# Integrating MATLAB with Verification HDLs for Functional Verification of Image and Video Processing ASIC

Dhaval Modi[1], Harsh Sitapara[2], Rahul Shah[3], Ekata Mehul[4], Pinal Engineer[5]

*Communication System Engineering, L.D. College of Engineering, Ahmedabad[1.]*
*Microprocessor Engineering, M.S. University, Baroda[2]*
*ASIC Division, EINFOCHIPS Pvt. Ltd., Ahmedabad[3]*
*ASIC Division, EINFOCHIPS Pvt. Ltd., Ahmedabad[4]*
*S.V.N.I.T., Surat[5]*
*dhavalmodi045@yahoo.com[1], harsh.sitapara@einfochips.com[2], rahulv.shah@einfochips.com[3] ,*
*ekata.mehul@einfochips.com[4] , pje@eced.svnit.ac.in[5]*

**Abstract -** *The ultimate Aim of ASIC verification is to obtain the highest possible level of confidence in the correctness of a design, attempt to find design errors and show that the design implements the specification. Complexity of ASIC is growing exponentially and the market is pressuring design cycle times to decrease. Traditional methods of verification have proven to be insufficient for Digital Image processing applications. We develop a new verification method based on SystemVerilog verification with MATLAB to accelerate verification. The co-simulation is accomplished using MATLAB and SystemVerilog coupled through the DPI. I will be using the Image Resize design as case study by using co-simulation method between SystemVerilog and MATLAB. Golden reference will be made using MATLAB In-built functions, while rest of the Verification blocks are in SystemVerilog. The goal is to find more bugs from Image resizing Design as compared to traditional method of Verification, reduce time to verify video processing ASIC, reduce debugging time, and reduce coding length.*

**Key words:** *Code base, API, DPI, Design cycle time*

## 1. Introduction

Leading chip development teams report that functional verification has become the biggest bottleneck, consuming approximately 70% of chip development time and efforts. For Image and video processing application, Register transfer level test-benches have become too complex to manage. New method has to be discovered to reduce verification cycle.

Image processing designs easily coded in MATLAB. Therefore, we believe that verification could be significantly improved and accelerated by reusing these golden references models in MATLAB. In this paper we explore combining the power of MATLAB, for Image processing application, signal content generation, spectral analysis, spectrum and waveform display and SystemVerilog, for random stimuli generation. MATLAB and SystemVerilog correlated with each other through the SystemVerilog DPI interface.

### 1.1 Verification Architecture using co-simulation interface

The MATLAB environment is a high-level technical computing language for algorithm development, data visualization, data analysis and numerical computing [1]. MATLAB also included the Simulink graphical environment used for multi-domain simulation and model-based design. Image processing designers take advantage of Simulink as it offers a good platform for preliminary algorithmic exploration and optimization.

First in MATLAB, algorithmic level model is developed. Second step is to start RTL level implementation. To verify this research work, we use SystemVerilog. SystemVerilog has become a concrete RTL level verification language used by many industries. One of the good capabilities of SystemVerilog is to generate random stimuli.

Here golden reference is in MATLAB and design Under Test is in HDL. Scoreboard compares the output of golden reference and DUT. We have to require an efficient transition between algorithmic level and RTL level design. Thus, we need a co simulation between the MATLAB environment and SystemVerilog.

SystemVerilog language doesn't provide any facility to directly call the MATLAB engine. The SystemVerilog Direct Programming Interface (DPI) is basically an interface between SystemVerilog and a foreign programming language, in particular the C language. It allows the designer to easily call C functions from SystemVerilog and SystemVerilog function from c. We make the same C program to call MATLAB Engine library.

