VERIFICATION OF D3G MODULE FOR WIRELESS CHIP IN 65nm

Major Project Report (Sem. III - IV)

Submitted in fulfillment of the degree of

Master of Technology

IN ELECTRONICS & COMMUNICATION ENGG.
(VLSI DESIGN)

By
Snehal Vora
(06MEC017)

Under the Guidance of

Girraj Agrawal
Manager(IPT)
Freescale Semiconducor India Pvt. Ltd.
Noida.

Prof. Usha Mehta
EC Department,
Instituteof Technology

Nirma University
Ahmedabad

Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382481

CERTIFICATE

This is to certify that the Major Project Report entitled "***Verification Of D3G module for wireless chip in 65nm***" submitted by ***Mr. Snehal Vora (06MEC017)***, towards the fulfilment of the requirements for the Semester-IV of the degree of **Master of Technology** in **Electronics & Communication (VLSI Design)** of **Nirma University of Science & Technology**, is the record of work carried out by him under our supervision and guidance during the period from **September 17, 2007 to Apr 22, 2008.**

Date :

Project Leader
Mr. Girraj Agrawal
Freescale Semiconductor Ltd.
Noida

Co-ordinator – M.Tech(VLSI Design)
Dr. N. M. Devashrayee
Institute of Technology,
Nirma University-Ahmedabad.

HOD - E&C Dept.
Prof. A. S. Ranade
Institute of Technology,
Nirma University-Ahmedabad.

Internal Project Guide
Prof. U. S. Mehta
Institute of Technology,
Nirma University-Ahmedabad.

ACKNOWLEDGEMENT

Sometimes our light goes out,
but is blown into flame by another human being.
I owe deepest thanks to those
who have rekindled this light.

"Verification Of D3G module for Wireless chip in 65 nm" is my project as Major Project for M.Tech (VLSI Design). I take this opportunity to express my gratitude to all those who have been instrumental in making my project successful.

I am indebted to **Mr. Ganesh Guruswami**, Contry manager, Freescale Semiconductor India Pvt. Ltd., for giving me an opportunity to work in such a vibrant organization. I am grateful to **Mr. Anand**, Manager, HR & D, coordinator of the Training program, for his co-operation. I am grateful to **Mr. Rajan Kapoor, Manager, CPG** and **Mr. Girraj Agrawal, Manager, IPT** for extending their support and providing us with requisite infrastructure for training at Freescale Semiconductor.

I pay my special regards to **Ms. Monika Goyal** for giving me an opportunity to work on this project and for her esteemed guidance, support and encouragement to achieve the aim, throughout the project.

I am thankful to **Gaurav, Shobit, Surbhi, Chirag, Saurabh** and others in the group for their guidance and timely help, their critical observations and encouragement that made successful completion of this work possible.

I am thankful to **Prof. Usha Mehta**, **Asst. Professor, EC Dept, IT(NU), Nirma University, Ahmedabad**, for her guidance.

I extend my thanks to **Prof. N.M. Devashrayee, Head and Course Coordinator – M.Tech (VLSI Design), IT(NU), Nirma University, Ahmedabad** for his unconditional support and encouragement. I am also thankful to all the faculty members of the department for this direct or indirect support. After undergoing this project training, I can confidently say that this experience has not only enriched me with technical knowledge, but also with the deep insight into quality concepts that are required to be a successful professional.

Last but not the least, I pay my sincere gratitude to my parents and family members due to whom today I could reach this stage. I would also like to pay my sincere thanks to my friends **Ankit, Ankush, Anand, Chandresh, Hardik, Mehul and mine seniors** who have helped me a lot not only in project but in every dimensions during this 8 months, thanks to them for being my family away from home.

Snehal Vora
06MEC017
M.Tech (VLSI Design)
IT, Nirma University.
Ahmedabad

Index

List of Figures

Abstract

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System On Chip (SOC) designs, verification consumes about 70% of the design effort. The number of verification engineers is usually twice the number of design engineers. When design projects are completed, the code that implements the test benches makes up to 80% of the total code volume. Verification of design is done at various levels of design phase

The project scope for verifying the D3G module. This module implements DigRF3G interfaces on Baseband IC. It is used to transfer data and control information between RFIC and BBIC. Verification of module is done using VERA®. The prerequisite of the undertaken project is to thoroughly understand the module under study and VERA the verification tool. In the first phase of the project task assigned was to deal with learning the verification tool, debugging various test cases. It demanded technical in-depth of each test case, followed by locating-defining the problem, find the cause and report it. The second phase of the project task involved random verification for functional coverage of the module that stresses demands for the conceptual understanding of the process of random verification. The next part pf the project involved Common Power Format activity for the same module, inorder to transfer the IP from 65nm to 45nm. In the fourth phase of the project task assigned was to perform the code coverage activity for the same module.

This volume describes various stages of the training cum project work. Apart from the above, it discusses about the VERA implementation and issues, the common power format and code coverage. Of course, only a few of the codes and algorithms are presented maintaining the confidentiality of the organization.

Company Profile

After more than 50 years as part of Motorola, Freescale started a new life as a stand-alone company in July 2004. Since then, under the leadership of Chairman and CEO Michel Mayer, the company has focused on improving financial performance, reenergizing the culture and building a global brand.

Freescale started its operation in India in 1998 with a mission of making India a center of excellence in SoC Integration & IP design. Over the years, the operations have gone from strength to strength to emerge as a team of professional engineers working tirelessly to move up the value chain and become a vital part of global network of design teams which form the core of Freescale business operations.

Today, India is one of Freescale's two Centers of Excellence in the region (China being the other) and is emerging as the largest design center outside of the U.S.

Soon, the India Design Center will be one of the only design centers outside the U.S. developing and delivering R&D products to all the three major business units--Transportation and Standard Products Group (TSPG), Networking and Computing Systems Group (NCSG) and Wireless and Mobility Systems Group (WMSG).

Comprehensive SoC Technology

The IDC is involved in developing SoCs supported by a comprehensive application-development toolkit for various applications:

- Wireless communications
- Networking
- Automobile electronics
- Standard DSP devices
- Microcontroller product families
- Microprocessor product families

The Freescale Center of Excellence (CoE) for Wireless and Mobility software located in Bangalore is envisioned to become one of the largest software centers for Freescale outside of the U.S.

Pervasive Innovation

Freescale may be one of the largest companies that people touch every day, but have never heard of. It has shipped more than 17 billion semiconductors, which can be found in everyday name brands:

- Motorola cell phones
- Sony electronics
- Whirlpool appliances
- Logitech keyboards and mice
- Life fitness cardiovascular and strength training equipment
- Cisco routers
- Bose Acoustic Wave radios
- Trane heating and cooling equipment
- Mercedes, BMW, Ford, Hyundai and General Motors vehicles Market Leadership

Freescale is a leader in many markets it serves:

- No. 1 in automotive semiconductors -- Gartner
- No. 1 in communications processors -- Gartner
- No. 2 in overall microcontrollers -- Gartner
- No. 2 in digital signal processors -- Forward Concepts
- No. 4 in wireless handset radio frequency microprocessors -- iSuppli

Chapter 1

Introduction to Baseband SOC

The Baseband SOC is a mid-tier implementation of the Mobile Extreme Convergence Architecture (MXC) addressing the UMTS mass market. It provides the baseband (modem) and application processor functions for the MXC. This chipset is intended to support the following radio features:

- Multiple Wireless Communications Standards
  - HSDPA 3.6Mbps (DL)
  - HSUPA 2.0Mbps (UL)
  - Concurrent HSDPA 3.6Mbps and HSUPA 1.48Mbps
  - WCDMA - FDD: Concurrent 384Kbps uplink and 384 Kbps downlink
  - EDGE Class 12
  - GSM/GPRS
- 3GPP WCDMA bands I through VI and W900
- Up to 3 active WCDMA bands (2H+1L)
- Quad-band GSM/EDGE (GSM850, EGSM, DCS, PCS)
- External connectivity via Bluetooth and WLAN
- Video Playback H.264/MPEG4 D1 at 30 fps
- Video Capture H.264/MPEG4 D1 at 30fps
- Full-duplex video (simultaneous encode and decode) H.264/MPEG4 CIF at 30fps
- Open OS support (Linux, Symbian, WinCE)

Major subcomponents of the Baseband SOC are described below.

1.1 Application Processor (AP)

The application processor (AP) and its domain are responsible for running the operating system and applications software, providing the user interface, and supplying access to integrated and external peripherals. The look and feel of the product depends on the

software running on this processor, ultimately tying market acceptance to the availability of a wide variety of off-the-shelf, third-party software and development tools.

1.2 Shared Memory Interface

The external memory subsystem represents a significant investment in chipset cost and board area. High-performance processor cores require a significant amount of bus bandwidth to achieve their maximum potential. The challenge here is to maximize the performance while minimizing the bandwidth.

1.3 Shared Domain

The shared domain is composed of the connectivity peripherals, security peripherals, a Smart DMA Engine (SDMA), and a number of miscellaneous modules. For maximum flexibility and to minimize module duplication, some peripherals are shared between the two processors (AP and BP).

Chapter 2

Introduction to D3G

D3G module implements DigRF3G interface on Baseband IC (BBIC). D3G module is used to transfer data and CTRL information between RFIC and BBIC. D3G module has the capability of transmission of 2G data, 3G data, ICLC, CTRL data, Time Accurate Strobes(TAS) and reception of normal and diversity 2G data, normal and diversity 3G data, CTRL, Unsolicited frames. D3G supports multiplexed diversity reception of 2G and 3G data.

