DSP VLSI DESIGNS ARE VERIFIED

` MADDI SOWJANYA, M.Tech Assistant Professor, jillela.sowjanya@gmail.com
[2] BURRI NARENDER REDDY, M.Tech Assistant Professor, narendarburri@gmail.com
[3] P NANDA KUMAR M.Tech Assistant Professor, nandha.iarevlsi@gmail.com
[4] ANAM SRINIVASA REDDY , M.Tech Assistant Professor, anam.srinivas@gmail.com
Department- ECE
Pallavi Engineering College Hyderabad, Telangana 501505.

Abstract:
Foregg, Digital Signal Processing (DSP) is growing with sophisticated capabilities in locally accessible space applications, thanks to the use of Field Programmable Gate Arrays (FPGA) and Specific Integrated Circuits for Application (ASICs). Proof of these perplexing systems is being checked inside tiny timetables and characteristics. It is critical to conduct strict functional monitoring in order to ensure that these systems operate reliably in all conceivable run-time scenarios. Even with the use of cutting-edge Hardware Verification Languages (HVLs) and approaches such as System-Virology (SV) and Universal Verification Methodology (UVM), improving a mechanized self-checking validation state or test seats, including the age of bit-exact genius reference values, is a complex and time-consuming task. This article investigates a utilitarian check method for the DSP-based VLSI setup utilizing SV and Mat lab. The design of the verify situation, method for integrating Mat lab with SV-based validation condition, and age of bit-accurate genius references are continuously examined in detail, in addition to two contextual investigations.

Keywords: DSP, VLSI, UVM, predictor, coverage-powered verification, DPI

I. INTRODUCTION

DigitalVLSI designs are becoming more dynamic in order to meet ever-increasing practical requirements. Design teams are packing more and more logic gates onto a single chip in order to provide the necessary functionality and efficiency within the specified footprint. Practical testing of such systems using a traditional approach using guided test benches does not give sufficient confidence within the time constraints. In terms of enabling limited generation of random stimuli, self-checking and assertion-based verification, as well as defining the useable coverage matrix, test benches built in SV provide advantages. Random testing improves performance over manual testing, reduces the number of test vectors generated, and produces test cases that the verification engineer isn't aware of. Binding assertions to a specification at the simulation level identifies design flaws in real time and significantly reduces debugging times compared to non-assertion-based design. The simulated design's performance is compared to golden reference values generated using HVL and verified automatically during runtime. Functional simulation is considered complete when the goal of 100 percent functional coverage is met. In this SV-based test bench, assertions are used to verify the designs' control signals, and a predictor or checker is used to assess the designs' data processing capabilities. The verification engineer typically hand-codes these checks using a higher level of abstraction. With sophisticated features like DSP, checker development is difficult. With an onboard architecture that includes multiple DSP IP cores with capabilities like sine-cosine lookup table, fixed-to-floating-point translation, FFT, FIR filter, and so on, this becomes much more complex. DSP algorithms are available as standard features in MATLAB. The test bench can be simplified and overall verification efficiency can be significantly improved if these jobs can be utilized as golden reference models/checkers on the test bench. This article investigates the use of SV and the Mat lab pairing method with SV to create a verification environment.

II. Indicator of Authentication Setting

The testing of a VLSI specification is divided into two stages.

Creating Stimuli is the first step.

2. Research on the design's response

In the step of stimulus production, the architecture is set in a certain mode, and stimulus is introduced. In the research portion, the actual verification takes place. Figure 1 depicts a prototype test bench design that performs all of these tasks automatically. Sousing UVM's test bench (verification area) is made up of reusable verification environment components known as verification components. Each subsystem is encapsulated, ready-to-use, and customizable so that it may be used to test any device protocol, sub module configuration, or whole framework. The verification components, in conjunction with the unit under test (DUT), are used to verify the protocol or system model's execution.

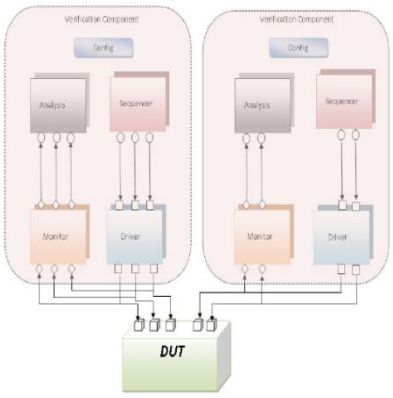

Figure 1: *Verification Environment*

The components that examine DUT activities make up the portion of the research shown in Figure 2. The research section's main components are coverage compilation and scoreboards. The Scoreboard determines whether or not the design is effective. The scoreboard's architecture separates its operations into two categories: forecasting and estimate. A prediction model, often known as a 'Golden Reference Model,' receives the same stimulus source as the DUT and produces established reaction transaction streams. The predictor, written in C, C++, SV, or System-C, applies the DUT functionality at a higher level of abstraction. After the appropriate feature is predicted, the scoreboard will compare the actual outcomes recorded on the DUT with the anticipated results.