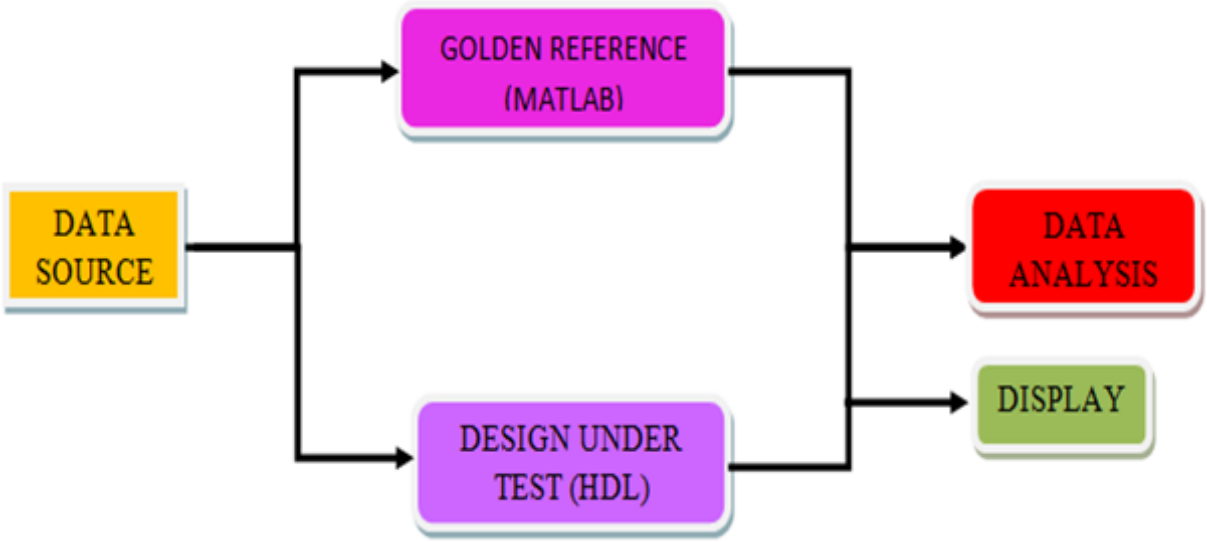


**Figure 1.** Verification Architecture

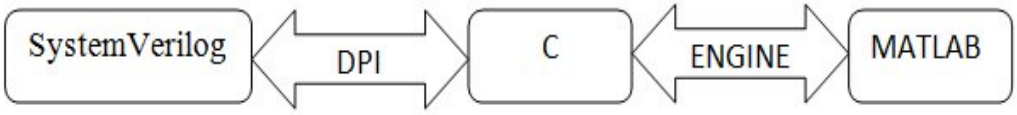


**Figure 2.** Interface between SystemVerilog and MATLAB

Once a link has been established between SystemVerilog and MATLAB, it opens up a wide range of additional capability to SystemVerilog, like stimulus generation and data visualization. The first advantage of our technique is to use the right tool for the right task. Complex stimulus generation and signal processing visualization are carried out with MATLAB while hardware verification is performed with SystemVerilog verification standard. The second advantage is to have a SystemVerilog centric approach allowing greater flexibility and configurability.

## 2. Co-simulation between System Verilog and MATLAB

## 2.1 Simulation between MATLAB and C

### 2.1.1 The MATLAB Engine Library

To enable C to call MATLAB, we use 'engine' library available within MATLAB. The MATLAB engine library contains routines that allow us to call MATLAB software from our own program. Engine programs are standalone C/C++ or FORTRAN programs that communicate with a separate MATLAB process via pipes. MATLAB provides a library of functions that allows us to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB. The MATLAB engine operates by running in the background as a separate process from our own program.

The MATLAB language works with only a single object type: MATLAB array. These arrays are manipulated in C using the 'mx' prefixed application programming interface (API) routines included in the MATLAB engine. This API consists of over 60 routines to create access, manipulate, and destroy mxArrays.

The engine library is part of the MATLAB C/C++ and Fortran API Reference. It contains routines for controlling the computation engine. The function names begin with the three-letter prefix "eng". MATLAB libraries are not thread-safe. If you create multithreaded applications, make sure only one thread accesses the engine application.