The block diagram of DigRF3G module (D3G) is shown in the Figure 2-1, D3G Block Diagram. The main submodules for the D3G module are:

1. Transmitter (TX)
2. TX RAM and TX RAM controller
3. Receiver (RX)
4. RX RAM and RX RAM controller
5. 3G TX Data Handler
6. 3G RX Data Handler
7. TAS FIFO
8. CTRL FIFO

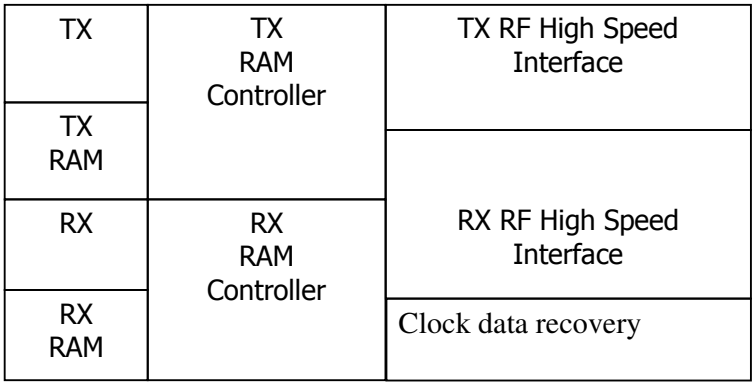

Figure 2.1 D3G Block Diagram

2.1 Transmitter (TX)

The TX block is responsible for transmission of 3G Data, 2G Data, RFIC Control (CTRL), TAS and ICLC frames. Upon receiving valid transmit requests, TX block arbitrates and schedules them to transmit. It fetches the respective data from the TX RAM This block has the following functionality:

1. Arbitration
2. Framing
3. Parallel to Serial Conversion
4. CTS Mode Support
5. Sleep Mode Support

2.2 TX RAM and TX RAM Controller

The TX of D3G module has two RAMs which store data to be transmitted to RFIC. A unified TX RAM stores 2G data, CTRL Data and TAS information to be transmitted. The Receiver (RX) block in D3G receives serial data from RFIC. RFIC sends the data in the form of frames as per the protocol defined in the DigRF3G Standard. RX block receives the frame, decodes it and takes appropriate actions on the data received,

2.3 Receiver (RX)

RX block consists of the following sub-blocks.

- Clock data recovery
- Synchronizer Block
- RX Controller Block

2.4 RX RAM and RX RAM Controller

There are two RX RAM blocks in D3G which are used for temporary storage of data received from the RFIC, prior to it being read out.

2.5 Modes of Operation

2.5.1 Normal Operating Modes

D3G transmits information to the RFIC depending on relevant triggers. The TAS, 2G data, 3G data and CTRL data triggers are received from the Timers. ICLC frame transmit is triggered by writing in the relevant register of D3G. The transfer requests are arbitrated and scheduled by D3G. When any trigger is scheduled, the data is fetched from respective FIFOs and sent to RFIC serially.

2.5.2 Multiplexed Diversity Mode

D3G module supports the reception of 2G and 3G data under the Multiplexed Diversity mode. Both Normal and Diversity RX Data are received by same Line Receiver and sampled through same RX Interface. Receiver differentiates 2G and 3G diversity data with 2G and 3G normal data on the basis of the header of the received frame.

2.5.3 CTS Mode

D3G module supports CTS mode of operation RFIC sends CTS information to D3G. RFIC sends CTS bit as 0, whenever RFIC is not ready to receive data. When it is ready to receive data sends a frame with CTS bit asserted. RFIC must send CTS bit correctly in every frame.

2.5.4 Low Power Modes

Sleep Mode

Sleep modes are implemented in order to save power during inter-frame gaps that are long compared to the frame durations but not long enough to allow the interface(s) or high-speed clock generators to be powered down completely. These modes shall be available at all interface operating speeds.

TX Sleep Mode

D3G TX block hardware detects the sleep mode condition automatically. When D3G decides that TX block can enter into sleep mode, it will drive ‘1’ on the interface at the end of the current frame transfer.

RX Sleep Mode

After detecting “1” at the end of the current frame, the D3G RX block enters sleep mode.

Chapter 3

VERA Introduction

Vera® is an integral part of the Synopsys Smart Verification platform. It is the comprehensive testbench automation solution for module, block and full system verification. The Vera testbench automation system is based on the OpenVera™ language. This is an intuitive, high-level, object-oriented programming language developed specifically to meet the unique requirements of functional verification. With Vera, it is easy to quickly model the target environment at a high level of abstraction while automatically generating constrained random stimulus. Vera's Constraint Solver Engine is used to automatically generate tests that mimic "real-life" stimuli with the application of the verification engineer's constraints. Constrained random stimulus enables the detection of a wide range of bugs including functional and corner cases. These self-checking tests written in Vera eliminate the need to analyze manually simulation waveforms and reports. Its built-in dynamic coverage analysis provides instant coverage feedback, facilitating the efficient generation of high coverage stimuli.

Key benefits of Vera are:

- It enables the generation of high coverage, constraints-driven random stimulus generation.
- It enables scalable and re-usable testbenches with Vera – the open source Hardware Verification Language.
- It leverages testbenches across all HDLs including VHDL, Verilog, and SystemC.
- It improves simulation efficiency with intelligent, reactive test generation.
- It increases simulation through the usage of distributed processing.
- It accelerates verification ramp up with a large offering of Vera Verification IP.
- It leverages high performance, Vera assertions, temporal_ assertion technology to accelerate the development of monitors and checkers.

3.1 The Vera Testbench Flow

A Vera testbench can drive the design under test (DUT) effectively for verification, abstracting the stimulus generation-response checking mechanism for productivity. The

DUT can be written in one or more of the existing Hardware Design Languages currently available: Verilog, VHDL, or SystemC.

3.1.1 Verilog-based Flow

There are numerous ways to construct a Vera testbench. This section outlines a simple flow for Verilog-based designs. The Vera template generator is used here to start the process. Figure 3.1 illustrates the components of the Vera testbench.

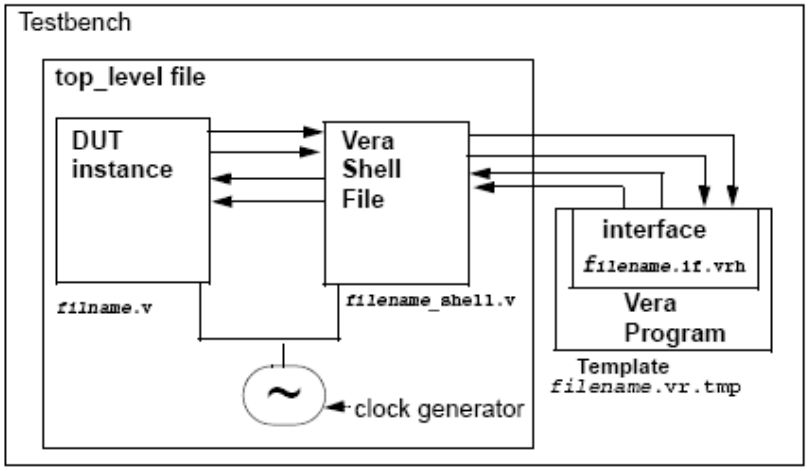

Figure 3.1 Template Generator and Components of a Vera Testbench

The procedure for creating a Vera testbench for Verilog-based designs is as follows:

1. The first step is to invoke the template generator:

    vera -tem -t *filename* -c clock *filename*.v

2. Rename the template Vera program file from "*filename*.vr.tmp" to"*filename*.vr".
3. Add code to the Vera program file (*filename*.vr).
4. Compile the Vera program file:

    vera -cmp -vlog *filename*.vr

This creates two files:

a. an object code file .vro file

b. a *filename*_shell.v file, which corresponds to the "Vera Shell File" component of the top_level depicted in Figure 3-1. The Vera *filename*.shell.v file acts as the interface between the Vera program and the DUT. Do *not* edit this file.

5. Create the simv executable:

vcs -vera *filename*.v *filename*_shell.v \ *filename*.test_top.v

6. Run the simulation:

simv +vera_load=*filename*.vro

Example

This example starts with the following design under test (an adder):

```
module adder(in0, in1, out0, clk);
input[7:0] in0, in1;
input clk;
output[8:0] out0;
reg[8:0] out0;
always@(posedge clk)
begin
out0 <= in0 + in1;
end
endmodule
```

1. The first step is to invoke the template generator:

vera -tem -t adder -c clk adder.v

which creates the following files:

a. The interface file (adder.if.vrh):

```
// adder.if.vrh
#ifndef INC_ADDER_IF_VRH
#define INC_ADDER_IF_VRH
interface adder{
output [7:0] in0 OUTPUT_EDGE OUTPUT_SKEW;
output [7:0] in1 OUTPUT_EDGE OUTPUT_SKEW;
input [8:0] out0 INPUT_EDGE #-1;
input clk CLOCK;
} //end of interface adder
#endif
```

b. The test_top file (adder.test_top.v):

```
module adder_test_top; //adder.test_top.v
parameter simulation_cycle = 100;
reg SystemClock;
wire [7:0] in0;
wire [7:0] in1;
wire [8:0] out0;
assign clk = SystemClock;
vera_shell vshell( //Vera shell file
.SystemClock(SystemClock),
.adder_in0 (in0),
.adder_in1 (in1),
.adder_out0 (out0),
.adder_clk (clk),
);
‘ifdef emu
/*DUT is in emulator, so not instantiated here*/
‘else
adder dut(
.in0 (in0),
.in1 (in1),
.out0 (out0),
.clk (clk),
);
‘endif
initial begin //clock generator
SystemClock = 0;
forever begin
#(simulation_cycle/2)
SystemClock = ~SystemClock;
end
end
endmodule
```

c. The Vera file (adder.vr), which is copied from the Vera template file (adder.v.tmp):

```
// adder.vr (copied from the template adder.v.tmp)
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
// define any interfaces, and verilog_node here
#include "adder.if.vrh"
// define any ports, binds here
// declare any external tasks/classes/functions here
// declare any hdl_task(s) here
// declare any class typdefs here
program adder_test
{ // start of top block
// define any global variables here
// Start of adder_test
// Type your test program here:
// Example of drive:
// @ 1 adder.in0 = 0;
// Example of expect:
// @ 1,100 adder.out0==0;
// Example of what you are to add:
@ (posedge adder.clk); // line up to first clock
@0 adder.in0=8'hff; // drive "ff" to the design
@2 adder.out0 == 9'h1fe; // expect "1fe" from the design
} // end of program adder_test
// define any tasks/classes/functions here
```

d. The Vera shell file (adder_shell.v; do *not* edit this file):

```
// adder_shell.v
module vera_shell(
SystemClock,
adder_in0,
```

```
    adder_in1,
    adder_out0,
    adder_clk
    );
    input SystemClock;
    output [7:0] adder_in0;
    output [7:0] adder_in1;
    input [8:0] adder_out0;
    inout adder_clk;
    //Nest which VMC will reference
    // System clock:SystemClock
    wire SystemClock;
    // Other components of this file are not listed here
    ...
    endmodule
```

