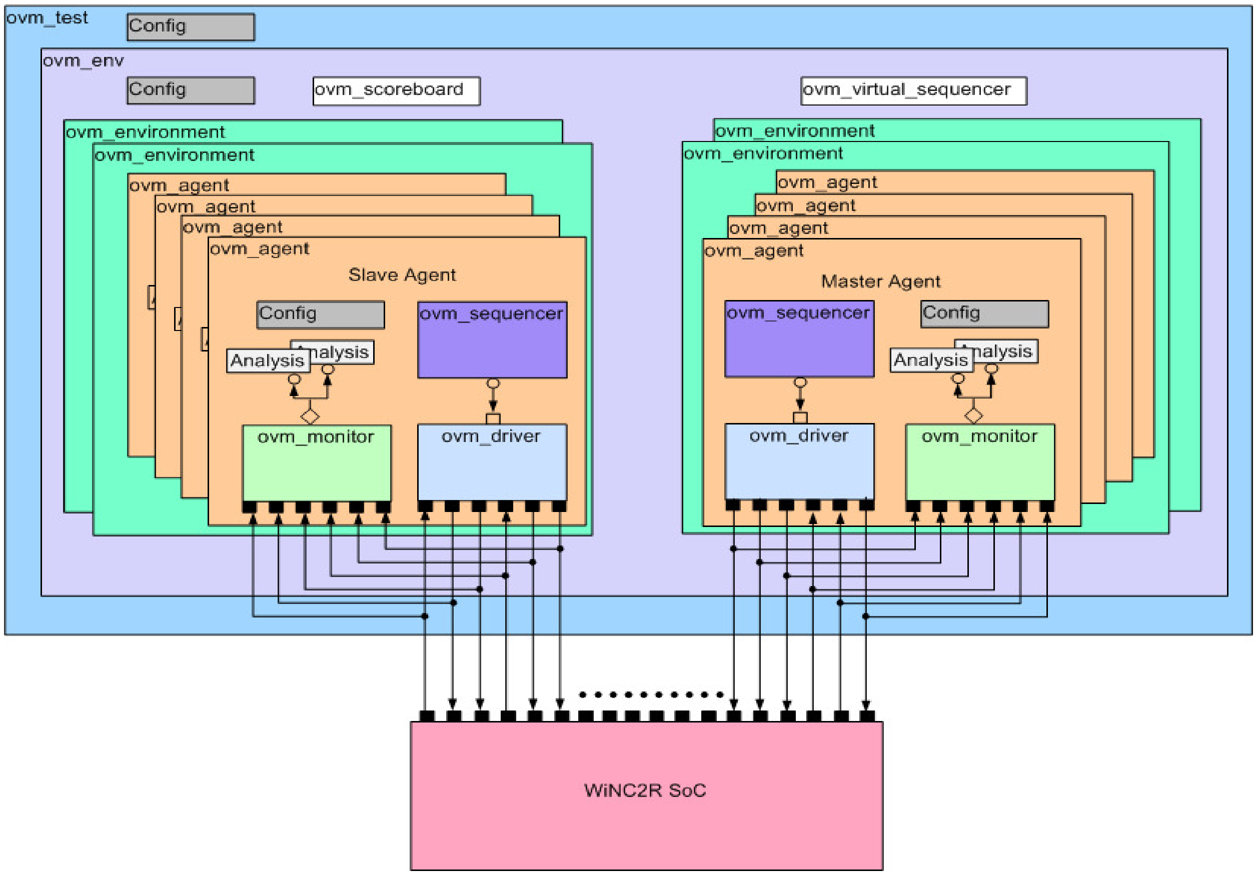

Figure 5: OVM Based Verification Environment (11)

1. build ( )

build ( ) is the first phase called automatically for all components in a top-down fashion. Build is a function call and executes in zero time. Build method creates its components child components and optionally configure them. The top-down execution order allows each parent's build ( ) method to configure or otherwise control child parameters before the child component's build ( ) method is executed.Since the build is called in top down fashion, to make sure that build is called only once, every build call has a super.build ( ). Following is sample SystemVerilog (SV) code

```
Class child1 extends ovm_component;
```

```
  ……..
     virtual void function build( );
       super.build ( );
     //get the configuration information
     //create child compoenent
     //configure child component
     endfunction
     ……
endclass
```

As shown in the Figure 4, test level class will call the build function of the env level class and then finally build of the driver, monitor, sequencer will be called.

2. connect ( )

The connect phase is executed after build ( ). connect ( ) is a function call which executes in zero time. connect ( ). connect ( ) phase makes the TLM connections between verification components in verification environment. Following is a sample SV code for connect ( ) phase.

```
Class child2 extends ovm_component;
……..
  virtual void connect( );
     if (is_active = = OVM_ACTIVE )
```

```
                driver.seq_item_port.connect (sequencer.seq_item_export);
        for(int i=0; I < = num_subscribers; i++)
                monitor.analysis_port.connect (subscr[ I ].analysis_export);
……..
    endfunction
……..
endclass
```

Above code makes first connection between sequencer and driver so that transactions can reach from sequence to driver via sequencer. Second connection is made between monitor and subscriber component (it can be scoreboard) in a loop.

3. end_of_elaboration ( )

This phase allows final adjustments to the environment after build ( ) and connect ( ) phases are over. User can assume that the entire environment is built and connected. This phase is a function call and executed in zero time.

4. start_of_simulation ( )

start_of_simulation ( ) phase provides a convenient place to perform any pre-run activity like displaying banners, printing final topology and configuration information. This phase is function call and executes in zero time.

5. run ( )

run ( ) phase is only predefined time consuming phase unlike other phases. During this phase components primary run time functionality is executed. Since this phase is implemented as a task it can spawn various processes. This phase can also have function calls during the execution of task. When a component return from run task it does not assure that run phase is complete. Since run is task which can spawn many processes, there can be many processes forked during run. There needs to be a mechanism to stop or kill these processes.

- stop- when component's enable_stop_interrupt bit is set and global_stop_request is called, components stop task is called. This essentially allows completion of current transaction, flush queues, etc. after stop call is returned, kill is executed to kill any remaining processes.
  (stop is a method user can implement for safe and desired shut down of processes. global_stop_request is a function call made to kill all the current processes of all the components)
- kill- when kill is called all component's run processes are killed immediately. It is recommended that kill should not be called explicitly, instead use stop method of the component for safe shutdown.
- timeout- If a timeout is set, the phase can end at the timeout, provided the timeout came before kill or stop.

Example of run phase will be driver driving the current transaction to the DUT.

6. extract ( )

This phase is useful for extracting the simulation results before checks are done in next phase. This phase is function call which executes in zero time. Due to this phase, results from all the components can be gathered and then decision or feedback can be provided later. Following things can be done in extract phase ( )

- Collect the assertion-error count
- Collect the coverage information
- Extract internal signals and registers of DUT
- Extract statistics or other information from all component

7. check( )

After extraction of the vital information, checks can be applied to decide the progress of the simulation. This phase is a function call which will executes in zero time.

8. report ( )

As name suggests this phase is used for reporting the results into the files or on screen. This phase is also a function call which will execute in zero time. Apart from these phases OVM allows users to define additional phases. For detailed information on how to insert new phases to OVM and how to create a verification environment using OVM, refer to OVM User Guide (17).

4. OVM Configuration for WiNC2R Platform

Deciding the parameters for sequence item is the primary step in constrained random stimulus generation. These parameters depend upon the Input variables of the DUT. WiNC2R is an AMBA AXI based bus system. To apply input to the WiNC2R system, bus transaction needs to be done in order to write a descriptor in the queues (as explained in chapter 1). So parameters in sequence item will be the parameters required to initiate bus transaction on AXI bus. The width of these parameters will be depending upon the AXI bus configuration. Following table shows the various parameters required to initiate a write bus transaction on AXI bus. The width of these parameters is decided during AXI bus configuration (3).

Table 1: AXI Write Channel Signals

Parameter Name	Width (bits)	Input/ Output	Description
awid_m	4	Input	Write address ID. This signal is used as identification tag for write address group of signals. This signal belongs to Write Address Channel.
awaddr_m	32	Input	Write Address. The write address bs gives the address of the first transfer in a write burst transaction. The associated control signals are used to determine the address of the remaining transfers in busrt. This signal belongs to Write Address Channel.
awlen_m	8	Input	Burst Length. The burst length gives the exact number of the transfers in a burst. If burst length is equal to 8, this implies 256 data words can be transferred in a single burst transaction. This signal belongs to Write Address Channel.
awsize_m	3	Input	Burst Size. This signal indicates the size of each transfer in the burst transaction. This signal belongs to Write Address Channel.
awburst_m	2	Input	Burst Type. The burst type coupled with size information details how the address for each transfer will be computed within the burst. This signal belongs to Write Address Channel.
awlock_m			Lock Type. This signal provides additional information about the atomic characteristics of the transfer. This signal belongs to Write Address Channel.
awcache_m	4	Input	Cache Type. This signal indicates the bufferable, cacheable, write-through, write-back and allocates attributes of the transaction. This signal belongs to Write Address Channel.
awprot_m	3	Input	Protection Type. This signal indicates the normal, privileged or secure protection level of the transaction and whether the transaction is data access of an instruction access. This signal belongs to Write Address Channel.
awvalid_m	1		Write Address Valid. This signal indicates that the valid write