### 2.1.2 How to communicate with MATLAB

In this paragraph we show detail about how to write our application with use of MATLAB engine library. Write your application in C/C++ using any of the engine routines to perform computations in MATLAB. Use the mex script to compile and link engine programs. mex has a set of switches you can use to modify the compile and link stages.

**Table 1.** MATLAB Engine routines to communicate with C

| Function | Purpose |
|---|---|
| engOpen | Start up MATLAB engine |
| engClose | Shut down MATLAB engine |
| engGetVariable | Get a MATLAB array from the MATLAB engine |
| engPutVariable | Send a MATLAB array to the MATLAB engine |
| engEvalString | Execute a MATLAB command |
| engOutputBuffer | Create a buffer to store MATLAB text output |
| engOpenSingleUse | Start a MATLAB engine session for single, nonshared use |
| engGetVisible | Determine visibility of MATLAB engine session |
| engSetVisible | Show or hide MATLAB engine session |

MATLAB supplies a mex options file to facilitate building MEX applications. This file contains compiler-specific flags that correspond to the general compile, prelink, and link steps required on your system. If you want to customize the build process, you can modify this file. The MATLAB Engine Library is an external library; several steps have to be taken in order to utilize its capability within SystemVerilog. The Linker path has to be modified. Use of the MATLAB Engine Library also requires modifications to the C compile command line.

### 2.1.3 Compiling and Linking MATLAB Engine Programs

**Step: 1** Write your application in C/C++ or FORTRAN using any of the engine routines to perform computations in MATLAB.

**Step: 2** Build the Application. Use the mex script to compile and link engine programs.

**Step: 3** Use of MEX Options File:- MATLAB supplies an options file to facilitate building MEX applications. This file contains compiler-specific flags that correspond to the general compile, prelink, and link steps required on your system. If you want to customize the build process, you can modify this file.

**Step: 4** Building an Engine Application on LINUX Systems.

Build the executable file using the ANSI compiler for engine stand alone programs and the options file engopts.sh:

**optsfile = [matlabroot '/bin/engopts.sh'];**

**mex('-f', optsfile, 'engdemo.c');**

Verify that the build worked by looking in your current working folder for the file engdemo:

**dir engdemo**

To run the demo in MATLAB, make sure your current working folder is set to the one in which you built the executable file, and then type:

**!engdemo**

We can change compiler using **mex -setup**. We can choose GCC or LCC compiler for our application. MATLAB provides inbuilt compiler LCC for C/C++ programs.

## 2.2 Simulation between SystemVerilog and C

### 2.2.1 Introduction about Direct Programming Interface

Direct Programming Interface (DPI) is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components can be written in a language (or more languages) other than SystemVerilog, hereinafter called the foreign language. On the other hand, there is also a practical need for an easy and

efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI.

### 2.2.2 Import Method

Methods implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog, such methods are referred to as imported methods. Imported tasks or functions are similar to SystemVerilog tasks or functions. Imported tasks or functions can have zero or more formal input, output, and inout arguments. Imported tasks always return an int result as part of the DPI-C disable protocol and, thus, are declared in foreign code as int functions.

The syntax import method:

**import {"DPI-C"}[context|pure][c_identifier = ] [function task] function_ identifier|task _identifier] ([port_list]);**

#### 2.2.3 Export Method

Methods implemented in SystemVerilog and specified in export declarations can be called from C, such methods are referred to as exported methods. Syntax of export method is same as import method.

The syntax import method:

**export {"DPI-C"}[context|pure][c_identifier = ] [function task][ function_ identifier|task _identifier] ([port_list]);**

## 2.3 Co-simulation between SystemVerilog and MATLAB

## 2.3.1 Combining power of SystemVerilog and MATLAB using DPI

Here I combine the SystemVerilog DPI and MATLAB application programming interface. I use wrapper of C around MATLAB Engine and use of DPI to communicate with SystemVerilog as shown in the figure 2.3.