2. The next step is to compile the adder.vr file, which creates the adder_shell.v and adder.vro files:

```
% vera -cmp -vlog adder.vr
```

3. Now you can create the simv executable:

```
% vcs -vera adder.v adder_shell.v adder.test_top.v
```

4. And finally, you can run the simulation:

```
% simv +vera_load=adder.vro
```

Figure 3.2 Testbench Flow using the Template Generator

Chapter 4

Introduction to VERA Base Class

Verification environments that are derived from reusable verification components improve verification effectiveness and increase productivity of the verification engineer. Since a design is verified at multiple levels of integration, the definition and implementation of verification components must rely on requirements that are accurate, complete, and concise. Complete and accurate requirements promote structured design specifications, consistent implementations, and maximal component reuse.

The SPS Base Class Library is the foundation for achieving testbench component reuse. The SPS Base Class Library defines a set of classes, based on the Vera hardware verification language (VeraHVL), that promotes the following:

- Consistent component interfaces
- Modular implementations
- Testbench component reuse

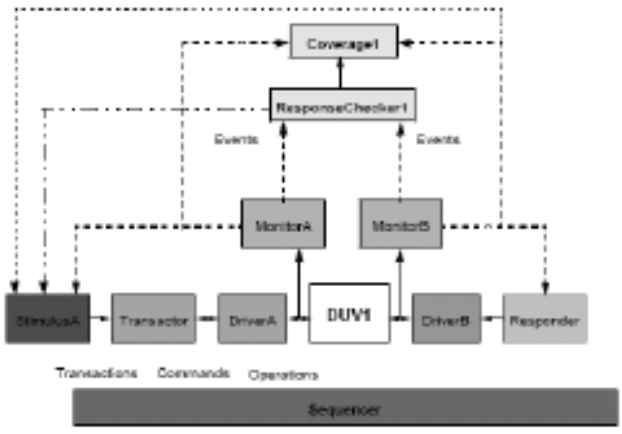

Figure 4.1 Modular Verification Environment

Consider the modular verification environment shown in Figure 4.1. Each component performs a specific role in the verification of the design-under-verification (DUV). Each component has a unique identity, and components interact amongst themselves and with the DUV by issuing commands, transactions, and Events. The modular verification environment consist of the following components;

• Sequencer - sequences each testbench component through the various testcase phases

• MonitorA - observes and checks DUV protocol on interface A; abstracts signal transitions into higher-level transaction Events and communicates those Events to ResponseCheckers and CoverageObjects, and possibly StimulusObjects

• MonitorB - observes and checks DUV protocol on interface B; abstracts signal transitions into higher-level transaction Events and communicates those Events to ResponseCheckers and CoverageObjects, and possibly StimulusObjects

• DriverA - uses commands, received from the Transactor to issue transactions on the DUV interface A. May also respond to DUV transactions

• DriverB - uses commands, received from the Responder to issue transactions on the DUV interface B. May also respond to DUV transactions

• Transactors - translates high-level transaction request, received from Stimulus, into low-level operations with the Driver

• StimulusA - creates and issues commands, to the Transactor, that cause transactions on the DUV interface A

• Responder - subscribes to Events from Monitors and uses Drivers to initiate appropriate transactions in response to Events

• Coverage - receives coverage Events from Monitors and ResponseCheckers in order to collect coverage metrics

• ResponseChecker (RSC) - receives transaction Events from MonitorA and MonitorB in order to check DUV responses to issued transactions

Modular testbench components have and/or use specific behavior that improves integration and promotes the re-use of components and Stimulus between different testbenches, and the re-use of components at the module and system level.

4.1 SPS Base Class Summary

The SPS Base Class contains a set of class definitions that are used for developing a modular, consistent, and reusable verification environment. Each class is intended to encapsulate concepts and specific functionality within a set of independent modular testbench components. Each testbench component class has a specific set of characteristics and plays a specific role within the testbench. The SPS base class hierarchy is shown in Figure 4.2 as a unified modeling language (UML) diagram. Figure 4.2 shows that by inheritance all classes inherit from SPSLoggingBase and as a result,

inherit a common set of message logging/reporting functions. Figure 4.2 also shows that all component classes (Monitors, Drivers, etc.,) inherit from the base type SPSComponentBase. The user must only inherit their testbench components from one of the italicized classes (see Figure 4.2).

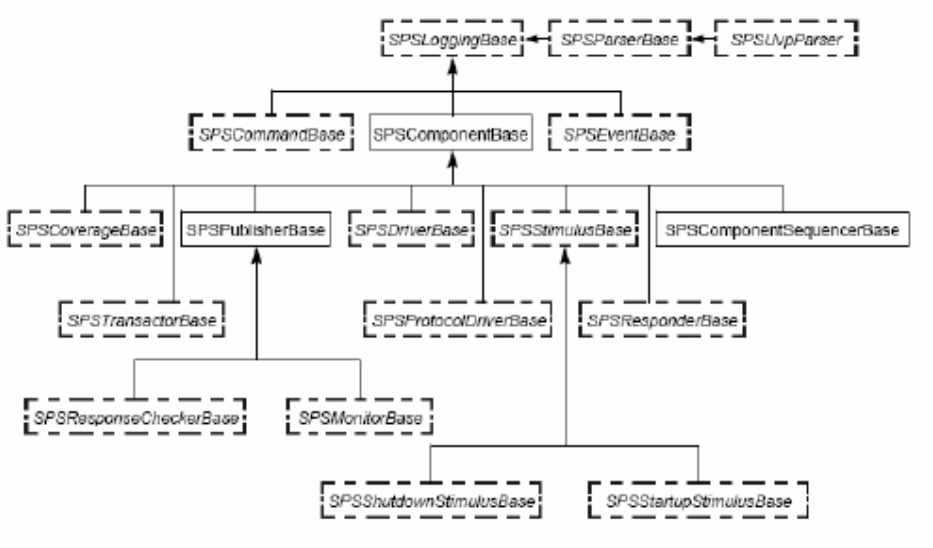

Figure 4.2 SPS Base Class Hierarchy

4.2 Testbench Component Sequencer

SPSComponentSequencerBase is derived from SPSComponentBase and provides functions required for sequencing components through the various test case phases (restart, configure, execute, final_check, final_report). The component sequencer class provides a complete implementation for controlling testbench component activities at appropriate points in the simulation in order to achieve an orderly and collective application of Stimulus during a succession of test cases.

4.3 Testbench Component Communication

SPSEventBase is derived from SPSLoggingBase. All user-defined Events must be derived from SPSEventBase. Events are used to capture the transitions and the state on an interface (for example, bus_grant and bus_idle). Events are created by Monitors and ResponseCheckers. ResponseCheckers subscribe to the Event published by Monitors. StimulusObjects subscribe to the Event published by Monitors and/or ResponseCheckers. SPSCommandBase is derived from SPSLoggingBase. All user-defined commands must

be derived from SPSCommandBase. Commands are created by StimulusObjects and are used to describe complete transactions such as read or write. StimulusObjects issue commands to Drivers.

4.4 Testbench Drivers and Monitors

SPSDriverBase is derived from SPSComponentBase. All user-defined Drivers must be derived from SPSDriverBase. Drivers attach to the signal interface of a DUV and process and translate commands issued from Stimulus into actual interface signal transitions to create DUV transactions.

SPSProtocolDriverBase is derived from SPSComponentBase. All new user-defined Drivers must derive form SPSProtocolDriverBase. SPSProtocolDriverBase supports a non-blocking interface that is better suited to pipelined models. SPSMonitorBase is derived from SPSPublisherBase. All user-defined Monitors must be derived from SPSMonitorBase. Monitors attach to the signal interface of a DUV and are responsible for observing and checking interface protocol. Monitors are also responsible for observation and translation of signal transitions into Events and the subsequent publication of Events for Responsecheckers, Coverage, and Stimulus.

4.5 Transactors

SPSTransactorBase is derived from SPSComponentBase. All user-defined Transactors must derive from this class. Transactors receive commands from StimulusObjects and translate transaction requests into operations with the Driver.

4.6 Testbench ResponseCheckers

SPSResponseCheckerBase is derived from SPSPublisherBase. All user-defined ResponseCheckers must be derived from SPSResponseCheckerBase. ResponseCheckers enable correlation of DUV behavior across multiple interfaces using Events published by various Monitors. The ResponseChecker subscribes to Events to determine the expected DUV behavior, correlate system behavior, and identify unexpected system behavior. ResponseCheckers also publish Events for Stimulus and Coverage.

4.7 Testbench Responder

SPSResponderBase is derived from SPSComponentBase. Responders listen to Events from Monitors, and issues transactions to Drivers. A Responder may correlate responses to Events on several SignalInterfaces. Therefore, it may listen to Events from several Monitors and issue transactions to several Drivers.