			address and control information is available. This signal belongs to Write Address Channel.
awready_m	1	output	Write Address ready. This signal indicates that the slave is ready to accept an address and associated control signals. 1 = Address and Control information available 0 = Address and Control information not available This signal belongs to Write Address Channel.
wid_m	4	Input	Write ID tag. This signal is the ID tag of the write data transfer. The wid, awid must match for same transaction. This signal belongs to Write Data Channel.
wdata_m	32	Input	Write data. The write data bus can be 8, 16, 32, 64, 128, 256, 512, 1024 bits wide. This signal belongs to Write Data Channel.
wstrb_m	4	Input	Write Strobe. This signal indicates which byte lanes to update in memory. This signal belongs to Write Data Channel.
wlast_m	1	Input	Write Last. This signal indicates the last transfer in a write burst transfer. This signal belongs to Write Data Channel.
wvalid_m	1	Input	Write Valid. This signal indicates that the write data and strobes are valid. 1 = Write data and strobes are available. 0 = Write data and strobes are not available. This signal belongs to Write Data Channel.
wready_m	1	Output	Write ready. This signal indicates that the slave can accept the write data. 1 = Slave ready. 0 = Slave not ready. This signal belongs to Write Data Channel.
bid_m	4	Output	Response ID. The identification tag of the write response. Bid, wid, awid must match for same write transaction. This signal belongs to Write Response Channel.
bready_m	1	Input	Response Ready. This signal indicates that the master is ready to accept the response information. 1 = Master is ready. 0 = Master is not ready. This signal belongs to Write Response Channel.
bresp_m	2	Output	Write Response. This signal indicates the status of the write transaction. This signal belongs to Write Response Channel.
bvalid_m	1	Output	Write Response Valid. This signal indicates that the valid write response is available. This signal belongs to Write Response Channel.
aclk	1	Input	Global Clock for AXI bus.
aresetn	1	Input	Global reset. This signal is active low.

AXI bus has five different channels as follows

1. Read Address Channel

2. Write Address Channel

3. Write Data Channel
4. Read Data Channel
5. Write Response Channel

Out of these five channels three channels namely Write Address Channel, Write Data Channel, Write Response Channel are used for write transaction on the AXI bus. Above table and Figure 5 shows details of only Write Channel signals. For detailed information on all channels and the encoding scheme for these parameters, refer to AMBA AXI Specification Document (3).

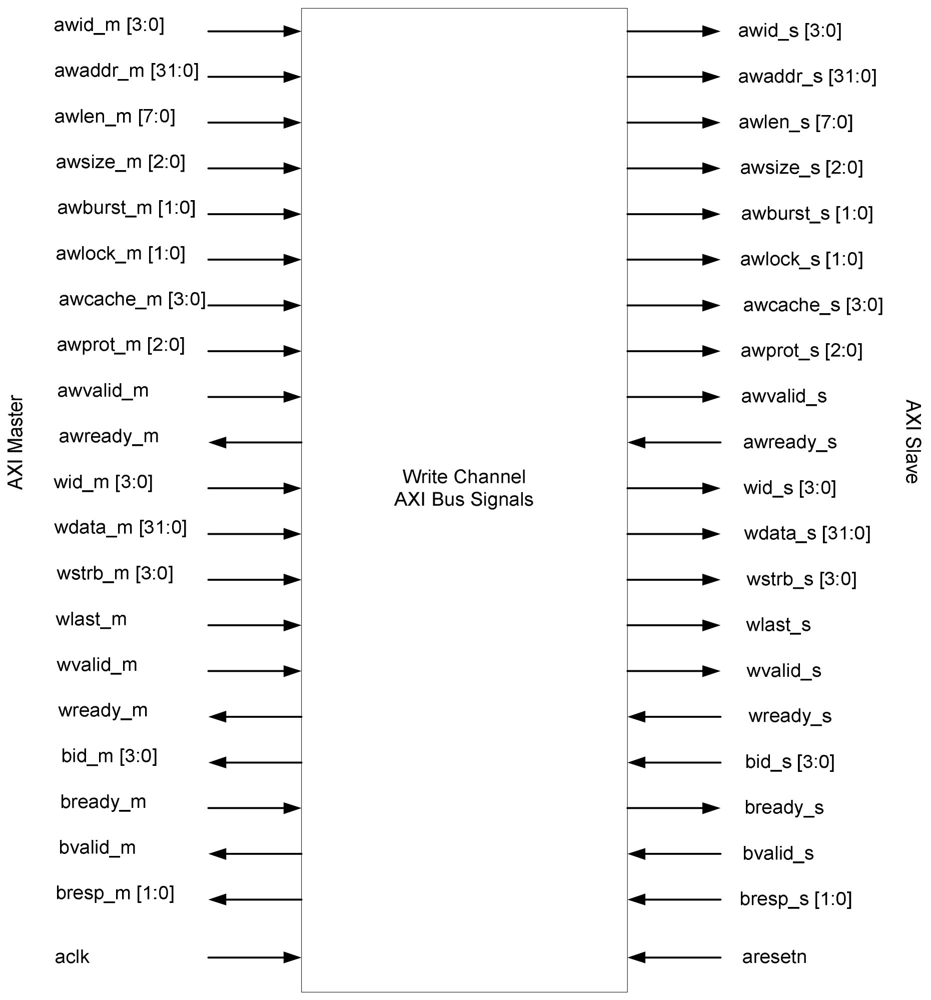

Figure 6: AXI Write Channel Signals

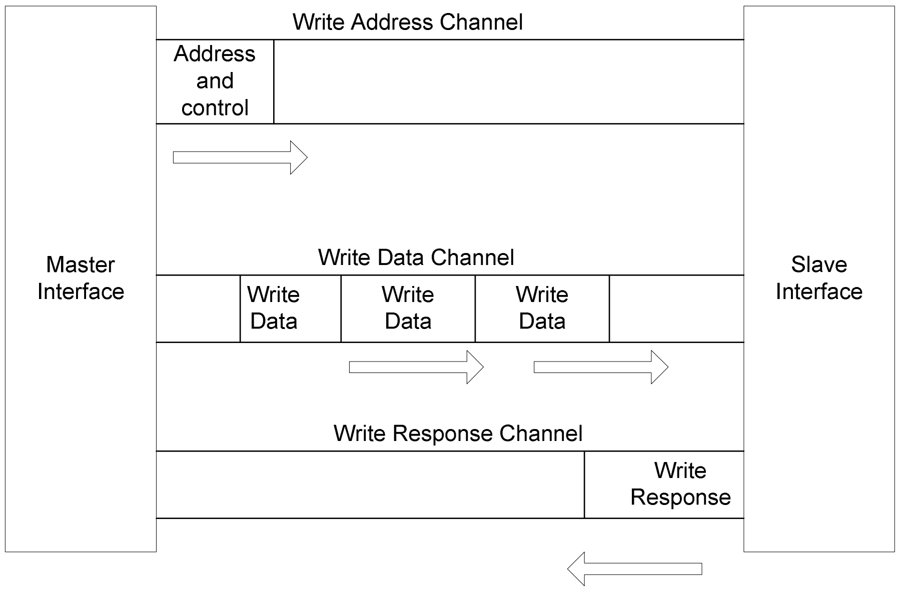

Figure 7: AXI Write Burst Transaction (3)

Figure 6 shows a write burst transaction on AXI bus. In order to initiate a write transaction on AXI bus, the master has to assert awvalid_m signal only when it drives the valid write address and control information (like awprot_m, awcache, awlen etc) about the write transaction(_m is appended to every signal to denote the master side signals and _s is used to denote the slave side signals). awvalid_m signal must remain high till slave asserts the awready_s signal. During write burst transaction then master drives wvalid_m and waits for wready_s from slave. Master is supposed to keep wvalid_m high till the entire burst transaction is complete. High wvalid_m validates the data on write_data. The master then asserts wlast_m signal, in order to indicate the last transaction from master. Once all the words have been transferred, master makes the wvalid_m low. Master needs to drive bready_m signal high to accept the responses from slave. After complete data transaction, slave asserts bvalid_s and drives correct responses on bresp_s. Slave drives responses per transaction by driving appropriate bid. For same transaction, bid, wid and awid must match. For detailed information on AXI protocol refer to AMBA AXI Specification Document (3).

s4.1 OVM Sequence Item for WiNC2R Platform

In order to do constrained random stimulus generation, various constraints need to be applied to the parameters involved in AXI bus transaction. So constraints will be applied on the write address channel, write data channel and write response channel parameters. Following snippet of code shows, how to write constraints and how to define sequence item needed for constrained random stimulus generation.

```
class write_addr_ch extends ovm_sequence_item;
……………..
        rand bit [3:0]i_awid_m;
        constraint awid {i_awid_m>=0; i_awid_m<16;}
        rand bit [AXI_AW-1:0]i_awaddr_m;
        rand bit [AXI_BLW-1:0]i_awlen_m;
        constraint length {i_awlen_m >=0; i_awlen_m<256; }
         rand bit [2:0]i_awsize_m ;
        constraint awsize {i_awsize_m==3'b010;}
        rand bit [1:0]i_awburst_m;
        rand bit [1:0]i_awlock_m;
        constraint lock {i_awlock_m==2'b00;}
        rand bit [3:0]i_awcache_m;
        constraint cache {i_awcache_m==4'b0000;}
        rand bit [2:0]i_awprot_m;
        constraint protection { i_awprot_m==3'b010;}
        rand bit i_awvalid_m;
        rand bit [AXI_MIDW-1:0]i_wid_m;
        rand bit [AXI_DW-1:0]i_wdata_m[];
        constraint data_size { i_wdata_m.size() == i_awlen_m+1;}
        rand bit [(AXI_DW/8)-1 :0]i_wstrb_m;
        rand bit  i_wlast_m;
        rand bit  i_wvalid_m;
        rand bit  o_wready_m;
        rand bit i_bready_m;
………
endclass
```