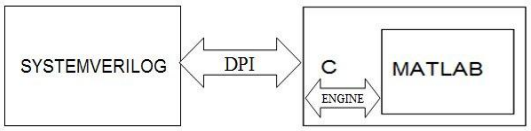


**Figure 3.** Co-Simulation between SystemVerilog and MATLAB

I make code in which I call "engdemo.c" from SystemVerilog using import DPI method. Here first SV code is executing and with import DPI engdemo.c is executing. Output is combination of both MATLAB and SystemVerilog.

To compile above program I use following command:

**Irun dpic.sv try1.c -I/opt/matlab2008/extern/include -L/opt/matlab2008/bin/glnx86 -leng**

When the SystemVerilog compiler while encountering the C code, It calls GCC compiler to compile the C code in background. The final control however remains within the SV compiler.

## 2.3.3 Flow Chart of co-simulation

A block diagram of the flow used for the Object tracking project is shown in the following diagram. The items existing in the SystemVerilog environment are in the left column. The middle two columns show tasks existing in the two interface C-layers. The far right column shows tasks existing in the MATLAB workspace.

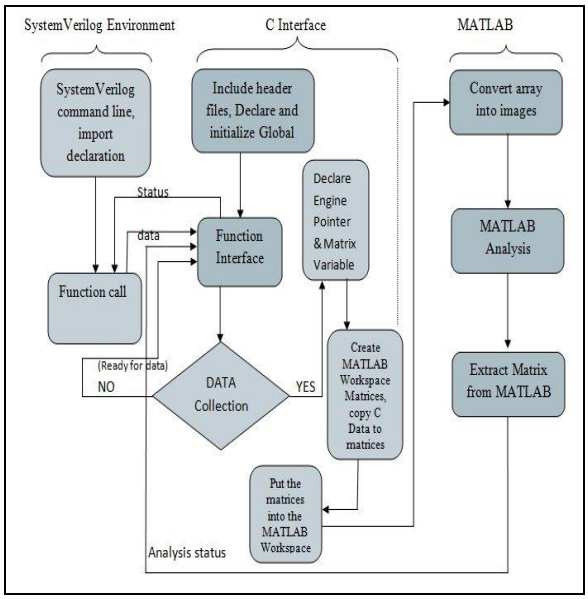


**Figure 4.** Flow chart of Co-simulation

## 3 Image Resizing as a case study

### 3.1 Overview and Objective

Image resizing includes image enlargement and image shrinking. The resizing of image is required in different applications. Now a days, in every television sets Picture-in-Picture (PIP) facility is available. In that a small image of one channel is shown in the main image of current channel. This facility uses the fundamentals of image resizing. The image of other channel is shrunk in smaller size and shown in current channel screen as a small window. The image resizing is also used in all media players available in PC now days.

### 3.2 Image Resizing Interface

The top level Input and Output pin diagram for IMR DUT device is shown below. It shows both host interface side and memory interface side input/output signals.

### Input Interface:-

Imr_host_clk_i

Imr_host_reset_ni

Imr_host_addr_i[15:0]

Imr_host_wr_data_i

Imr_host_wr_en_i

Imr_host_rd_en_i

### Output Interface:-

Imr_clk_i

Imr_reset_ni

Imr_wr_hold_ni

Imr_wr_grant_i

Imr_rd_hold_ni

Imr_rd_grant_i

Imr_wr_brust_o

Imr_rd_brust_o

imr_wr_ald_o[31:0]

imr_rd_ald_o[31:0]

### 3.3 Verification Architecture for Image Processing Application

As we know, Image processing Application is easily made in MATLAB. Due to visualization capability of MATLAB, it is very easily checked by human beings. So I can make Scoreboard is in MATLAB.

The output of Scoreboard and DUT in checker is compared & with the help of co-simulation, the output of MATLAB is transferred in SystemVerilog. So checker is also in SystemVerilog. Rest of the blocks is in SystemVerilog.