I. IN MATLAB PREDICTOR APPLICATION

Mat lab is the industry standard for applying DSP algorithms. The available DSP functions or algorithms created with Mat lab may be used to directly evaluate the output of HDL designs. These DSP functions would be identical to the HDL designs in terms of functionality. Figure 3 shows how the Mat lab DSP function was used within the predictor variable to generate the golden reference values. It greatly simplifies the verification of dynamic architecture. However, since Mat lab DSP functions do not directly support HVL or HDL structures, it is impossible to integrate them directly into the predictor. The following methods are suggested for utilizing Mat lab functionalities in verification settings:

A. MATLAB and Direct Programming Interface (DPI)

Engine:-DPI can be used to connect Mat lab with the SV test bench. The bridge between them is software written in the 'C' language containing Mat lab engine routines.

B. HDL Verifier:-The Mat lab tool box called HDL Verifier offers an EDA connection to help simulators such as Cadence IEV and Questasimsim to bind to

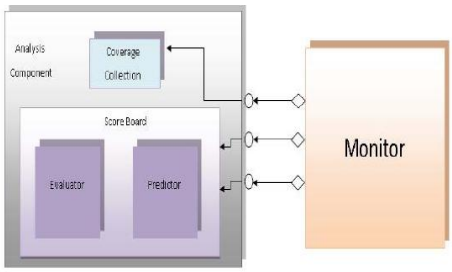

Figure 2: *Analysis Component*

Figure 3: *Mat lab in Predictor*

C. TLM2 transaction communication in MATLAB:-TLM2 Framework C environment transactions are supported by both the UVM library and the Mat lab. This program may be used to create an interface between UVM and System C TLM2 as well as System C TLM2 and Mat lab. Method A is not reliant on an EDA simulator, is customizable, and does not need any additional tools. It was, nevertheless, chosen for presentation.

DPI is an SV and 'C' module that allows users to make direct inter-language feature calls from either side of the interface. Mat lab provides functions for the engine library, including methods for invoking Mat lab from C and FORTRAN programmers. The engine library contains nine methods for controlling the Mat lab computer engine from a 'C' package. Table 1 summarizes these procedures. As shown in Figure 4, these methods may be utilized in the 'C' program to establish a relationship between SV and Mat lab [1].

As illustrated in Figure 5, a prototype C program was developed to link an SV test bench to a Mat lab algorithm.

Table 1: Mat lab engine routines

C Routines	Matlab Functionality
eng{open|close}	Used to Start/Close its engine
engEvalString	Used for execution of its command
eng{get|put}	Used to get/put its variable array value from/to the engine
engOutputBuffer	Used for reading its buffer text content
eng{get|set}Visible	Used for making its engine visibility on /off
engOpenSingleUse	Start for its engine session

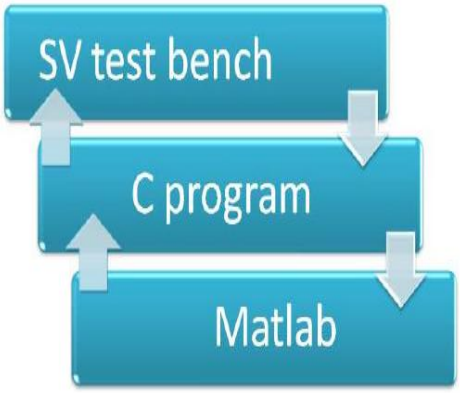

Figure 4: SV- Mat lab Interface

```
#include "engine.h"
#include "matrix.h"
#define  BUFSIZE 80000

Engine *ep;
mxArray *T = NULL, *result = NULL;
char buffer[BUFSIZE+1];
int  buffer_x1 ;
double  result_1[8];
mxArray *result ;

int matlab_engine_start(char  *cmd) //  starting matlab engine
{
        if (!ep && !(ep = engOpen(cmd))) {
                fprintf(stderr, "\nMATLAB engine can not be started \n");
                return 0;
        }
        engSetVisible(ep ,1 );
        engOutputBuffer(ep, buffer, BUFSIZE);
        return 1;
}

int matlab_cmd( const char** cmd)  // execution of matlab command/functions
{     printf("\ncmd is\n %s" , cmd);
        return engEvalString(ep, cmd);
}

void matlab_variable( char** cmd,double z[] ,int k) //reading of matlab buffer
{
double *y;
int i;
mxArray *result = NULL;
```