Consider awid_m parameter, for writing constraints. AXI bus supports only 16 slaves. While configuring the AXI bus, awid field was set to 4 bits to uniquely identify 16 different slaves. For example to access slave0, awid can be set to 0, for accessing slave1, awid can be set to 1 and so on. Hence the constraint was put on awid so that randomly different values can be generated from 0 to 16. It is not necessary to set awid to 0,1,2 and so on to identify the slaves. awid field is just transaction ID. By keeping the awid 0 for slave0, 1 for slave1 helps in debugging the transactions going on bus. Note that the key word rand was used while defining awid field. Key word rand denotes that the particular field will be randomized in the given range. For example a parameter is defined as follows

```
rand bit [7:0]val
```

Here if the constraints are not specified explicitly, then randomization engine of SystemVerilog (SV) will generate values from 0 to 255. But if the constraint was written as follows like

```
constraint  limit_val {val >= 2; val <= 50;}
```

Then above constraint define the range in which the field val must be randomized. Any attempt to randomize the val outside the given range will result in randomization error. By writing constraints, SV allows users to generate constrained random stimulus. AXI bus was configured to support burst transaction of length 255, hence the constraint was put awlen parameter so that randomly values can be selected only from 0 to 255. Encoding of the bits for awsize, awprot, awlock, awcache, awburst etc is decided by AMBA AXI specification. awsize is set to 2, since that implies 4 bytes are begin transferred in one burst transaction. awlock is set to zero, implying normal access. AXI bus supports normal, exclusive locked access. awburst is set to one, implying incrementing address burst mode. AXI bus supports 2 more burst types, one is fixed address burst and second is wrap burst transaction where incrementing address burst wraps to a lower address at the wrap boundry. wdata is declared as dynamic array whose length is constrained by the awlen. SV has constraint solver which will assign value to awlen first then constraints will be put on wdata size dynamically. Constraints are not written for awburst since the type of the burst transaction will be set during the runtime. For more information on constraints, constraint solver etc in SV refer to book SystemVerilog for verification (6). Sequence item is a class where the parameters, their ranges, and their constraints are defined. The advantage of extending the class write_addr_ch with ovm_sequence_item is the built in methods, macros defined in ovm_sequence_item. User can completely focus on the parameters, their constraints and their ranges in order to generate constrained random stimulus instead of interaction between sequence item and sequences. Before the data is applied through the driver to DUT, the data or the transaction is generated in sequence item. From sequence item the transaction is passed to sequence. After this transaction is sent to the driver via sequencer. Sequence item, sequence, sequencer, driver are all classes. The communication between different classes is managed by OVM for User. Through various OVM macros various parameters can be set from the test level as per the test writer. OVM methods and macros allow test writer to modify various verification environment configurations without changing underlying code. This sequence item forms the basis for write burst transactions which will be used in generating constrained random stimulus for the WiNC2R Platform. All the SV code related to building verification environment is stored under cog_svn/Design/trunk/ovm_tb folder.

4.2 OVM Sequence for WiNC2R Platform

The next verification component in OVM hierarchy is Sequnce. Sequence decides the nature of the transaction. Sequences are the back bone in generating constrained random stimulus which in turn are backbone for generating various complex scenarios for validating/ verifying DUTs.

Following snippet of a code shows a sequence written to initiate a single AXI Write burst transaction.

```
class single_write_burst_tran extends ovm_sequence;

write_addr_ch w;

……..

task body( )

for (int i=0; i<1; i++)

`ovm_do_with(w,{   w.i_awvalid_m==1;   w.i_aresetn         ==1;   w.i_awaddr_m==addr[i];
w.i_awlen_m==255;

w.i_awburst_m ==1; w.i_wdata_m[0]==data[i];});

for(int z= 1; z<256; z++ )begin

`ovm_do_with(w,{w.i_wdata_m[0]==data[z];

w.i_aresetn   ==1;});

endtask
…..

endclass
```

OVM provides default task called body. User must write the procedural code in the task body so as to generate and drive the transactions. The calling of the task body is managed by OVM for users. Consider the snippet of code as shown above. Inside the task body there is for loop has been coded so as to provide facility to generate different scenarios particular number of times. The loop control parameter can be easily set by test writer before executing this sequence.

Consider ovm_do_with macro. The first argument of the ovm_do_with argument is the object of the sequence item class write_addr_ch. ovm_do_with macro is used when inline constraints are applied on the parameter defined in sequence item. First in-line constraint in ovm_do_with macro is applied to awvalid signal. Note to assign a value 1 to awvalid signal assignment operator is not used, instead equality operator is used. Since the awvalid is forced to 1, whenever the randomization engine will be called, assigned value, in this case, decimal one value will be checked against the constraints written in sequence item class for the awvalid parameter. In the write_addr_ch class the awvalid field is defined as rand bit awvalid_m, implying the awvalid can be assigned any value between 0 and 1 (since this is one bit signal). Therefore by controlling the awvalid signal, the transaction going on AXI bus can be validated or invalidate as per the requirement. Similar explanations can be given for aresetn, awburst, awlen. So there are two places where constraints can be defined. One is sequence item and another place is in sequence by writing in-line constraints. Refer to book, SystemVerilog for verification (6) for detailed information on in-line constraints. Consider a simple example as follows

```
//constraint defined in sequence item

rand bit [31:0] addr

constraint addr_range {addr >= 100; addr <= 32’h4000_0000;}

// in-line constraint defined in sequence
`ovm_do_with (w, {w.addr == 200;})
```

Note that user can specify the in-line constraints only in the range defined in the sequence item class. Using the in-line constraints a specific scenario can be generated very easily. A logic can be written to increment the addr in linear fashion or decrement the addr in linear fashion or do increment or decrement in any fashion provided all operations generate the values in the specified range. Also note that awsize, awcache, awprot have not been assigned a value from ovm_do_with macro. Constraints written in sequence item class will be applicable to these variables when randomization call will be made. Once the ovm_do_with macro is called the in-line constraint will be considered while applying constraints to the parameters defined in sequence item class. Based on the in-line constraints, constraint solver of SV will generate random values for other parameters in their ranges. ovm_do_with macro will make sure that the newly generated values reach from sequence item to sequence class, from which it can be applied to the driver via sequencer. If the ovm_do_with macro calling sequence has a priority than other sequences or it's the only sequence running on sequencer then the generated parameters will reach driver via sequencer. This sequence will be basic for applying constrained random input to the WiNC2R platform as well for initial configuration of WiNC2R, implying for loading GTT, NT, TD memories.

4.3 OVM Sequencer for WiNC2R Platform

As explained earlier Sequencer just mediates between various sequences and established link between sequences and driver. The main use of the sequencer comes into the picture when there are many sequences running in parallel and they have weighted or fixed priority. In this scenario, sequencer needs to make sure that at the correct time correct sequence and driver link is established. Connection between driver and sequencer is made during the connect phase which is called at agent level.

4.4 OVM Driver for WiNC2R Platform

OVM Driver has three main duties to follow.

1. Get new transaction from Sequencer
2. Drive new transaction on virtual interface connecting verification environment to DUT
3. Comply to the Protocol for driving the transaction

OVM gives seq_item_port which is built into ovm_driver class. seq_item_port is a bidirectional port and includes TLM methods called get ( ) and peek ( ). Using the seq_item_port driver and sequencer interact with each other via TLM channel. Series of actions take place when either get or peek task is called. There are two modes in which driver and sequencer can work namely

1. Push mode
2. Pull mode

In push mode, a sequencer drives a produced item into a driver when that item is generated and waits till the driver consumes this item. In pull mode driver demands the new transaction and gives feedback when the driver is done consuming the data. The pull mode is superior that push mode for following reasons

1. In pull mode, a sequence item is immediately consumed after it leaves the sequencer. This means that sequencer can customize the sequence item to the timing consumption of the sequence item unlike push mode where sequencer will wait till the driver finishes consuming the data.
2. Single stream of sequence items leaving a sequencer may represent multiple concurrently running scenarios and pull mode gives sequencer chance to arbitrate among the items generated by these concurrently running sequences.

Hence pull mode is preferred than push mode. So while implementing driver pull mode was use than push mode. get ( ), get_next_item ( ), item_done ( ), peek ( ) etc are various built in tasks provided by OVM for interaction between driver and sequencer. get( ) task is used in driver implementation since item_done ( ) need not be called explicitly after the items are consumed by the driver. If get_next_item ( ) task is used then item_done task needs to be called explicitly and if peek ( ) task is called then also item_done ( ) task call is required to make. Once get ( ) task is called feedback is already sent to the sequencer and there is no need to explicitly send feedback to sequencer. In pull mode, once driver requests for the transaction then following series of actions occur as follows

1. Driver sends request to the sequencer
2. Depending upon the current arbitration scheme, sequencer will choose one sequence at a time
3. Selected sequence will be executed and Send the item to sequencer's fifo
4. Send the item to the driver from sequencer
5. Driver can send feedback by calling item_done ( )

Again driver can send the request to the sequencer. For more information on driver, sequencer interactions refer to OVM User Guide (17). Once the driver gets the transaction, driver assigns the transaction to the virtual interface. Driver is the place where transactions are converted into pin level signals.

Since the WiNC2R is AXI bus based system, the driver has to follow the AMBA AXI protocol as explained earlier with Figure 6. Following snippet of code shows the run phase of the driver

```
Class write_driver extends ovm_driver;

………

virtual task run();

forever begin

get_new_transaction();

drive_addr_ch(t);

check_awready();

drive_data_ch();

read_response();
```

```
end

endtask

……

endclass
```

run task of driver is kept in forever loop, so that transactions can be sent continuously to the DUT. This run phase involves following tasks

1. Get the transaction from sequencer
2. Drive address and corresponding control information on the interface
3. Wait for awready from slave
4. Drive the data and corresponding control information on the interface and wait for acknowledgement from slave
5. Once all the transactions are over then wait for the responses for the current transaction.