4.8 Testbench Stimulus

Three types of Stimulus base classes exist, including:

SPSStimulusBase - derives from SPSComponentBase; all StimulusObjects that issue commands that cause interface transactions like read or write should derive from this class.

• SPSStartupStimulusBase - derives from SPSStimulusBase; Startup StimulusObjects are used to issue SoC-specific startup commands such as reset and configuration.

• SPSShutdownStimulusBase - derives from SPSStimulusBase; Shutdown StimulusObjects are used to issue SoC-specific commands such system shutdown.

4.9 Testbench Coverage

SPSCoverageBase is derived from SPSCoverageBase. Coverage is used to measure the effectiveness of testbench Stimulus. Coverage measures what has been tested, identifies test cases that are missing from the Stimulus, and checks the random generation of the Stimulus. CoverageObjects subscribe to Events from Monitors and ResponseCheckers and use these Events to collect metrics for each simulation. Interesting coverage points must be identified, and appropriate Events defined to satisfy goals of the verification plan.

4.10 Unified Memory Model

The unified memory model (UMM) provides the functions required to manage and model a memory system.

Chapter 5

Random Verification and Functional Coverage Using VERA

There are two class of functional verification : a) Directed b) Random

In directed verification verification engineer writes the testbench according to his understanding of the specification or the functionality he thinks is important. So with directed verification only the scenarios which are decided by the engineer are tested and it may happen that certain corner cases are ignored.

In random verification approach the stimulus to duv is random. So each run of random verification puts the duv in the different scenario. So it may happen that certain cases which even not thought to occur may also got covered. It resembles the real time situation.

5.1 Random Verification with VERA

5.1.1 Random Number Generation

random()

The random() function returns a 31-bit pseudo-random positive integer value from the current thread RNG.

Syntax

function integer random([integer *seed*]);

seed

is an optional argument that determines which random number sequence is generated. The seed can be any valid Vera expression, including variable expressions. The random number generator generates the same number sequence every time the same seed is used.

The random() function takes an optional seed argument that is used to call srandom() before computing the random value. In this case the system behaves as if srandom(seed) had been called before random(). The random number generator is deterministic. Each time the program executes, it cycles through the same random sequence. You can make this sequence non-deterministic by seeding the random() function with an extrinsic

random variable, such as the time of day. Since random() always returns a positive number, the sign bit is 0. Only the 31 least significant bits are random.

urandom()

The urandom() function returns a 32-bit pseudo-random unsigned bit vector value from the current thread RNG.

Syntax

function bit[31:0] urandom([integer *seed*]);

The urandom() function takes an optional seed argument that is used to call srandom() before computing the random value.

rand48()

The rand48() function generates an integer (random) number based on the lrand48 algorithm.

Syntax

function integer rand48([integer *seed*]);

Since rand48() always returns a positive number, the sign bit is 0. Only the 31 least significant bits are random.

urand48()

The urand48() function generates an unsigned 32-bit random number based on the mrand48 algorithm.

Syntax

function bit [31:0] urand48([integer *seed*]);

Random Variables

Class variables can be declared random using the rand keyword.

Syntax

rand variable;

randc variable;

• Vera can randomize scalar variables of type integer, bit, and enumerated type. Bit variables can be any size.

• Arrays can be declared rand in which case all of their member elements are treated as rand.

5.1.2 Constraint Based Randomization

This section introduces basic concepts and uses for generating random stimulus within objects. The Vera language uses an object-oriented method for assigning random values to the member variables of an object subject to user-defined constraints.

Example

```
class Bus{
rand bit[15:0] addr;
rand bit[31:0] data;
constraint word_align {addr[1:0] == ‘2b0;}
}
```

The Bus class models a simplified bus with two random variables: addr and data, representing the address and data values on a bus. The word_align constraint declares that the random values for addr must be such that addr is word-aligned (the low-order 2 bits are 0).

randomize()

The randomize() method is called to generate new random values for a bus *object*:

```
program test{
Bus bus = new;
repeat (50){
integer result = bus.randomize();
if (result == OK)
printf("addr = %16h data = %32h\n",
bus.addr, bus.data);
else
printf("Randomization failed.\n");
}
}
```

New values are selected for all of the random variables in an object such that all of the constraints are true (“satisfied”). In the program test above, a “bus” object is created and then randomized 50 times. The result of each randomization is checked for success. If the

randomization succeeds, the new random values for addr and data are printed; if the randomization fails, an error message is printed. In this example, only the addr value is constrained, while the data value is unconstrained. Unconstrained variables are assigned any value in their declared range.

Constraint Blocks

The values of random variables are determined using constraint expressions that are declared using constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. They must be defined after the variable declarations in the class, and before the task and function declarations in the class. Constraint block names must be unique within a class.

Syntax

constraint *constraint_name* {*contraint_expressions*}

constraint_name

is the name of the constraint block. This name can be used to enable or disable a constraint using the built-in constraint_mode() method.

constraint_expressions

are a list of expression statements that restrict the range of a variable or define relations between variables. A constraint expression can be any Vera expression, or can use the constraint-specific set operators: in, !in, and dist.

Variable Ordering

Vera assures that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of values have the same probability of being chosen). This important property guarantees that all value combinations are equally probable. Sometimes however, it is desirable to force certain combinations to occur more frequently. Consider this case where a 1-bit “control” variable (s) constrains a 32-bit “data” value (d):

Example

```
class B
{
rand bit s;
rand bit[31:0] d;
```

```
constraint c { s => d == 0; }
}
```

The constraint c says “s implies d equals zero”. Although this reads as if s determines d, in fact s and d are determined together. There are 2**32 valid combinations of {s,d}, but s is only true for {1,0}. Thus, the probability that s is true will be 1/2**32, which is almost never. Vera provides a mechanism for ordering variables so that s can be chosen independent of d. This mechanism defines a partial ordering on the evaluation of variables, and is specified using the solve keyword.

Example

```
class B
{
rand bit s;
rand bit[31:0] d;
constraint c { s => d == 0; }
constraint order { solve s before d; }
}
```

In this case, the order constraint instructs Vera to solve for s before solving for d. The effect is that s is now chosen true with 50% probability, and then d is chosen subject to the value of s. Accordingly, d == 0 will occur 50% of the time, and d != 0 will occur for the other 50%. Variable ordering can be used to force selected corner cases to occur more frequently than they would otherwise. The syntax to define variable order in a constraint block is:

solve *variable_list* before *variable_list*;

variable_list

is a comma-separated list of integral scalar variables or array elements

5.2 Coverage Overview

Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, have been exercised. It is, thus, a model of the verification plan. It measures whether interesting scenarios, corner cases, specification invariants, and other applicable DUT conditions captured as features of the test plan-- have been observed, validated and tested.

The key aspects of functional coverage are:

- It is user-specified, and is not automatically inferred from the design
- It is based on the design specification (i.e., its intent) and is thus independent of the actual design code or its structure. Since it is fully specified by the user, functional coverage requires more upfront effort from designers (someone has to write the coverage model).

Vera supports a functional coverage system. This system is able to monitor all states and state transitions, as well as changes to variables and expressions. By setting up a number of monitor bins that correspond to states, transitions, and expression changes, Vera is able to track activity in the simulation.

Each time a user-specified activity occurs, a counter associated with the bin is incremented. By establishing a bin for each state, state transition, and variable change that you want monitored, you can check the bin counter after the simulation to see how many activities occurred. It is, therefore, simple to check the degree of completeness of the testbench and simulation. Vera further expands this basic functionality to include two analysis mechanisms: open-loop analysis and closed-loop analysis.

• Open-loop analysis monitors the bins during the simulation and writes a report at the end summarizing the results.
• Closed-loop analysis monitors the bins during the simulation and checks for areas of sparse coverage. This information is then used to drive subsequent test generation to ensure satisfactory coverage levels.

Defining Coverage Models Using Coverage Groups

The coverage_group construct encapsulates the specification of a coverage model or monitor. Each coverage_group specification has the following components:

1. coverage_points
2. cross products
3. sampling event
4. set of state and/or transition bins.

• A set of coverage points. Each coverage point can be a variable or a DUT signal to be sampled. A coverage point can also be an expression composed of variables and signals. The variables may be global, class members, or arguments passed to an instance of the coverage_group.
• A sampling event (a general event not just a Vera sync event) that is used for synchronizing the sampling of all coverage points.

• Optionally, a set of state and/or transition bins that define a named equivalence class for a set of values or transitions associated with each coverage point.
• Optionally, cross products of subsets of the sampled coverage points (cross coverage).

The coverage_group construct is similar to an Vera class in that the definition is written once and is instantiated one or more times. The construct can be defined as a top-level (file scope) construct (referred to as standalone), or may be contained inside of a class. Once defined, standalone coverage is instantiated with the new() system call while embedded (contained) coverage groups are automatically instantiated with the containing object.

The basic syntax for defining a coverage_group is:
coverage_group *definition_name* [(*argument_list*)]
{
sample_event_definition;
[*sample_definitions;*]
[*cross_definitions*;]
[*attribute_definitions;]*
}

The syntax for defining an embedded coverage_group is:
class class_name
{
// class properties
// coverage
coverage_group *definition_name* .... (same as external).
// constraints

```
// methods
}
```

Example shows a standalone coverage_group definition and its instantiation.

Example

```
interface ifc
{
input clk CLOCK;
input sig1 PSAMPLE #-1;
}
coverage_group CovGroup
{
sample_event = @ (posedge CLOCK);
sample var1, ifc.sig1;
sample s_exp(var1 + var2);
}
program covTest
{
integer var1, var2;
CovGroup cg = new();
}
```

In this example coverage_group CovGroup defines the coverage model for global variable var1, the signal ifc.sig1, and the expression composed of global variables var1 and var2. s_exp is the name that will be used to refer to the sampled expression. The two variables and the expression will be sampled upon every positive edge of the system clock. The coverage_group is instantiated using the new() system call.