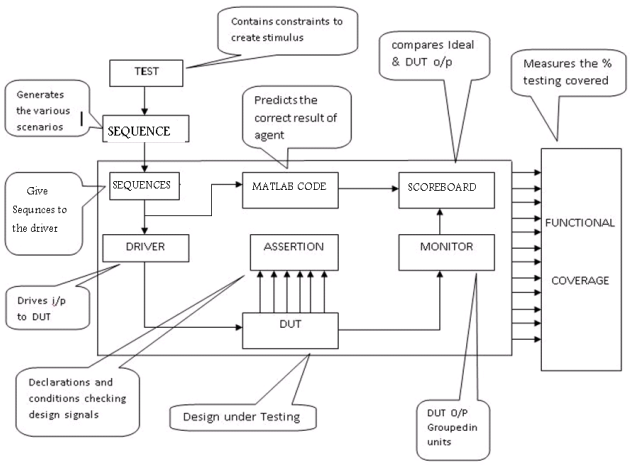


**Figure 5.** Image Resizing Verification Environment

**Table 2.** Verification Architecture using Co-simulation

| Sr No. | OVM Component | Coding Languages |
|---|---|---|
| 1 | Packet | SystemVerilog |
| 2 | Sequence | SystemVerilog |
| 3 | Sequences | SystemVerilog |
| 4 | Driver | SystemVerilog |
| 5 | Monitor | SystemVerilog |
| 6 | Golden reference | MATLAB |
| 7 | Scoreboard | SystemVerilog |
| 8 | Coverage | SystemVerilog |

## 3.4 Image Resizing OVM Verification Components:-

OVM is a complete verification methodology that codifies the best practices for development of verification environments. OVM supports the transaction level modeling with built in class which reuse easily by extending it.

OVM uses a SystemVerilog implementation of standard TLM interfaces for modular communication between components. The architecture of OVM is same as shown in SystemVerilog for my case study of Image Resizing. But communication is simple and easy.

#### 3.4.1.1 Top level:-

**Blocks:-**

- Interface

- Design instance

- Clock Declaration and Generation

**Description:-** It has Description of Interface and Top module and mapping the interface with DUT and with testbench environment block.

#### 3.4.2 Class Environment extends ovm_env:-

**Methods:-**

Build()

Connect()

**Description:-** Environment class is used to implement verification environments in OVM. It is extension on ovm_env class. The testbench simulation needs some systematic flow like building the components, connection the components, starting the components etc. ovm_env base class has methods formalize the simulation steps.

#### 3.4.3 Class Packet extends ovm_sequence_item:-

**Members: -** rand bit [7:0] sa;

rand bit [7:0] da;

rand bit [31:0] rowpixel;

rand bit [31:0] colpixel;

rand bit [3:0] hodis;

rand bit [3:0] verdis;

**Description: -** One way to model Packet is by extending ovm_sequence _item. ovm_sequence_item provides basic functionality for sequence items and sequences to operate in a sequence mechanism. Packet class should be able to generate all possible packet types randomly. To define copy, compare, record, print and sprint methods, we will use OVM field macros.

#### 3.4.4 Class Sequencer extends ovm_sequencer #(Packet):-

**Methods: -** end_of_elaboration();

**OVM macro:-**
`ovm_sequencer_utils(Sequencer)

**Description:-** A Sequencer is defined by extending ovm_sequencer. ovm_sequencer has a port seq_item_export which is used to connect to ovm_driver for transaction transfer.

#### 3.4.5 Class sequence extends ovm_sequence #(Packet):-

**OVM macros: -**
`ovm_sequence_utils(sequence , sequencer)

**Description:-** A sequence is defined by extending ovm_sequence class. This sequence of transactions should be defined in budy() method of ovm_sequence class. OVM has macros and methods to define the transaction types.