Once valid responses are received, driver can ask for the next new transaction from sequencer. Currently implemented driver supports AMBA AXI protocol.

4.5 OVM Monitor for WiNC2R

The basic purpose of the monitor is as follows

1. Monitor the transactions,
2. Send the appropriate information to scoreboard,
3. Perform protocol related checks and other basic checks
4. Gather coverage information.

OVM monitor provides ovm_analysis_port, using which monitor can send the necessary information to the scoreboard and to any verification component in the verification environment.

There can be many monitors in the environment for various different interfaces serving different purpose. Verification/ Validation environment implemented for WiNC2R has many different monitors. There are two monitors which are monitoring the read and write transaction going on AXI bus from OVM driver. There is one monitor per single Functional Unit (FU) of WiNC2R. Currently WiNC2R has seven FUs, implying seven monitors. These monitors are connected on different interface than two monitors connected to the AXI bus of OVM driver. Following snippet of code shows one OVM monitor connected to the AXI bus monitoring OVM driver.

```
class monitor extends ovm_monitor;
…
write_addr_ch write_tran; // creating a object of write_addr_ch
ovm_analysis_port #(write_addr_ch) write_tran_port;
…
virtual task run();
 forever begin
begin_recording_tran();
check_awready();
write_tran_port.write(write_tran);
end // end of the forever begin
endtask : run
….
endclass
```

Run phase of the monitor is also kept in the forever loop, so that signals can be monitored continuously. Run phase of the monitor waits till the awvalid signal. After that monitor waits for the awready signal. Checks are provided so as to check that valid acknowledgements are received from slave and correct valid signals are driven as per the protocol. Once the valid data is put on the bus, coverage group is triggered every time new data is driven on the bus. For detailed information on coverage, refer to SystemVerilog for Verification (6). Consider `ovm_analysis_port macro. By using this port, monitor can send the current transaction to the scoreboard. Write_addr_ch in `ovm_analysis_monitor specifies data structure of the transaction to be sent to the scoreboard. Write_tran_port is the name of the analysis port. Due to use of `ovm_analysis_port macro, users get write method by which they can send the current transaction to the necessary verification component.

Following snippet of code shows a simple covergroup

```
covergroup cov_write_tran @ record_coverage;
option.per_instance =1;
addr : coverpoint write_tran.i_awaddr_m {
bins valid[]    = { [0:9] };
illegal_bins invalid_addr = { [10:$]}; }
data : coverpoint write_data { option.auto_bin_max=8; }
len: coverpoint write_tran.i_awlen_m  { option.auto_bin_max=8; }
endgroup : cov_write_tran
```

cov_write_tran is the name of the covergroup which will be triggered by record_coverage event. Once new data is validated on the bus, record_coverage event will be triggered. There are various options in the covergroup. Option.per_instance set to one implies that per monitor object this covergroup will be instantiated once per object of the class. Various coverpoints are defined in the covergroup to sample the values of the parameters. The goal of the coverage is to measure how many valid input combinations have been applied to the DUT. For more information on the coverage refer to SystemVerilog for Verification (6). Coverage information helps in understanding the progress of the verification process. addr is name given to the coverpoint write_tran.i_awaddr_m. write_tran is an object of class write_addr_ch. This will sample the AXI address available on its interface. User should specify the valid range for sampling of the parameters. Specifying the valid range for the parameters is necessary in order to have correct measure of the coverage. Bins is a SV construct by which user can specify the valid range. So line 4, implies that only 10 bits of the awaddr will be sampled. In this case, that means addresses from 0 to 1024 will be considered in a valid address range. Next SV construct is illegal_bins, which allows user to specify invalid range of the parameter. Here in this case address from 1025 to (2^32 -1) will be considered as illegal address. SV can create automatic bins for user considering the width of the parameter. For example for 3 bit variable, SV will create 8 automatic bins, one per unique combination. User can also specify the number of automatic bins SV should create for the parameter. Line 6 shows option for automatic bin creation. As shown in line 6, data will be sampled into 8 bins. So first bin will sample values from [0:3] that is from 0th bit to 3rd bit, then second bin will sample values from [4:7] and so on. For more information on coverage refer to SystemVerilog for Verification (6).

4.6 OVM Scoreboard for WiNC2R

Through driver, transaction or particular input combination is applied to the DUT. Based on the input combination, DUT produces output. The correctness of the output is determined in the scoreboard. Monitor sniffs on the interfaces and send the information to the scoreboard. In basic implementation of the scoreboard for WiNC2R, during write transaction, correct data is sent to the scoreboard which is getting loaded into the memories. During the read transaction another monitor sends the read data to scoreboard for checking the data integrity. Scoreboard has `ovm_analysis_imp_port due to which scoreboard can receive the transactions from different monitors. Every ovm_analysis_imp_port provides write method which is called automatically when the monitor sends the data. Scoreboard can have only function, since all function calls are returned in zero time. Consider following snippet of code for scoreboard

```
`ovm_analysis_imp_decl(_write)

`ovm_analysis_imp_decl(_read)
```

```
class scoreboard extends ovm_scoreboard;

ovm_analysis_imp_write #(write_addr_ch,scoreboard)write_port;

ovm_analysis_imp_read #(read_addr_ch_seq_item,scoreboard)read_port;

…..

virtual function void write_write(input write_addr_ch write_tran);

memory_write(write_tran);

endfunction : write_write

……

virtual function void write_read(input read_addr_ch_seq_item read_tran);

memory_read(read_tran);

endfunction : write_read

………

endclass
```

`ovm_analysis_imp_decl(_write) implies that name of the write method associated with this ovm analysis import will be write_write and `ovm_analysis_omp_decl(_read) implies that the write method associated with this ovm analysis import will be write_read. There is one write method associated with each ovm_analysis_imp port. If there are more ovm_analysis_import ports then every port needs to have different write method. Consider following line of the code

```
ovm_analysis_imp_write #(write_addr_ch,scoreboard)write_port;
```

Here write_addr_ch specifies that scoreboard class will receive a transaction of type write_addr_ch and name of the port is write_port. Similar explanation can be given for read_port. write_write method will be called when the monitor class will send the transaction to the scoreboard. write_read method will be called when monitor_read class will send the transcation to the scoreboard (monitor_read, monitor is used for sniffing read transaction on AXI bus initiated by OVM driver). memory_write function is used to store the data sent by monitor class. memory_read function is called to check the data integrity when monitor_read send the data to the scoreboard. The connection between scoreboard's ovm_analysis_imp and monitor's ovm_analysis_port is made during the connect phase. Connect phase is called at an environment level where scoreboard is instantiated as shown in Figure 4.

OVM agent, OVM env are implemented just to complete the hierarchy for verification environment. For WiNC2R, there is no need for multiple environments. Current verification environment has 2 agents one encapsulating the driver, monitor and sequencer responsible for write burst transaction called agent_write and other encapsulating driver, monitor and sequencer responsible for read burst transaction called agent_read. Next chapter will focus on generating constrained random stimulus to validate the architecture and on checking the correctness of the output from WiNC2R.

4.7 OVM Test for WiNC2R

OVM test is the level from which test writer can do the necessary configuration for the particular scenario. Test writer can implement his algorithm at this level to generate a particular scenario, by selecting sequences. Three different environment setups are created for the current implementation of WiNC2R. One setup is created to support single flow. Second setup is created to support multiple flows. Third setup is created where instead of processing engines in 802.11a like OFDM transmitter flow, Golden PE is used. Golden PE is like a dummy PE which will mimic the behavior of the normal PE. This setup is useful for experimentation in terms of determining certain system related parameters. Single flow means currently system is configured to support one virtual flow. As described in chapter 1, it is essential to load the correct GTT, TD and NT tables for correct functioning of the system. So while doing any setup for the WiNC2R, it is essential to load the Memories i.e. GTT, TD, NT tables correctly as part of the configuration. Later part of the setup is related to the requirements. Consider following steps for the single flow setup

1. Load the GTT, TD, NT table
2. Load the Task Descriptor for MAC
3. Wait for an interrupt from MAC
4. Insert the data into Input Buffer of MAC
5. Repeat step 2-4 till specified number of frames

OVM sequences are chosen for particular activity from OVM test level. By selecting proper sequences, any setup can be done using the same environment. For example, to create a setup for multiple flows, just one parameter of the sequence needs to change and rest of the setup can still remain same as single flow. Due to this feature of OVM Methodology, verification components become highly reusable.

Due to ease of adaptability to different requirements, OVM based verification environment becomes versatile. Currently configured OVM based verification environment for WiNC2R is used for single flow setup, multiple flow setup and setup with Golden PE. Also same verification environment is used for extracting performance from the system. Setup with the Golden PE will be used for validating the architecture and to determine certain system related parameters like guard time.

5. Constrained Random Stimulus Generation for validating WiNC2R Scheduler

Constrained random stimulus generation is done by applying constraints on the input parameters. In order to generate constrained random stimulus for WiNC2R, it is important to understand how input is applied to WiNC2R platform, the nature of the asynchronous (async) descriptor, synchronous (sync) descriptor, GTT and TD table descriptors. Parameters defined in the GTT, TD tables and parameters defined in async and sync task descriptors determine the input variables which can be constrained. Whenever any block in the system needs to insert one task in the system, has to form a sync or async descriptor and write that descriptor to the Task Scheduler Queue (TSQ). Once the task is written to TSQ then, it is the scheduler's job to schedule the particular task as per priority. Consider async task descriptor as shown in Figure 8.