Chapter 6

Introduction to Common Power Format

With the rapid growth of the wireless and portable electronic markets, there is a constant demand for new technological advancements. This has resulted in more and more functionality being crammed into battery-operated products, increasing design and verification challenges for power management.

Challenges like how to minimize leakage power dissipation, or how to design efficient packaging and cooling systems for power-hungry IC's, or how to verify functionality of power shut-off sequences early in the design, are expected to get even worse with the continuous shrinking of process nodes using today's CMOS technology. Managing design and verification for power will be as critical, if not more than, for timing and area in today's IC design flow for portable consumer electronics.

Today's current power optimization and implementation techniques are leveraged at the physical implementation phase of the design. Certain advanced power management techniques like multiple power domains with power shut-off (PSO) methodology can only be implemented at the physical level (post synthesis). These advanced power management design techniques significantly change the design intent, yet none of the intended behavior can be captured in the RTL. This creates a big hole in the whole RTL to GDSII implementation and verification flow where the original RTL is no longer golden and cannot be used to verify the final netlist implementation containing the advanced power management techniques.

6.1 An Ad-Hoc Verification Solution

Faced with the challenge of verifying low power logic, an engineer may try to piece together an ad-hoc verification solution. If the domain being powered off is small, the verification may begin using the RTL netlist. Most likely, however, the number of memory elements will delay the verification effort until a gate level netlist is available.

RTL simulations require test bench elements to mimic the state retention, power switch off, and isolation. For isolation, the value to be isolated is forced on the primary outputs

by a test-bench element. In some cases, an isolation wrapper may be inserted by the design team and the test-bench element used to mimic isolation will not be required. In order to simulate state retention, the verification or design engineer must first identify all registers whose state is to be retained. A test-bench element must be created to store values from the registers when the retention sequence occurs. The values are deposited back to the registers by the test-bench element after restore sequence take places. For power switch off, all nodes must be forced to 'bX by test bench elements. One can see as the design grows this task becomes overwhelming.

Gate level simulations have isolation cells and state retention registers in the netlist. Therefore the test bench is not required to model these elements. The test bench is still required to model the power switch off unless a physical netlist with power and ground is used (and the verilog library elements support power down). Because there is no way to verify that the elements i.e., isolation and retention, identified in RTL are the same as are present in the gate netlist, the full verification suite must be re-run.

The need for gate level simulation means that design defects may not be identified until late in the cycle. An incorrect isolation value may not be found until days before tape-out. One may not identify a register that needed to be retained until after the masks have been created. With the cost of masks growing exponentially as the lithography shrinks, these can be extremely costly mistakes.

6.2 CPF Verification Solution

There are many problems with the ad-hoc solution: a large effort to create test bench elements, reliance on gate level simulations, and the numerous opportunity to introduce human error. A CPF based solution addresses many of these concerns and enables an enhanced verification solution.

The Common Power Format (CPF) is intended to address the current limitation in the design automation tool flow by enabling the capture of the designer's intent for advanced power management techniques. CPF provides support for all design and technology-related power constraints to be captured in a single file format for use throughout the RTL

to GDSII design flow including verification, validation, synthesis, test, physical implementation, and signoff analysis.

The automation enabled through CPF infrastructure support will be the answer to the growing power management design challenges faced by the industry. The introduction of CPF and its support will bring productivity gains and improved quality of silicon to designers without requiring any change to current legacy RTL.

A CPF file can be considered a netlist or specification of power related information. Just as the RTL verilog netlist is read by both the design and verification tools, the CPF file is also read by multiple tools in the flow such as the synthesis tool and the RTL simulator. The file contains a description of the low power architecture of design: which domains can be powered off, which values to isolate, and which registers to retain. In addition, the power control signals that govern when to retain, restore and isolate are contained within the file.

When a simulation is run with the CPF file, the simulator takes care of isolating the output of the powered off domains. It will also drive powered down nodes to ‘bX, and is responsible for retaining and restoring the correct register values. Written in Tcl, wild cards enable registers and isolation ports to be identified easily. This removes the burden of writing cumbersome test bench code that existed with the ad-hoc solution. With the weight lifted, verification engineers can concentrate their efforts on ensuring that the four stages of the low power modes are adequately tested.

The second major benefit of CPF is that it can be read by formal tools. While using CPF in simulation enables one to accomplish the same tasks faster and more efficiently then the ad-hoc solution, using it in a formal environment allows one accomplish things that were impossible in the ad-hoc flow. The logical equivalency between what used for RTL simulation, the netlist and CPF file, and the gate level netlist can be proven. This removes the need to re-run every simulation from the RTL level on the gate level netlist.

In order to get the most from a CPF based verification flow, the same file which is verified by simulation should be used by synthesis. The chance of human error being

introduced is decreased by having one central location for the power specification. An error present in the CPF used for synthesis is caught early by the RTL verification. In the ad-hoc solution, the error would not be found until gate simulations had begun. In addition, the CPF can be checked formally against low power rules at on the RTL level, rather than having to wait for a structural netlist.

6.2.1 Terminology

The Common Power Format (CPF) is a strictly Tcl-based format. All commands, command options, and object names are case sensitive. The CPF file is a power specification file. This implies that the functionality of the design does not change when sourcing a CPF file. The CPF file complements the HDL description of the design and can be used throughout the design creation, design implementation, and design verification flow.

The CPF file contains two categories of objects:

Design Objects

Design objects are objects that are being named in the description of the design which can be in the form of RTL files or a netlist. Design objects can be referenced by the CPF commands.

Design

The top-level module.

Instance

An instantiation of a module or library cell.

- Hierarchical instances are instantiations of modules.
- Leaf instances are instantiations of library cells.

Module

A logic block in the design.

Net

A connection between instance pins and ports.

Pad

An instance of an I/O cell.

Pin

An entry point to or exit point from an instance or library cell.

Port

An entry point to or exit point from the design or a module.

CPF Objects

CPF objects are objects that are being defined (named) in the CPF constraint file. CPF objects can be referenced by the CPF commands.

Isolation Rule

Defines the location and type of isolation logic to be added and the condition for when to enable the logic.

Level Shifter Rule

Defines the location and type of level shifter logic to be added.

Library Set

A set (collection) of libraries that was characterized for the same set of operating conditions. By giving the set a name it is easy to reference the set when defining operating corners.

Nominal Operating Condition

A typical operating condition under which the design or blocks perform.

Mode Transition

Defines when the design transitions between the specified power modes.

Power Domain

A collection of instances that use the same power supply during normal operation and that can be switched on or off at the same time. You can also associate boundary ports with a power domain to indicate that the drivers for these ports belong to the same power domain.

At the physical level a power domain contains

- A set of (regular) physical gates with a single power and a single ground rail connecting to the same pair of power and ground nets.
- The nets driven by these physical gates.
- A set of special gates such as level shifter cells, state retention cells, isolation cells, power switches, always-on cells, or multi-rail hard macros (such as, I/Os, memories, and so on) with multiple power and ground rails.

- At least one pair of the power or ground rails in these special gates or macros must be connecting to the same pair of power and ground nets as the (regular) physical gates connect to.

At the logic level a power domain contains

- A set of logic gates that correspond to the (regular) physical gates of this power domain.
- The nets driven by these logic gates.
- A set of special gates such as level shifter cells, state retention cells, isolation cells, power switches, always-on cells, or multi-rail hard macros (such as, I/Os, memories, and so on) that correspond to the physical implementation of these gates in this power domain.

At RTL a power domain contains

- The computational elements (operators, process, function and conditional statements) that correspond to the logic gates in this power domain .
- The signals that correspond to the nets driven by the corresponding logic gates.

Power Mode

A static state of a design in which each power domain operates on a specific nominal condition.

Power Switch Rule

Defines the location and type of power switches to be added and the condition for when to enable the power switch.

State Retention Rule

Defines the registers to be replaced with state retention flip-flops and the conditions for when to save and restore their states.

Special Library Cells for Power Management

Always On Cell

A special cell located in a switched-off domain, and whose power supply is continuous on even when the power supply for the rest of the logic in the power domain is off.

Isolation Cell

Logic used to isolate signals between two power domains where one is switched on and one is switched off. The most common usage of such cell is to isolate signals originating in a power domain that is being switched off, from the power domain that receives these signals and that remains switched on.

Level Shifter Cell

Logic to pass data signals between power domains operating at different voltages.

Power Clamp Cell

A special diode cell to clamp a signal to a particular voltage.

Power Switch Cell

Logic used to connect and disconnect the power supply from the gates in a power domain.

State Retention Cell

Special flop or latch used to retain the state of the cell when its main power supply is shut off.

Command Categories

The following table shows how the CPF commands can be categorized.

Category	**CPF Command**
version command	set_cpf_version
scope commands	set_design
	set_instance
	end_design
general purpose commands	set_array_naming_style
	set_hierarchy_separator
	set_power_unit

	set_register_naming_style
	set_time_unit
design specifications	create_analysis_view
	create_bias_net
	create_global_connection
	create_ground_nets
	create_isolation_rule
	create_level_shifter_rule
	create_mode_transition
	create_nominal_condition
	create_operating_corner
	create_power_domain
	create_power_mode
	create_power_nets
	create_power_switch_rule
	create_state_retention_rule
	define_library_set
	identify_always_on_driver
	identify_power_logic
	set_power_target
	set_switching_activity
	update_isolation_rules
	update_level_shifter_rules
	update_nominal_condition
	update_power_domain
	create_analysis_view
	create_bias_net
	create_global_connection
	create_ground_nets
	create_isolation_rule
	create_level_shifter_rule
	create_mode_transition
	create_nominal_condition
	create_operating_corner
	create_power_domain

Category	CPF Command
design specifications	create_power_mode
	create_power_nets
	create_power_switch_rule
	create_state_retention_rule
	define_library_set
	identify_always_on_driver
	identify_power_logic
	set_power_target
	set_switching_activity
	update_isolation_rules
	update_level_shifter_rules
	update_nominal_condition
	update_power_domain
	update_power_mode
	update_power_switch_rule
	update_state_retention_rules
library-related commands	define_always_on_cell
	define_isolation_cell
	define_level_shifter_cell
	define_open_source_input_pin
	define_power_clamp_cell
	define_power_switch_cell
	define_state_retention_cell

6.2.2 General CPF Commands

create_isolation_rule

create_isolation_rule
-name *string*
-isolation_condition *expression*
{-pins *pin_list* | -from *power_domain_list* | -to *power_domain_list*}...
[-isolation_target {from|to}] [-isolation_output {high|low|hold}]
[-exclude *pin_list*]

Defines a rule for adding isolation cells. This command allows to specify which pins must be isolated.