Figure 5: *Interface C program*

Device Virology and Mat lab both contain a variety of data. The Mat lab's default data type is a double matrix, while the SV test bench transmits and receives data in binary format. As a result, new procedures in the 'C' program were written to conduct the necessary conversion for compatibility with the target environment on the acquired data. Matlab files (.m) and C files are compiled into an ashamed library, which produces a shared library file (.so) and a header file (.h). Mat lab entails interacting with the shared library using the present matrix functions. The gecko compiler and MatlabAPI header data are used to build the 'C' file. Compilation machines are often stored in a central repository.

I. DSP Event Analysis

The practical verification of two onboard IP key designs is known as a case study. These case studies use Matlabalgorithms as a real-time golden guide. For both concepts, a realistic simulation based on coverage was created.

A. Case Study No. 1: Fast Fourier Transform

The first design is an IP core for the dynamic Fast Fourier Transform with an 8192 transition time (FFT). The FFT is calculated using the Cooley-Turkey method on unsealed output. This design was evaluated using the Mat lab FFT method with the same transform size as the gold comparison. Restricted random test vectors produced using SV were implemented on both DUT and Mat lab models. Bit-by-bit DUT outputs were compared to Mat lab performance in real time.

A regular frequency signal with random noise was also generated in Mat lab and applied to both the reference model and the form assessment DUT. The results from each were plotted in real time in Mat lab. The charting of the DUT response over the Matlaboutput, as shown in Figure 6, revealed that the design outputs were not completely balanced and that minor deviations existed owing to DUT's inclusion of superfluous frequency components. Further debugging of the DUT reveals that these differences are created at any time during processing by compression of the stored data owing to limited register duration and verified by the design team.

Case Study 2 of the Sine-Cosine LUT:

The Sine Cosine look up table generator IP heart is the alternative design. The input angle width is 16 bits in this version, providing more than 0.005 resolution (step size) and 10 bits of output width for improved precision. It was verified by utilizing a limited random test variable to run the Mat lab method in a loop with all possible input angle values.

The design's performance (sine/cosine values) was compared to those generated by Mat lab. The angle values (16 bits) generated by the SV test bench are used as sine or cosine LUT addresses in the specification. In order to obtain the angle value from the Matlabalgorithm for the same input angle value, additional procedures for angle conversion activities have to be developed in 'C'. In Mat lab, these values were fixed point translated to make the Mat lab algorithm performance (double - 64 bits long) equivalent to the DUT (10 bits). The final output of DUT was successfully compared and balanced with Mat lab's result (i.e. golden reference values).

II. RESULT

The use of the MATLAB algorithm as a golden guide has helped to reduce the test bench code's production time and complexity. Debugging complex DSP functions has become easier thanks to output comparison on MATLAB plots. The practical verification's overall dependability was also significantly improved.

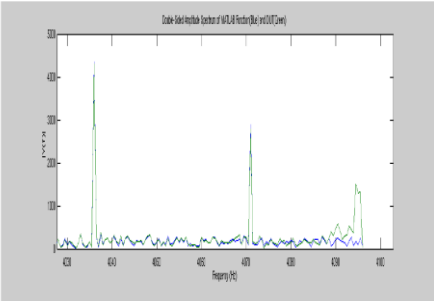

Figure 6*: FFT responses of DUT (green) and Mat lab function (blue)*

CONCLUSION

A new automatic self-checking functional testing method was utilized to guarantee the practical correctness of complicated DSP dependent systems. DSP-based capabilities may be validated with wider coverage in less time using this approach. Mat lab is being utilized to create such unique stimulus feedback signals that are tough to achieve in SV. It is also possible to do a more in-depth examination of the DUT's performance utilizing spectrum analysis and filtering at the RTL stage.

REFERENCES

[1] Havel Modi, Integrating MATLAB with verification HDLs forFunctional Verification of Image and Video Processing ASIC"International Journal of Computer Science & Emerging Technologies"Volume 2, Issue 2, April 2011"

[2] Cookbook–UVM by Verification Academy Mentor graphics.

[3] Writing Testbenches using System Verilog by Janick BergeronSynopsys, Inc.

[4] SystemVerilog for Verification -A Guide to Learning the TestbenchLanguage Features by Chris Spear