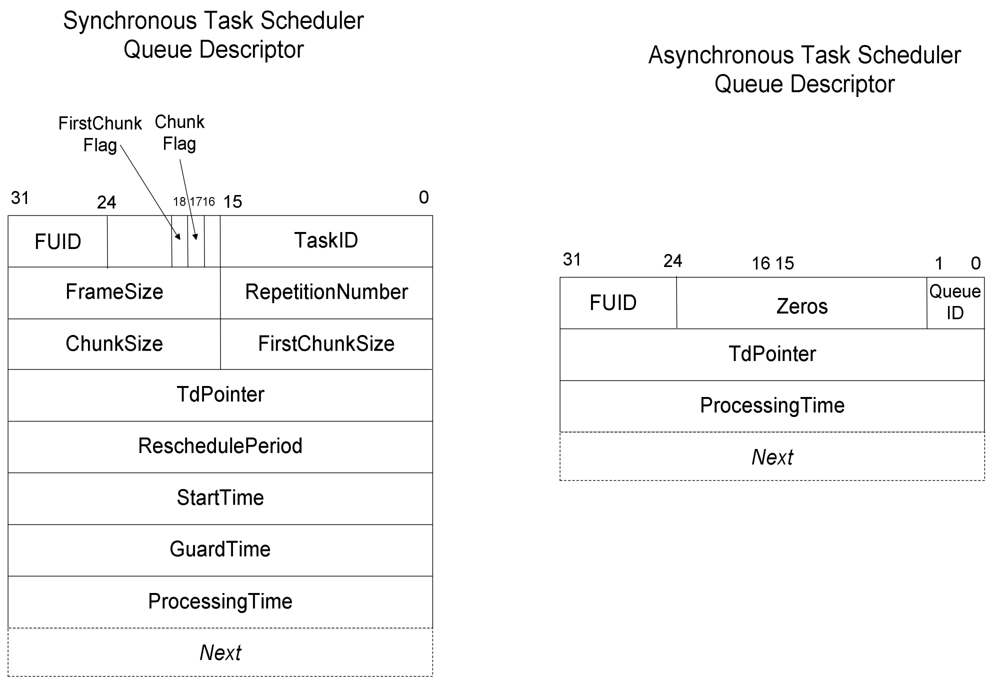

Figure 8: SYNC and ASYNC Task Queue Descriptor (18)

Async descriptor has following fields

1. FUID- FU Identification is Fixed for single FU. Current WiNC2R platform has 7 FUs. So FUID ranges from 0 to 6.
2. Queue ID- Asynchronous tasks can be inserted into 4 different queues. There are 7 FUs and per FU, 4 different asynchronous queues are there, totaling 28 asynchronous queues.
3. TdPointer- TdPointer is the address location in TD table associated with current async task.
4. Processing Time- Processing time indicates the expected processing delay from PE for the current async task

In WiNC2R platform, scheduler is responsible for resolving priority between async and sync task. Processing Time is used to determine the priority between sync and async task. If the processing time of the async task is less than the addition of the start time and guard time for the clashing sync task then async task is given priority over sync task. If the processing time of the async task is greater than addition of the start time and guard time of the clashing sync task then priority is given to sync task.

Sync Descriptor has following fields

1. FUID- FU Identification is fixed for single FU. Current WiNC2R platform has 7 FUs. So FUID ranges from 0 to 6.
2. First Chunk Flag- First Chunk flag if set has valid information related to the first chunk size, chunk size and frame size.
3. Chunk Flag- Chunk flag if set indicates the chunking sync task.
4. Frame Size- Frame Size indicates the remaining bytes to be processed by the PE compared to the initial Frame Size.

5. Repetition Number- Repetition Number field is used when pure sync task is scheduled for PE. Repetition number indicates how many times, current sync task needs to be executed.
6. Chunk Size- Chunk size indicates the total number of bytes processed by PE for the current task.
7. First Chunk Size- First Chunk Size field indicates the number of bytes to be processed by PE, when the current sync task was activated for the very first time.
8. TdPointer- TdPointer is the address location in the TD table pointing to current sync task.
9. Reschedule Period- Reschedule Period indicates the time when the sync task needs to be rescheduled for next execution.
10. Start Time- Start time field indicates the time at which sync task should start execution.
11. Guard Time- Guard time field indicates the end of the timing window during which the sync task needs to be activated for PE. Timing window begins at the start time of the sync task and ends at time where time is addition of start time and guard time.
12. Processing Time- Processing time indicates the expected processing delay from PE for the current sync task.

Sync tasks are repetitive in nature. Sync tasks needs to be activated in a particular time window based on start time and guard time. If First chunk Flag is set along with chunk flag, this is an indication to scheduler, that the current sync task is the chunking sync task which got inserted in the sync queue for the first time. This also implies that the Frame Size is the total number of bytes which needs to be processed by PE. First Chunk Size indicates the initial number of bytes which need to be processed by PE when the task is activated for the first time. For example, First Chunk Flag and Chunk Flag is set to one, Frame Size is set to $(800)_{10}$ and First Chunk Size is set to $(200)_{10}$ and Chunk Size is set to $(100)_{10}$. When this chunking task sync task will be activated, PE will process $(200)_{10}$ bytes for the first time. Once this sync task is activated, scheduler will form the sync descriptor again and the newly formed descriptor will be written to sync queue. This process will repeat till all the bytes are processed by PE. Sync task is activated based on the start time and guard time mentioned in the sync descriptor. Reschedule period decides the start time when the sync task descriptor is formed again. When this task will be activated for the second time, the PE will process $(100)_{10}$ bytes and this time Frame size will be updated as $(400)_{10}$ indicating $(400)_{10}$ bytes are left for processing.

Consider the TD Table format as shown in Figure 9

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TaskID_SendDataFrame = 0x000																FUID = 0x00								Cmd_SendDataFrame = 0x01							
StartTime[31:0]																															
-	-	-	-	-	-	-	-	NumNextTaskID = 2				NumInBuf = 1				-	-	ContextFlag = 0	DeChunk Flag = 0	Chunk Flag = 0	First Chunk Flag = 1	Intr Enable Flag = 0	Reload-Eble= 1 Flag	Reserved							
ChunkSize[15:0]																FirstChunkSize[15:0]															
FlowContextPtr[15:0]																FlowContextSize[15:0]															
Processing Time																															
NTPointer[31:0] = VFP Base + NT Offset = 00380000 + 50000 = 003D 0000																															
InDataPtr_0 = Mac_Tx Base + Inbuff offset + Data buffer offset = 00000000+50000+20 = 00050020																															
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	InDataSize_0 = 0x0000															

Figure 9: TD Table Format (18)

Consider GTT table format as shown in

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BusyVector = 0x0000 (default)																Enable Flag = 0								EnableFlagInit =0x1							
FuncUnitID_0_MacTx = 0000 0000								FlowContextSize=0x0								-	-	-	-	-	-	First Chunk Flag = 0	Chunk Flag = 0	-	-	Asynch Move = 0	DynDataSize = 1	Task Priority = 01		Sync/Async=1	TaskActive=1
ChunkSize[15:0] = 0x0000 (default)																FirstChunkSize[15:0] / RepetitionNumber[15:0] = 0x0000 (default)															
TD Pointer_0_SendDataFrame = MAC_TX base + TD pointer = 0x0000 0000 + 0x58000 = 0x00058000																															
Tsq Pointer = vfp_base + tsq offset + AsyncQ1 offset = 0x00380000 + 0x20000 + 0x0004 = 0x003A0004																															
ProcessingTime																															
ScheTimeSft																															
Guard Time																															
Reschedule Period																															

Figure 10: GTT Table Format (18)

For every task in the system, there will be corresponding necessary information present in GTT and in TD table. GTT, TD table, sync and async task descriptor have certain parameters common. It is essential to know which parameters, are common among GTT, TD table, sync and async task descriptor in order to generate the GTT, TD tables automatically at run time and then apply the constraints dynamically as per the generated tables. Task descriptors will be formed based on the automatically generated GTT, TD tables. Earlier to verify the system's behavior, GTT, TD, NT tables were computed manually. This restricts the number of tasks written to test or verify the system. Also this process of writing task is erroneous, strenuous and slow. Due to automatic table generation, quickly many tasks can be generated randomly per FU. 32KB space has been allocated for TD table and each task descriptor is atleast of 36 bytes. Considering 36 bytes task descriptor and 32KB TD table size, 910 tasks can be generated per FU. The purpose of generating all these random tasks is to verify the performance of the system and to validate the architecture. The procedure to generate the tables automatically in random fashion is quickly scalable and compared to manual effort, reduces the time to verify or validate the architecture. Considering the parameters in GTT, TD table and task descriptors, following parameters are initialized and constrained per FU while generating automatic GTT, TD tables.