-isolation_condition *expression*

Specifies the condition when the specified pins should be isolated. The condition can be a function of pins and ports.

-isolation_output {high|low|hold)

Controls the output value at the output of the isolation gates when the isolation condition is true. The output can be high, low, or held to the value it had right before the isolation condition is activated.

Default*:* low

-isolation_target {from|to)

Specifies when this rule applies.

- from indicates that the rule applies when the power domain of the *drivers* of the specified pins is switched off.

- to indicates that the rule applies when the power domain of the *loads* of the specified pins is switched off.

Default: from

-name *string* Specifies the name of the isolation rule.

-pins *pin_list* Specifies a list of pins to be isolated. You can list input pins and output pins of power domains. You can further limit the pins to be isolated using the -from, -to, and -exclude options.

-to *power_domain_list*

Limits the pins to be considered for isolation to input pins in the specified power domains. The power domains must have been previously defined with the create_power_domain command.

create_power_domain

create_power_domain

-name *power_domain*

{ -default [-instances *instance_list*] [-boundary_ports *pin_list*]

| -instances *instance_list* [-boundary_ports *pin_list*]

| -boundary_ports *pin_list* }

[-shutoff_condition *expression*]

[-default_restore_edge *expression*]

[-default_save_edge *expression*]

[-power_up_states {high|low|random}]

Creates a power domain and specifies the instances and boundary ports and pins that belong to this power domain. All top-level boundary ports are considered to belong to the default power domain, unless they have been associated with a specific domain.

Options and Arguments

-boundary_ports *pin_list*

Specifies the list of inputs and outputs that are considered part of this domain.

- For inputs and outputs of the top-level design, specify ports.
- For inputs and outputs of IP instances, specify a list of the instance pins that are part of the domain.

-default Identifies the specified domain as the default power domain. All instances of the design that were *not* associated with a specific power domain belong to the default power domain.

-instances *instance_list*

Specifies the names of all instances that belong to the specified power domain.

-name *power_domain* Specifies the name of a power domain.

-shutoff_condition *expression*

Specifies the condition when a power domain is shut off. The condition is a boolean function of pins and ports. If this option is omitted, the power domain is considered to be always on.

create_power_mode

create_power_mode

-name *string*

-domain_conditions *domain_condition_list*

[-default]

Defines a power mode.

If your design has more than one power domain, you must define at least one power mode. If you define any power mode, you must define one (and only one) power mode as the default mode.

Options and Arguments

-default Labels the specified mode as the default mode. The default mode is the mode that corresponds to the initial state of the design.

-name *string* Specifies the name of the mode.

-domain_conditions *domain_condition_list*

Specifies the nominal condition of each power domain to be considered in the specified power mode. Use the following format to specify a domain condition:

domain_name@nominal_condition_name

create_state_retention_rule

create_state_retention_rule

-name *string*

{ -domain *power_domain* | -instances *instance_list* }

[-restore_edge *expression* [-save_edge *expression*]]

Defines the rule for replacing selected registers or all registers in the specified power domain with state retention registers.

Options and Arguments

-instances *instance_list*

Specifies the instances that you want to replace with a state retention register. Register variable or hiearchical instance in RTL.

-domain *power_domain*

Specifies the name of a power domain containing the target registers to be replaced. All registers in this power domain will be replaced.

-name *string* Specifies the name of the state retention rule.

-restore_edge *expression*

Specifies the condition when the states of the registers need to be restored. The expression can be a function of pins and ports. When the expression changes from false to true, the states are restored.

define_library_set

define_library_set

-name *library_set*

-libraries *library_list*

Creates a library set.

end_design

end_design

Used with a set_design command groups a number of CPF commands that apply to the current design or top design.

set_cpf_version

set_cpf_version [*value*]

Specifies the version of the format. The command returns the new setting or the current setting in case the command was specified without an argument. If specified, this command must be the first CPF command in a CPF file.

set_design

set_design *module* [-ports *port_list*]

Specifies the name of the module to which the power information in the CPF file applies.

set_hierarchy_separator

set_hierarchy_separator [*character*]

Specifies the hierarchy delimiter character used in the CPF file. The command returns the new setting or the current setting in case the command was specified without an argument.

update_isolation_rules

update_isolation_rules -names rule_list

{ -location {from | to}

| -cells cell_list -library_set library_set

| -prefix string

| -combine_level_shifting

| -open_source_pins_only}...

Appends the specified isolation rules with implementation information.

Options and Arguments

-cells cell_list Specifies the names of the library cells that must be used as isolation cells for the selected pins.

-library_set library_set References the library set to be used to search for the specified cells. Specify the library set name. All matching cells will be used. The libraries must have been previously defined in a define_library_set command.

-location {from|to}

Specifies the power domain to which the isolation logic must be added.

- from stores the isolation logic with the instances of the originating power domain
- to stores the isolation logic with the instances of the destination power domain

Default: to

-names *rule_list* Specifies the names of the rules to be updated. The name can contain wildcards. The rule must have been previously defined with the create_isolation_rule command.

update_state_retention_rules

update_state_retention_rules

-names *rule_list*

{-cell_type *string* | -cell *libcell*}

-library_set *library_set*

Appends the specified rules for state retention logic with implementation information.

Options and Arguments

-cell *libcell* Specifies the library cell to be used to map the flops.

-cell_type *string*

Specifies the class of library cells that can be used to map the flops. The specified string must correspond to the value of a power_gating_cell Liberty attribute of a library cell.

-library_set *library_set*

References the library set to be used to search for the specified cell or specified cell type. Specify the library set name.

-names *rule_list* Specifies the names of the rules to be updated. The name can contain wildcards.

Chapter-7

Work done

Work done mainly includes

- Random Verification
- Common Power Format
- Code coverage

7.1 Random Verification

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System On Chip (SOC) designs, verification consumes about 70% of the design effort. The number of verification engineers is usually twice the number of design engineers. When design projects are completed, the code that implements the testbenches makes up to 80% of the total code volume. Generally the design are very bigger and having a lot of functionality .The state space for design is too big for exhaustive coverage. Two approaches are used to cover the state space.

1) Directed Test Bench

- In this approach the known stimulus are given.
- Here we can track down the problems which we have thought of
- When the nature of the design allows easy identification of corner cases, relevant cases.
- When the golden model is easily available is significantly abstract compared to the RTL model

2) Random Approach

- Randomly generate the stimuli and apply to the design
- In each run different stimulus are applied to the design.
- It can find out the problems which might not have been thought
- It may hit corner cases.
- It removes any biases introduced by verification engineer during designing the testbench
- They create more realistic situation

7.1.1 Debugging the testcases

The task of the functional verification is to check that the required functionality is implemented by the RTL. Verification of the design is generally done using the various testcases written in HVL(High level Verification Language) due to various advantages of the HVLs. In the initial phase of the verification a large number of the bugs are found in the RTL as RTL may not be stable. As verification goes further the number of the bugs starts reducing which is desired as the bugs found in the latter stages are very costly.

How ever when any bug is found it is very important to check is that if it is a RTL bug or testcase one. The perception of the specification may differ from designer to verification engineer. This may cause that bug for verification engineer may not be same for designer. So whenever any bug is found it is very necessary to check that if it is due to something done wrong in the testcase.

The steps followed for debugging are listed below.

a) Understand the specification

As mentioned above that understanding of the specification may be different for verification engineer and the designer. So it is very important that first clear the understanding specification by going through the specification doc and try to understand the expected behavior of design. This requires the patient reading of the specification doc and separate out the important points.

b) Study the testcase

Once the specification of design it is next step to understand the testcase. While reading the testcase it is required to understand that which scenario is implemented in it. It is also required to see that all the necessary condition are satisfied by it.

c) Go through the log file

When testcase is run the log file is generated along with the dump. This log file has various details such as which stimulus are given, which phase of the testcase is running at

correspond time, all the info messages printed for debugging purpose, information regarding the coverage and also the various error.

d) Locate the problem

Once the gone through the log file one can list down the errors in the separate file. Then the real task of the debugging starts. One need to locate the error first in the particular section of the testcase. The info messages printed may help in it. Once the erroneous section of the testcase is found then it is required to understand the stimulus given. From this stimulus one should write down expected response of the duv from the specification. Then this response can be compared with the actual one from the dump. From this comparison one can find the mismatch then it is required to debug this mismatch, which leads to the bug. One may need to go up to RTL level so it can be seen that if bug id not from RTL .

e) Make the corrective action

Once the bug find, if the bug is from testcase then proper correction should be made to it. If the bug is from the RTL then it has to be reported to the RTL team and it is fixed.

f) Re-verify testcase

Once the bug is fixed we need to confirm that testcase which failed previous now passes.