1. Task ID- Initialized to Zero only for the first time and then later incremented by 36, since every descriptor in GTT table is of 36 bytes.
2. FUID- FUID is unique per FU. It ranges from 0, 1, 2, 3, 4, 5, 6 (0 to 6)
3. Physical Address(PA) of FU- This address is unique per FU and it has following valid values

   (FU0 PA- 0x0, FU1 PA-0x80000, FU2 PA-0x100000, FU3 PA-0x180000, FU4 PA-0x200000, FU5 PA-0x280000, FU6 PA-0x300000)
4. Starting Address of TD Table- This value is initialized per FU to 0x58000

5. TSQ address for async task queue- Every FU has one async task queue and one sync task queue. From async task queue address it is very easy to compute the address for sync task queue, since PA of sync task = PA of async task – 4. Valid values for TSQ are (FU0 async TSQ PA- 0x3A0004, FU1 async TSQ PA- 0x3A2004, FU0 async TSQ PA- 0x3A4004, FU0 async TSQ PA- 0x3A6004, FU0 async TSQ PA- 0x3A8004, FU0 async TSQ PA- 0x3AA004, FU0 async TSQ PA- 0x3C0004)
6. Start time of sync task- start time is randomly chosen in a range of 500 to 1500.
7. Guard time of sync task- guard time is chosen randomly from 50 to 150
8. Reschedule period of sync task – Reschedule period is chosen randomly between 500 to 800
9. Input Data pointer- Input data pointer is set to 0x50020.
10. Input Size pointer- Input size is chosen randomly between 100 to 300
11. Processing time of FU- Processing time FU is chosen randomly between 100 to 400
12. Frame Size- Frame size is chosen randomly between 20 to 1536
13. Repetition Number- Repetition number is randomly chosen between 10 to 200
14. Chunk Size- Chunk size is randomly chosen between 20 to 100
15. First Chunk Size- First chunk size is chosen between 100 to 300
16. Queue ID (QID) for async task- QID is chosen randomly from 0, 1, 2, 3, since there are 4 different async task queues.
17. Chunk Flag- Depending upon async or sync task, chunk flag is set. For async task, chunk flag is set to zero and vice versa.
18. First Chunk Flag- Depending upon the async or sync task, first chunk flags is set. For async task, first chunk flag is set to zero and vice versa.

These parameters are selected so that valid task descriptors are formed. There are other parameters as well as shown in GTT and TD table, but apart from above parameters, all others parameters are initialized to valid values. It is essential to correctly define valid input variables and their valid ranges, in order to generate constrained random stimulus. While generating Tables, constraints are applied on above parameters, so as to generate valid input combination. At times, heuristics are applied to constrain certain parameters. Once the valid input range is defined, constraints are applied while selecting the particular input combination. Consider following snippet of code showing, automatic tasks generation.

```
for (int i=0; i<NO_FU; i++) begin

//initialization for FUs

for (int z=0 ; z< NO_ASYNC_TASK; z++) begin // logic for async task generation

if(z > (NO_ASYNC_TASK/2)) QID= $urandom_range(1,3); // task written with QID != 0
are async data tasks

else QID= 0; // task written with QID=0 are async control tasks

//code to store the values in memory which can be accessed later for generating random
task descriptors

//code to write the current descriptors into a file, which can be loaded into Design via
OVM sequences

TaskID= TASKID + 36; TD_Pointer= Td_Pointer + 36;

…………

end

for (int z=0 ; z< NO_SYNC_TASK; z++) begin // logic for sync task generation

if(z > (NO_SYNC_TASK/2)) begin
```

```
chunk_flag=1; first_chunk_flag=1;
end
else begin
chunk_flag=0; first_chunk_flag=0;
end
start_time = $urandom_range (500, 1500);
……….
end
end
```

Once the tables are created, sequences will chose tasks randomly. There are various different scenarios which can be created using OVM sequences.

Consider Figure 11 and Figure 12 depicting scenario where many parallel sequences are running simultaneously.

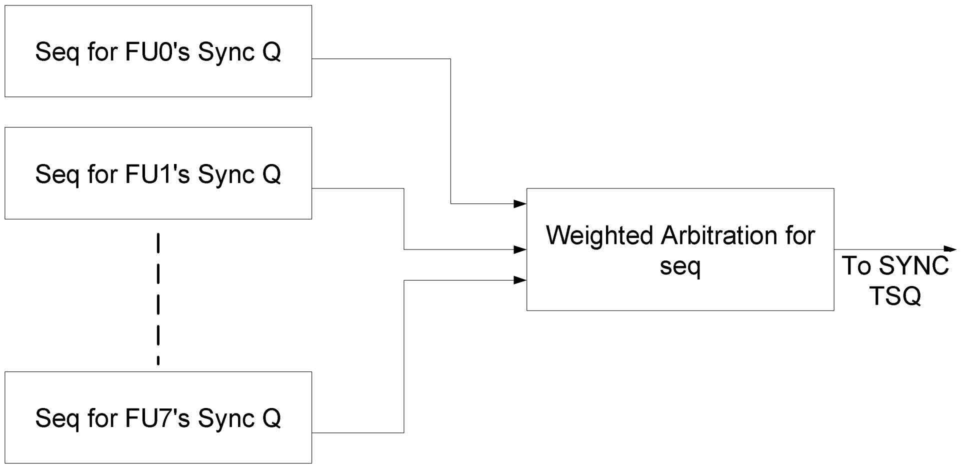

Figure 11: Arbitration between Sync Sequences

Each sequence will write a sync task descriptor to particular FU's sync task queue. Based on priority, only one sequence will be executed at a time. User can define the weightage given to every sequence. Current implementation of OVM based environment allows user to define the weightage. Also user can specify the number of transactions per FU. For example, user can specify how many task descriptors he wants to insert into a TSQ. Similar explanation can be given for scenario shown in Figure 12.

As shown in Figure 13, very realistic scenario is generated using OVM sequences. Based on priority, a sequence will be executed which may result in writing async task or sync task in one TSQ. This has generated very powerful and realistic scenario to validate the system. This setup is useful for finding the bugs in the system. Also same setup is used to validate the functionality of the scheduler. It is very much important to validate the system's behavior before extracting the performance of the system or characterizing certain parameters of the system.

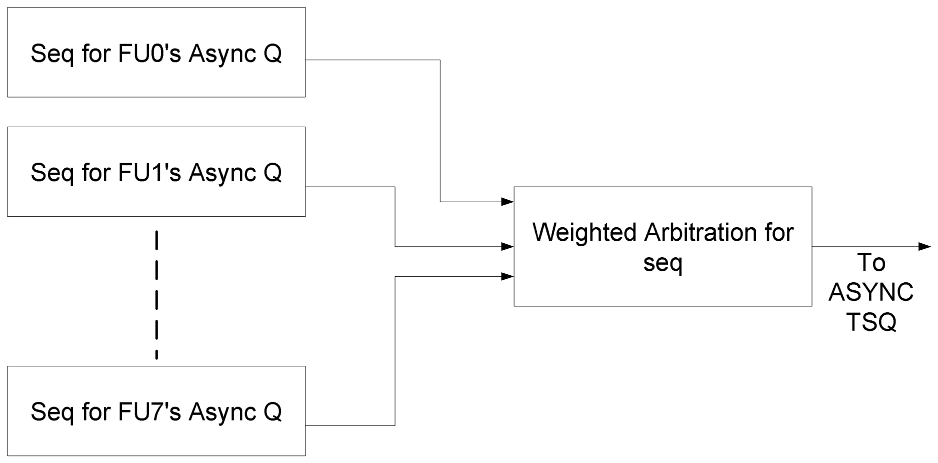

Figure 12: Arbitration between Async Sequences

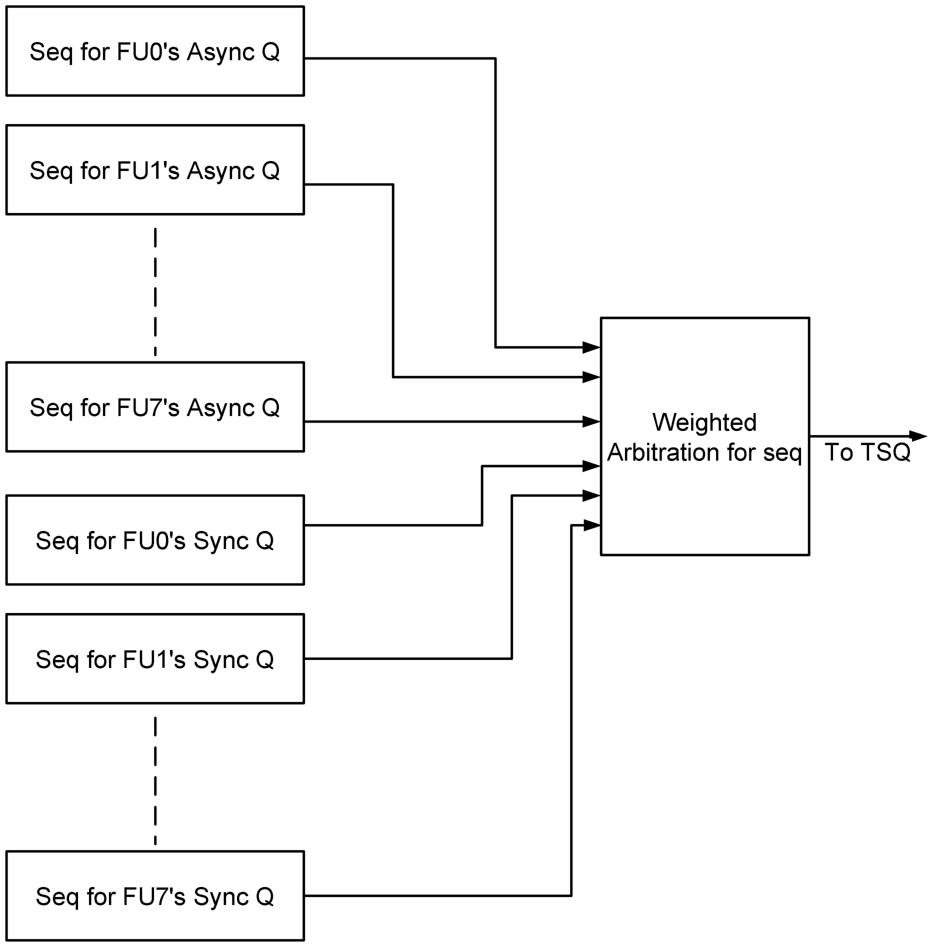

Figure 13: Arbitration between Async and Sync Sequences

Scenario shown in Figure 13 was used to verify the basic functionality of the scheduler which is backbone of VFP controller. Scheduler resolves the priority between sync and async task as follows

1. Async Control task has the highest priority
2. Sync data task has second highest priority
3. Async Data task has least priority.

Among async data task, async data task written into async Q1 has highest priority over async task written in async Q2 and async Q3. While checking the basic functionality of the scheduler it was checked that the scheduler resolves the priority correctly, schedules and activated the correct task at correct time in case of sync task. Scheduler passes the basic functionality test. Due to this, this setup can be used more confidently to characterize certain system related parameters.

The current setup was used to budget for guard time in case of synchronous (sync) tasks. From the performance analysis it was known that the Scheduler takes around 20-30 clock cycles to activate a sync task. In order to cater different sync task of different FUs having same start time, needs proper estimation of guard time. If the guard time values are wrong, the sync task will not be activated at a correct time interval. Therefore it is absolutely necessary for the system programmer to know what should be the guard time. Theoretically the guard time should be total number of FUs which can have sync task times the average task activation required for sync task. Based on theoretical grounds, experiments were ran to check the validity of the claim. While testing this, 7 different sync tasks were inserted for 7 different FUs having same start time and guard time equal to 7*20. But this showed that atleast 2-3 sync tasks missed their start time. Hence the guard time was increased to 7*30 and this showed that scheduler could schedule all the 7 sync tasks for all 7 FUs having same start time. Hence the guard time is a function of average sync task activation period and Number of FUs having sync task at a time in a system.

6. Performance Analysis of VFP overhead

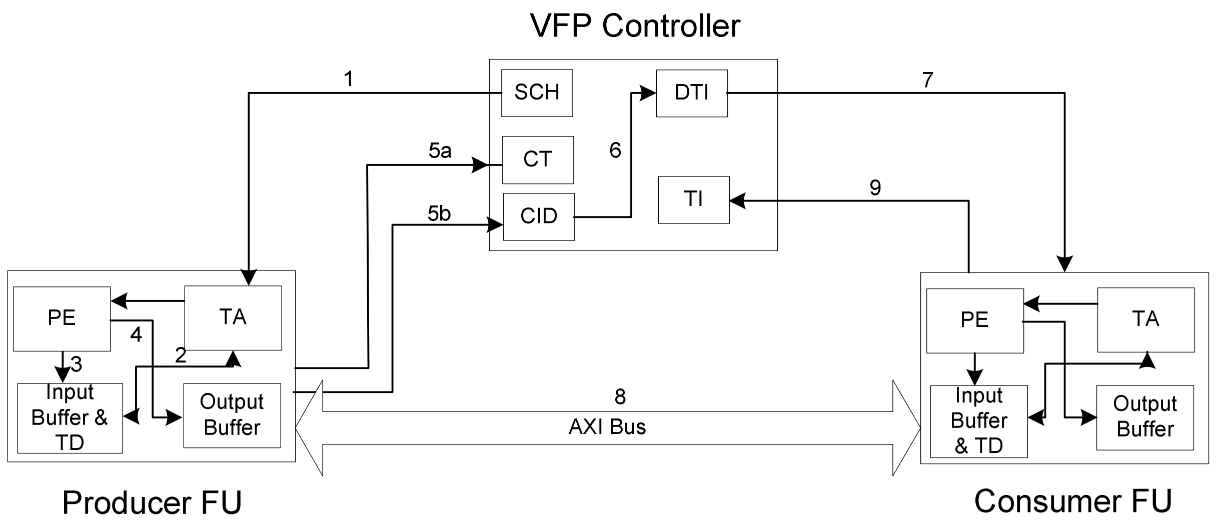

Figure 14: Producer Consumer Interaction in WiNC2R

Consider Figure 14 depicting producer and consumer Interaction with the help of VFP controller. PE will start processing data only after task activation from VFP. So even, Input buffer of consumer may have data present which is to be processed by the consumer, but the consumer will wait for the command from VPF. The messaging between producer and consumer via VPF and the waiting period of PE for task activation command is termed as VFP overhead. As shown in Figure 14, Scheduler (SCH) of VFP controller gives task activation (TA) command to particular FU, here to the producer FU. TA block present in FU then based on the command from SCH, reads the TD table for data size to be processed by PE, the starting address of the data to be processed by PE, and number of output buffers associated with it. TA block gives this information to PE and gives him activation command. PE finishes its processing the writes the processed output to Output Buffer. After writing output to Output buffer, PE sends a message to VFP indicating the task termination. Command Termination (CT) block is responsible for task termination, which tells the VFP that this producer has finished its processing. Consumer

Identification (CID) block, then identifies the corresponding next consumer by accessing the NT table. After Identifying the correct consumer a request is put into Data Transfer Initiator (DTI) block. DTI is responsible for initiating the DMA transfer between appropriate producer and consumer. DTI sends a message to appropriate consumer if the consumer is not busy with the previous same task. If the consumer is busy with the previous same task, then DTI request is held back till the consumer finishes previous same task. Each task has a buffer region associated with it. If the consumer is busy with the same task, implies it has not finished reading the data for processing. If the new data is written to the same region, before PE finishes reading it, will result in error. Consider Scrambler as a consumer. Scrambler currently just scrambles the data. So every time scrambler scrambles the data, based on the VFP command. If the scrambler has not finished reading the data from the input buffer, and now if new data is inserted into the input buffer of scrambler, scrambler will process this new data. This will result in generating wrong output from scrambler. To avoid this erroneous situation, DTI request is held back till the consumer finishes processing data. Once consumer is free, DTI request is sent to the consumer. This message tells consumer, from which producer, he has to read the data. After this DMA transfer occurs on AXI bus. After DMA transfer, consumer informs VFP, that he is ready to process the data. This message is sent to Task Inserter (TI) block in the VFP. TI block inserts the task for the consumer into the TSQ. Once all dependency checks are done by SCH, SCH activates the task for the consumer. This interaction between producer and consumer occurs every time producer produces the data to be processed by consumer. Due to this, it is very important to characterize VFP overhead under different condition. The VFP overhead is characterized under single and multiple flow scenarios, also resource utilization is computed to check which resources are underutilized. Since WiNC2R can support multiple concurrent OFDM flows, performance analysis is done for following cases

1. Single ODFM flow with different data rates (6, 12, 24 Mbps) with 400bytes frame sizes

2. Two OFDM flows with different data rates (6, 12, 24 Mbps) with 400 bytes frame size.

Following are various latencies involved in VFP overhead

1.TA block in FU to Valid Command to PE Delay (t_{ta2cmd_valid})

2.Consumer Identification Delay (t_{cid})

3.Data Transfer Initiation Delay (t_{dti})

4.DMA overhead ($t_{dma\_overhead}$)

5.Task Insertion Delay (t_{ti})

6.Task Activation Delay (t_{ta})

7.Command Termination Delay (t_{ct})

To extract various latencies from the system, many monitors were written in SystemVerilog. These monitors are part of OVM based verification environment. Through various interfaces connected to OVM based verification environment, latencies are calculated. To understand the effect of VFP overhead, the latencies are compared to the processing latency (t_{tp}) of the PE. Following tables show the average VFP overhead for single and two flows scenario.

Table 2: Average VFP Overhead Latencies for Single Flow

Rate (MBPS)	t_{ta2cmd_valid}	t_{cid}	t_{dti}	$t_{dma\_overhead}$	t_{ti}	t_{ta}	t_{ct}
6	15	39	68	19	4	28	8
12	15	40	68	23	4	28	8
24	15	39	65	23	4	28	8

Average values of the VFP overhead latencies are tabulated. Also all the latencies are in terms of clock cycles. Except t_{dti} all other latency values are computed per FU.

Table 3: Average VFP Overhead Latencies for Two Flows

Rate (MBPS)	t_{ta2cmd_valid}	t_{cid}	t_{dti}	$t_{dma\_overhead}$	t_{ti}	t_{ta}	t_{ct}
6	15	37	528	22	4	361	8
12	15	37	553	23	4	387	8
24	15	34	556	26	4	316	8

As shown in Table 2, Table 3, the t_{ta2cmd_valid}, t_{cid}, $t_{dma\_overhead}$, t_{ti}, t_{ct} remains pretty constant under single and two flows case. The t_{ta} and t_{dti} change drastically in two flows case compared to single flow case. The utilization of PEs and utilization of Bus is also considered while deciding the reasons for increased t_{ta} and $t_{dti,}$ since if the bus utilization is very high, it will indicate that the bus is creating the bottleneck. If the utilization of the bus is low, then it implies bus is not creating the bottleneck. Similarly high utilization of PEs will indicate that the PEs are busy for maximum amount of time, and hence it can create a bottleneck. If the PEs utilization is low, then it clearly indicates that the VFP is creating the bottleneck, due to which t_{ta} and t_{dti} increases in two flows case. Following tables show the PE utilization in single and two flows case under 6, 12 and 24 MBPS rate.