7.1.2 Example of debugging Testcase

Design: 4 bit counter

RTL:

```
module counter (C, CLR, Q);
input C, CLR;
output [3:0] Q;
reg [3:0] tmp;
  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
      else
        tmp = tmp + 1'b1;
      end
  assign Q = tmp;
endmodule
```

Testcase:

```
#include <SPSStimulusBase.vrh>
#include <CntTran.vrh>
#include <CntCmd.vrh>
#include "CntIf.vri"
class cnt_sn extends SPSStimulusBase{

local CntTran cnt_tran;
    task new(string name, CntTran cnt_tran_in);
    virtual protected task restart();
    virtual protected task execute();
    reg [3:0] cnt_test;// count generated in the testcase locally to cpmpare it with generated
                       // by the design
    }

task cnt_sn1::new( string      name, CntTran cnt_tran_in)
```

```
{
  super.new(name);
  info(psprintf("Constructing a CNTStim: %s",name));
  this.cnt_tran = cnt_tran_in;
}

task cnt_sn1::restart()
{
        cnt_test=0;
}

task cnt_sn1::execute()
{
@ cnt_mon.clk  cnt_test-cnt_test+1;
 if(cnt_mon.check_cnt=cnt_test)
        psprintf(“Matching the actual count with expected \n”);
else
{
        error(“Mismatch between actual and expected count \n”)
        psprintf(“Acutal=%d \n Expected=%d\n”);
}
```

log file:

```
Error: Mismatch between actual and expected count
Actual=xxxx
Expected=0000 @T=0
Error: Mismatch between actual and expected count
Actual=xxxx
Expected=0001 @T=1
Error: Mismatch between actual and expected count
Actual=xxxx
Expected=0010 @T=2
Error: Mismatch between actual and expected count
Actual=xxxx
```

Expected=0011@T=3

Error: Mismatch between actual and expected count

Actual=xxxx

Expected=0100 @T=3

……..

Here it is very is to debug the error as one can see that expected count increases per clk cycle but actual remains the “xxxx”. As we track the problem to RTL we can see that actual count will not increase till CLR is applied once. As in the testcase side this is not taken care so corrective action is to assert the CLR once in the starting. So testcase after the corrective action is:

Corrected Testcase:

```
#include <SPSStimulusBase.vrh>
#include <CntTran.vrh>
#include <CntCmd.vrh>
#include "CntIf.vri"
class cnt_sn extends SPSStimulusBase{

local CntTran cnt_tran;
   task new(string name, CntTran cnt_tran_in);
   virtual protected task restart();
   virtual protected task execute();
   reg [3:0] cnt_test;// count generated in the testcase locally to cpmpare it with generated
                          // by the design
   }

task cnt_sn1::new( string      name, CntTran cnt_tran_in)
{
  super.new(name);
  info(psprintf("Constructing a CNTStim: %s",name));
  this.cnt_tran = cnt_tran_in;
}
task cnt_sn1::restart()
```

```
{
        cnt_test=0;
        cnt_tran.clr=0;//correction made
}
task cnt_sn1::execute()
{
@ cnt_mon.clk  cnt_test-cnt_test+1;
 if(cnt_mon.check_cnt=cnt_test)
        psprintf("Matching the actual count with expected \n");
else
{
        error("Mismatch between actual and expected count \n")
        psprintf("Acutal=%d \n Expected=%d\n");
}
```

7.2 Common Power Format

As the design elements entered the sub-micron space, the leakage current began to grow to a point where it could no longer be ignored. New methodologies and design techniques were invented to diminish the effects of leakage. Many of these techniques, such as power switch off, can change the functionality of a design if either specified or implemented incorrectly.

Various advanced techniques used for reducing the leakage power require
New techniques such as isolation and state retention used for reducing the leakage power reduction. Now it becomes very necessary to verify that this logic is correctly applied. As mentioned in the earlier chapters it can be verified by two methods :

1) Ad-Hoc method

2) CPF

If the power down domain is small then it is easy to verify using RTL netlist. For this RTL simulators need to mimic the isolation and state retention flops. For simulating srpg the verification engineer must first identify all the srpg registers. A testbench element must be created to store values from the registers when retention sequence occurs. Then when ever power on occurs he need to restore thier values. Also he need to force all the nodes to 'bX during power off by testbench elements. One can see as the design grows this task becomes overwhelming.

Gate level simulation can have isolation cells and srpg flops. So testbench do not need to replicate them. In order to know that isolations and srpg are same as applied in the RTL one need to rerun the full verification suite. Need of gate level simulation means that defects may not be identified until late in the cycle. So errors can not be found out before the tape-out. So it may results in costly mistakes.

To overcome all these problems CPF method is used. When a simulation is run with the CPF file, the simulator takes care of isolating the output of the powered off domains. It will also drive powered down nodes to ‘bX, and is responsible for retaining and restoring the correct register values. Written in Tcl, wild cards enable registers and isolation ports

to be identified easily. This removes the burden of writing cumbersome test bench code that existed with the ad-hoc solution. With the weight lifted, verification engineers can concentrate their efforts on ensuring that the four stages of the low power modes are adequately tested.

The second major benefit of CPF is that it can be read by formal tools. While using CPF in simulation enables one to accomplish the same tasks faster and more efficiently then the ad-hoc solution, using it in a formal environment allows one accomplish things that were impossible in the ad-hoc flow. The logical equivalency between what used for RTL simulation, the netlist and CPF file, and the gate level netlist can be proven. This removes the need to re-run every simulation from the RTL level on the gate level netlist.

In order to get the most from a CPF based verification flow, the same file which is verified by simulation should be used by synthesis. The chance of human error being introduced is decreased by having one central location for the power specification. An error present in the CPF used for synthesis is caught early by the RTL verification. In the ad-hoc solution, the error would not be found until gate simulations had begun. In addition, the CPF can be checked formally against low power rules at on the RTL level, rather than having to wait for a structural netlist.

Advantages of CPF Methods:-

- Eases adoption
    - Language neutral
    - No need to change golden RTL
- Supports IP reuse and portability
    - Eases migration of non power-aware RTL to power-aware
    - Supports multiple instantiation of a module with varying power architecture
- Single view for all power spec
    - Common specification for both design and verification
    - Eliminates duplication of power related information
- Eases architectural exploration & tradeoff

7.2.1 CPF Example

```
set CPF_PATH "/mot/b14177/personal/rc"
source  library_sets_ptvc45.cpf
source  csi_and_regular_srpg_celllist.tcl
source  sc_hd_ctm_lr_alwayson_cells.cpf
source  sc_hd_ctm_lr_isolation_cells.cpf
source  sc_hd_ctm_lr_levelshifter_cells.cpf
source  sc_hd_ctm_lr_srpg_cells.cpf

set_design full_add -ports {iso_en1 switch_en1_b}
set_hierarchy_separator "/"
source /vobs/vb_vault_d3g/d3g_c45/constraints/nominal_condition_ptvc45.cpf
set lp_multiple_power_domain_load_handling 1
set lp_insert_isolation_for_same_domain 1
##############power domains for full adder###############
create_power_domain -name DOMAIN_FULL_ADD_OFF \
-default \
-shutoff_condition !switch_en1_b
create_power_domain -name DOMAIN_SOG \
-boundary_ports *
##############power modes for full adder ################
create_power_mode -name MODE_FULL_ADD_ACTIVE1 -domain_conditions \
{ \
DOMAIN_FULL_ADD_OFF@sog_switchable_normal \
DOMAIN_SOG@sog_always_on_normal \
} -default
create_power_mode -name MODE_FULL_ADD_DISABLE1 -domain_conditions \
{ \
DOMAIN_FULL_ADD_OFF@sog_switchable_normal \
DOMAIN_SOG@sog_always_on_normal \
}
create_power_mode -name MODE_FULL_ADD_STANDBY1 -domain_conditions \
{ \
```

```
DOMAIN_SOG@sog_always_on_normal \
}
############################################################
# Defining Isolation Rules for Full Adder
############################################################
create_isolation_rule -name ISO_DEFAULT_DOMAIN_FULL_ADD_OFF \
-isolation_condition iso_en1 \
-isolation_output low \
-from DOMAIN_FULL_ADD_OFF \
-pins { \
DFT_sdo* \
s \
cout \
}
update_isolation_rules -names ISO_DEFAULT_DOMAIN_FULL_ADD_OFF \
-location from \
-prefix ISO_DEFAULT_DOMAIN_FULL_ADD_OFF

end_design
```

7.3 Code Coveargе

Code Coverage is a technique to measure the completeness of the verification process.Code coverage analysis uses HDL simulation to examine the design and identify design areas that are not being tested, are redundant, or are not being used.

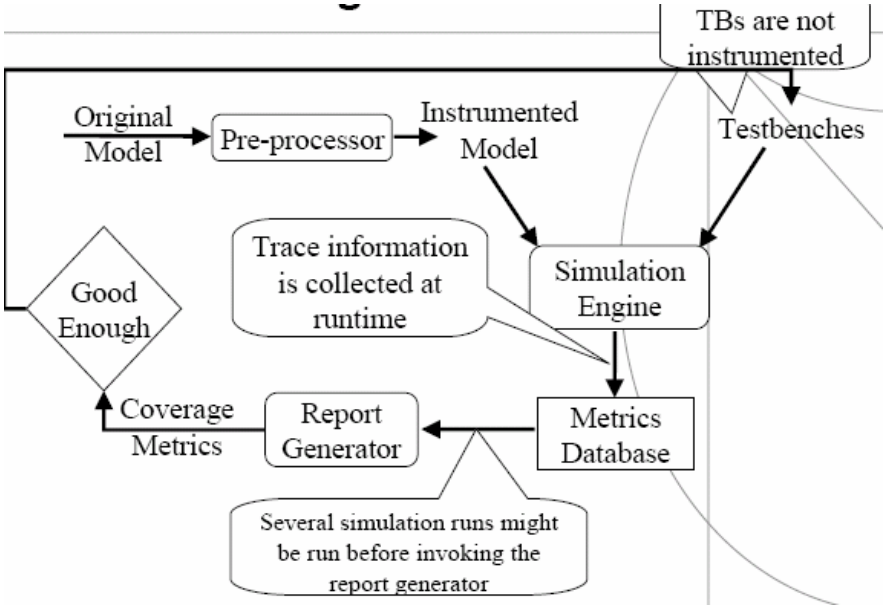

Figure 7.1 Code coverage Process

7.3.1 Code coverage using HDLScore

Coverage analysis at the RTL or behavioral level is analogous to fault coverage at the gate level. Performing coverage analysis prior to synthesis reduces the test verification cycle by moving the process to a higher level of abstraction where the test-bench is more easily understood with regard to the code it is testing.