Table 4: PE utilization for single and Two Flows at 24 MBPS

PE	PE Utilization in Single Flow	PE Utilization in Two flows	Relative difference in PE Utilization
Header	8.574958264	51.91300276	6.054023957
Scrambler	40.84540902	81.43303674	1.993688855
Encoder	51.51919866	83.92816302	1.62906577
Intlerleaver	44.40734558	83.36400963	1.877257209
Modulator	57.87178631	83.96519239	1.450883025
IFFT	51.14657763	66.47752644	1.29974535

Table 5: PE utilization for single and Two Flows at 12 MBPS

PE	PE Utilization in Single Flow	PE Utilization in Two flows	Relative difference in PE Utilization
Header	7.137813618	58.09916701	8.139630722
Scrambler	22.57849516	85.67717299	3.794636107
Encoder	28.25099975	91.20852377	3.228506056
Intlerleaver	40.8494899	92.98114402	2.27618862
Modulator	54.57047077	93.51694664	1.713691403
IFFT	49.14641872	80.29025939	1.633695018

Table 6: PE utilization for single and Two Flows at 6 MBPS

PE	PE Utilization in Single Flow	PE Utilization in Two flows	Relative difference in PE Utilization
Header	6.128061699	60.60343422	9.889494785
Scrambler	12.3927352	86.43135541	6.974356672
Encoder	15.1352784	89.81423786	5.93409883
Intlerleaver	28.90083947	91.69998285	3.172917623
Modulator	49.81487874	92.38205861	1.854507347
IFFT	46.14402974	82.46080469	1.78703085

Table 7 indicates the bus utilization under single and two flows case at 6, 12, 24 MBPS rate.

Table 7: Bus utilization for Read Channel of AXI bus

Rate	Single flow	Multiple flow	Relative Difference Between single and Two Flows
6MBPS	23.14612881	40.9412729	1.768817293
12MBPS	26.10764297	42.5790942	1.630905335
24MBPS	29.59732888	38.40054891	1.297432922

The AXI bus has write channel as well, but the write channel's utilization for single and two flows case under 6, 12, 24 MBPS is around 1%. Reason for low write channel utilization of bus is, write channel is used only during the configuration time to load the GTT, TD, NT tables. Every producer consumer interaction happens on read channel of the axi bus. As shown in Table 7, under single flow case bus is only 25-30% utilized, implying bus is idle for long time, and a lot of band of the bus is getting wasted. Under two flows the AXI bus has a utilization of around 40%. This indicates that compared to single flow case, in two flows case bus utilization increase by atleast 1.5 times. But still this utilization shows that bus is not responsible for creating a bottleneck resulting in increase of t_{ta} and $t_{dti.}$

Consider the Utilization of PEs under single and two flows case at 6, 12, 24 MBPS. In all the cases compared to single flow, utilization of PEs is quite high. This indicates that the PEs were more busy compared to single flow case. In two flows 12 MBPS case, modulator was utilized 93%, indicating that scheduling scheme is keeping the PEs busy for maximum amount of time.

The processing latencies of the PEs has increased in case of two flows case. This indicates that the bottleneck was created due inherent slow response of the PEs. The slowest PE creates a back pressure on other PEs in the WiNC2R. The slowest PE in WiNC2R is Modulator. If the modulator is busy processing the data, then a task cannot be scheduled to modulator till the modulator finishes its current task. All PEs can process only one task at a time. If modulator cannot accept the new task then the output generated in interleaver cannot be transferred to Input buffer of Modulator. Hence the back pressure gets created for interleaver and it gets transferred to encoder, scrambler and to header. The back pressure is created by the slowest PE, results into rapid increase of t_{ta} and $t_{dti.}$ In case of $t_{ta,}$ the task stays dormant in the TSQ till the PE becomes free to accept the new task. Also the data transfer cannot be initiated, if the PE is reading the input buffer and new data transfer request comes for the same input buffer.

Hence the scheduling scheme implemented is not responsible for increased t_{ta} and $t_{dti.}$ The scheduling scheme is keeping the PEs busy for maximum amount of time, and also trying to balance the pipeline, indicating success of the scheduling scheme. In order to avoid this bottleneck, PEs should be fast enough to accept new task from VFP controller. Another possible solution is to replicate the PEs so that when one PE is busy processing the data, other PE can work parallel on another set of input data.

7. Conclusion

WiNC2R is a cognitive radio platform which is easily programmable via software and can be reconfigured dynamically to support multiple protocols. The verification and validation of such a complex platform is necessary for the success of the platform. Choice of verification methodology dictates the success of the product. This thesis first defined the need of a layered testbench in order to verify or validate WiNC2R platform. Thesis focuses on choice of verification methodology and explains why OVM is based verification methodology. This thesis explained various OVM phases provided by OVM along with the small examples to implement OVM based verification environment. The later chapter explained how OVM based verification environment is configured to cater the needs of the WiNC2R platform. Constrained random stimulus generation using OVM for WiNC2R was explained. Due to constrained random stimulus, and various OVM based sequences, WiNC2R platform was validated under very realistic scenarios. This helped in finding the hidden bugs from the system. The OVM based verification environment with golden PEs helped in fixing the system related parameter like guard time. System programmer will get benefitted once he has fair idea about guard time budgeting.

VFP overhead was characterized under single and two flows case at 6, 12, 24 MBPS. VFP overhead was characterized in order to understand the performance and cost penalties of added flexibility provided by WiNC2R platform. Characterization of VFP overhead showed that most of the factors involved in VFP overhead remain constant. Analysis of the VFP overhead showed the reason for increased t_{ta} and $t_{dti.}$ PE and Bus utilization were computed along with the VFP overhead. The analysis of PE Utilization, Bus Utilization and VFP overhead confirmed the success of the scheduling scheme and indicated the need for faster PEs. This gives a very important feedback to system architect in order to improve the architecture.

Hence OVM based verification environment was used for

1. Validation of the VFP controller’s scheduler
2. Extracting PE Utilization
3. Extracting Bus Utilization
4. Computing VFP Overhead
5. Fixing system parameter like guard time in case of sync tasks.

The current implementation of OVM based verification methodology will act as a tool to extract the necessary performance from the WiNC2R system. Log files are created while extracting the performance from the system, which gives a proper timeline of events happening inside WiNC2R. The automation provided in extracting the performance analysis helps system architect in terms of reducing the debugging time and quickly providing very good feedback about the system.

8. Future Work

1. More focus should be given on functional verification of the system which will make sure that the system is bug free.
2. System should be tested for more flows and VFP overhead should be characterized under more flows. Also PE and Bus utilization should be computed in order to understand the effect of the varying throughput conditions.
3. By using fast PEs, variation of VFP overhead in terms of t_{ta} and t_{dti} should be measured using the existing setup.
4. Design changes can also be done to compensate for increased t_{ta} and t_{dti} by replicating slower PEs. After design changes, performance should be extracted from the system to understand the success of the new scheme.

References

1. *The WINLAB Network Centric Cognitive Radio Hardware Platform- WiNC2R.* Zoran Miljanic, Ivan Seskar, Khanh Le, Dipankar Raychaudhuri. Orlando, Florida : Springer, 2007. Proc CrownComnn.

2. *Resource Virtualization with Programmable Radio Processing Platform.* Zoran Miljanic, Predrag Spasojevic. ICST (Institute for Computer Sciences Social-Informatics and Telecommunications Engineering), Brussels, Belgium : s.n., 2008. Proceedings of the 4th Annual international Conference on Wireless internet (Maui, Hawaii, November 17 - 19, 2008).

3. ARM. ARM Specification. [Online] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html.

4. Onkar Sarode, Zoran Miljanic, Predrag Spasojevic, Khanh Le. *prograammable radio form.* New Brunswick : Winlab, 2009.

5. OVM White Paper. *OVM WORLD.* [Online] 2008. http://www.ovmworld.org/downloads-files.php.

6. Spear, Chris. *SystemVerilog for Verification: A guide to Learning the Testbench Language Features.* New York, NY, USA : Springer, 2006.

7. IBM. Processor Local Bus. [Online] https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4.

8. Satarkar, Sumit. *Performance Analysis of the WiNC2R Platform.* New Brunswick : Rutgers University, 2009.

9. Joshi, Madhura. *system integration and perf. evaluation of winc2r platform for 802.11a like protocol.* New Brunswick : Rutgers University, 2010.

10. Bergeron, Janick. *Writing Testbenches Functional Verification of HDL Models.* Norwell, MA, USA : Kluwer Academic Publishers, 2003.

11. Iman, Sasan. *Step by Step Functional Verification with SystemVerilog and OVM.* Santa Clara, CA, USA : Hansen Brown Publishing Company, 2008.

12. Purisai, Rangarajan. How to choose a verification Methodology. *EE Times.* [Online] 07 09, 2004. http://www.eetimes.com/news/design/features/showArticle.jhtml?articleID=22104709.

13. Fitzpatrik, Tom. OVM at DAC 2009: Moscone Center. *OVM WORLD.* [Online] http://www.ovmworld.org/tradeshows.php.

14. Synopsys. Verification Methodology Manual for SystemVerilog. *VMM SV.* [Online] http://vmm-sv.org/.

15. Cadence, Mentor Graphics. Open Verification Methodology. *OVM WORLD.* [Online] http://www.ovmworld.org/.

16. Overview OVM. *OVM WORLD.* [Online] http://www.ovmworld.org/overview.php.

17. OVM User Guide. *OVM WORLD.* [Online] 2008. http://www.ovmworld.org/downloads-files.php.

18. Onkar Sarode, Khanh Le, Zoran Miljanic, Predrag Spasojevic. *WiNC2R Platform Centralized Unit Control Module Architecture.* WINLAB, Rutgers University. New Brunswick : s.n., 2009.

19. OVM Reference Manual. *OVM WORLD.* [Online] 2008. http://www.ovmworld.org/downloads-files.php.

20. cadence, Mentor Graphics. Overview OVM. *OVM WORLD.* [Online] http://www.ovmworld.org/overview.php.

21. Document, WINLAB Internal. *FU Architecture .* New Brunswick : s.n., 2008.