Without code coverage, the design engineer can only guarantee that the outputs from functional simulation match the expected results for a given set of test vectors; there is no way to ensure the effectiveness of the test vectors. Without knowing how well the test vectors are exercising the design, the designer does not know if more test vectors are needed, where more test vectors are needed, or when to stop simulating.

HDLScore enables the designer to answer all of these questions. Using HDLScore prior to synthesis is the most productive way of using the tool. HDLScore supports coverage

analysis at all levels of design abstraction and for all Verilog/VHDL language constructs. HDLScore enables designers and design verification engineers to generate a quantitative answer to the question: Have we simulated our design enough to commit it to synthesis?

HDLScore works in conjunction with HDL simulation to quantify how well the test vectors exercise the design and to identify parts of a design that require more testing. Coverage analysis reports identify Design areas not completely tested, Redundancy in testing, and Unused portions of the design that can be removed before synthesis. HDLScore s Finite State Machine (FSM) monitoring capability gives you a unique way of quantifying coverage of the control portions of the design. HDLScore s USX technology performs a synthesis interpretation of the control logic and provides analysis using FSM representation. Together with code coverage analysis, HDLScore provides a comprehensive, quantitative measure of the quality of the simulation tests applied to a design.

7.3.2 HDLScore Process

The HDLScore process, in its simplest form, can be thought of as iterating the following steps until you get the coverage you want.

Step 1: Instrumentation compile an HDL design and generate the instru-mented design.

Step 2: Simulation simulate the instrumented design to measure the effective-ness of the test stimulus.

Step 3: Reporting generate reports to analyze the scored data to determine whether you need additional tests.

Step 4: Toggle Reporting (Verilog only) generate reports to analyze toggle scoring using HDLScore s togreport application.

7.3.3 HDLScore Generated Files

The following files are generated during the HDLScore flow. The functions of these files are explained along with the HDLScore process.

HDLScore Design File (.dgn)

Instrumented Verilog Description File (.vin)

Instrumented VHDL Description File(s) (.vhi)

HDLScore Coverage File (.cov)

HDLScore FSM Coverage File (.fsm)

Toggle Master File (.mst)

Toggle Coverage File (.tog) Instrumentation

7.3.3.1 Instrumentation

At this stage, HDLScore generates the following files:

HDLScore Design File (.dgn): The design file contains the design topology necessary for compact storage of simulation coverage data and their subse-quent analysis. HDLScore creates the file once for each compilation using the original source. The contents of this file do not change as the coverage pro-cess proceeds. However, if the original source file changes, the associated HDLScore design file must be re-created.

Instrumented Verilog Description File (.vin): The instrumented Verilog file, which is simulated to obtain the coverage data. This file is functionally equivalent to the original Verilog source files and contains the HDLScore PLI calls.

Instrumented VHDL Description File(s) (.vhi): These are the instrumented VHDL files which are simulated to obtain coverage data. These files are functionally equivalent to the original VHDL source. They contain foreign program-ming language interface calls used in scoring for code coverage and FSM coverage (libfsmi.so or hdlspli.so).

HDLScore Coverage File (.cov): This file contains the coverage data for the selected modules and instances in the design. Hdl generates this file with initialized counts for each coverage item of the design. Coverage counts in the coverage (.cov) file are updated during simulation by HDLScore routines. The coverage file allows incremental coverage runs by design area or coverage type. The user can create separate coverage files for different test-benches. Thus, for a single HDLScore design file, there can be multiple associated coverage files.

7.3.3.2 Simulation

Instrumentation is followed by simulation of the instrumented code. During simulation the HDLScore PLI scores the instrumented design for coverage types selected during

instrumentation. At the end of simulation, the following files are generated with the coverage information.

HDLScore Coverage File (hdls_sim.cov) This file contains the block, path and expression coverage statistics.
HDLScore FSM Coverage File (.fsm) This file contains the FSM coverage statistics.
HDLScore Elaborated Design File (hdls_elab.dgn) (VHDL simulation only).This file contains the design file generated at the end of the Instrumentation phase. It has been elaborated by HDLScore.
Toggle Master and Coverage Files (.mst and .tog) (Verilog simulation only).

7.3.3.3 Reporting

Hdlr reads the HDLScore design file (.dgn), coverage data file(s) (.cov) and/or FSM coverage file(s) (.fsm) generated during simulation and creates customized coverage reports. Hdlr also provides graphical analysis of coverage data with source back annotation in the Source Report Window and Hierarchy Browser Window. Coverage analysis for individual runs (single .cov/.fsm file) or multiple runs (multiple .cov/.fsm files) can be done in hdlr. Hdlr has the unique capability of doing union, difference and intersection of coverage data from different runs. Finite State Machines (FSMs) can be viewed using the State Diagram Viewer within hdlr.

7.4 Types Of Code Coverage

HDLScore provides the following types of coverage monitoring: Block, Path, and Expression Coverage (Code)

Block Coverage: allows monitoring of all exercisable blocks in the Verilog/ VHDL source code and identifies those not exercised during simulation.

Path Coverage: allows monitoring of all logical paths through conditional branches in the Verilog/VHDL source code and identifies logical paths not exercised during simulation.

Expression Coverage: allows monitoring of expressions in continuous assignments and procedural control constructs (if/case conditions).

7.4.1 Expression Coverage

Expression coverage is a debugging tool that factorizes logical expressions and monitors them during simulation. Expression Coverage provides a much finer granularity of coverage metrics than statement, toggle, or branch coverage analysis. For each expression in Verilog or VHDL code, a set of cases is identified, each case specifies one of the total possible combinations of inputs to the expressed logic. Expression coverage then considers whether a simulation run exercises each case of the expression. An expression is fully covered when all of the individual expression coverage cases are exercised. How the expression is exercised may depend on different types of operators. For example, an expression that uses an && operator can have two or more arguments:

```
c = a && b;
f = a && b && c && d && e;
```

Expression Coverage Modes

It allows running evaluation and testing logical expressions for all or individual design blocks by using expression coverage. You can score your entire design or enable scoring for selected blocks in your design (by compiling all or selected source files with the -exc argument). The coverage tool offers two scoring modes to evaluate expression testing. You can score a design by using one selected mode by specifying it on initialization of simulation (-exc control | vector).

The following modes are supported by expression coverage:

Control

This mode controls expressions of single bit signals and checks whether each input of an expression has contributed to an expression result during simulation. Expression coverage checks if and when an expression element solely controls the output of the entire expression. When the control mode is selected, coverage is scored if, and only if, an expression contains a term that controls a result of an expression.

Vector

This mode can be used in case of expressions employing vectors. When you choose this mode, each bit of a multi-bit signal is exercised separately by using the control scoring style.

Expression Coverage Report

Expression coverage factorizes logical expressions and monitors them during simulation. When simulation is finished, statistics are processed and scoring results are saved to a coverage report. Expression coverage data from multiple simulation runs can be merged together in a single report to show expression coverage statistics for an entire suite of simulations.

7.4.2 Block Coverage

The chief advantage of this metric is that it can be applied directly to object code and does not require processing source code. Performance profilers commonly implement this metric. The chief disadvantage of statement coverage is that it is insensitive to some control structures. Without a test case that causes condition to evaluate false, statement coverage rates this code fully covered. In fact, if condition ever evaluates false, this code fails. This is the most serious shortcoming of statement coverage. If-statements are very common. Statement coverage does not report whether loops reach their termination condition - only whether the loop body was executed. With C, C++, and Java, this limitation affects loops that contain break statements. Since do-while loops always execute at least once, statement coverage considers them the same rank as non-branching statements. Statement coverage is completely insensitive to the logical operators (|| and &&). Statement coverage cannot distinguish consecutive switch labels.

Test cases generally correlate more to decisions than to statements. You probably would not have 10 separate test cases for a sequence of 10 non-branching statements; you would have only one test case. For example, consider an if-else statement containing one statement in the then-clause and 99 statements in the else-clause. After exercising one of the two possible paths, statement coverage gives extreme results: either 1% or 99% coverage. Basic block coverage eliminates this problem.

One argument in favor of statement coverage over other metrics is that bugs are evenly distributed through code; therefore the percentage of executable statements covered reflects the percentage of faults discovered. However, one of our fundamental assumptions is that faults are related to control flow, not computations. Additionally, we could reasonably expect that programmers strive for a relatively constant ratio of branches to statements. In summary, this metric is affected more by computational statements than by decisions.

7.4.3 Path Coverage

This metric reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit. Also known as predicate coverage. Predicate coverage views paths as possible combinations of logical conditions . Since loops introduce an unbounded number of paths, this metric considers only a limited number of looping possibilities. A large number of variations of this metric exist to cope with loops. Boundary-interior path testing considers two possibilities for loops: zero repetitions and more than zero repetitions. For do-while loops, the two possibilities are one iteration and more than one iteration.

Path coverage has the advantage of requiring very thorough testing. Path coverage has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 10 if-statements has 1024 paths to test. Adding just one more if-statement doubles the count to 2048. The second disadvantage is that many paths are impossible to exercise due to relationships of data.

Future Scope

- As the future scope one can update the cpf work with new TOOL version and new libraries.

References

1) D3G Blockguide
2) Vera LRM
3) rg_VeraBaseclass document
4) CPF_refrence_Guide