#### 3.4.6 Class Driver extends ovm_driver #(Packet):-

**Methods: -** build();

end_of_elaboration()

reset_dut()

cfg_dut();

drive(Packet pkt)

run()

**OVM macros: -** ovm_analysis_port #(Packet) Drvr2Sb_port

`ovm_component_utils(Driver)

**Description:-** Driver is defined by extending ovm_driver. Driver takes the transaction from the sequencer using seq_item_port. This transaction will be driven to DUT as per the interface specification. After driving the transaction to DUT, it sends the transaction to scoreboard using ovm_analysis_port.

#### 3.4.7 Class Receiver extends ovm_component:-

**Methods: -** build()

end_of_elaboration()

run()

**OVM macros: -** `ovm_component_utils(Receiver)

**Description: -** Receiver collects the data bytes from the interface signal. Receiver class is defined by extending ovm_component class. It will drive the received transaction to scoreboard using ovm_analysis_port.

#### 3.4.8 Class Scoreboard extends ovm_scoreboard:-

**OVM macros: -** `ovm_component_utils(Scoreboard)

**Description:-** Scoreboard compare the output of golden reference and DUT.

### 3.5 Advantages compare to traditional method of Verification

Here Golden reference is in MATLAB for Image Resizing ASIC. So we can reduce code length for the same Image Resizing logic if we can code in verification HDLs. Another advantage is reduction in debugging time. We can use the MATLAB inbuilt function In our Verification architecture. So we have advantages of both SystemVerilog and MATLAB. This way we can reduce ASIC design Cycle for Image processing ASIC.

## 4 Conclusions

In this project a verification environment based on co-simulation interface between SystemVerilog and the MATLAB environment has been presented. The DPI C-layer can be used to interface to a wide variety of C base libraries and also the MATLAB Engine Library. The simulation stimulus could be generated from SystemVerilog; this would allow more robust image. Use of the MATLAB graphics capabilities could be more fully utilized. A more complete testbench can be build up in a shorter period of time than with traditional methods. Use of the SystemVerilog and MATLAB could be extended in a variety of directions for various applications.

## 5 Future Works

The co-simulation between SystemVerilog and MATLAB has been used in the digital image processing application project. Instead of creating time consuming stimuli in SystemVerilog, data generated from MATLAB environment is used to drive the testbench. Further using the MATLAB, a golden reference model is created. This Golden reference model is used in SystemVerilog environment to compare behavior of the Design under verification.

## 6 References

1. Compiling and Linking MATLAB Engine Programs, which is available online at http://www.mathworks.com/help/techdoc/matlab_external/f39903.html
2. MATLAB Application Program Interface Guide (December 1996).
3. Calling existing C code from MATLAB which is available online at http://www.mathworks.com/support/compilers/interface_r13.html#Call_MATLAB_from_C
4. SystemVerilog Language Reference Manual by Accellera's Extension to Verilog, 2002, 2003.
5. Irun-user guide from cadence, product version 9.2, july 2010.
6. Brian Bailey, "CoVerification: From Tool to Methodology," white paper, www.mentor.com, June 2002.
7. Jean-François Boland "USING MATLAB AND SIMULINK IN A SYSTEMC VERIFICATION ENVIRONMENT" McGill University, QC, Canada
8. John Stickley, & Wade Stone, "Accelerated Verification of a MATLAB-Driven Digital FIR Filter RTL Design Using Veloce and TBX" Mentor Graphics Corporation
9. A Zuloaga, J. L. Martín, U. Bidarte, J. A. Ezquerra "VHDL test bench for digital image processing systems using a new image format" Department of Electronics

and Telecommunications, University of the Basque Country

10. Janick Bergeron. Writing Test benches using SystemVerilog. Springer, 2006.
11. William K. Lam. Hardware Design Verification. Pearson Education, Inc., 2005.
12. Jean Francois Boland, Cosimulation of Matlab with system C, Mcgill University, Canada, 2004.
13. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, 2006 by Chris Spear. ISBN:0387270361, Publisher:Springer
14. Hardware Verification with System Verilog, 2007 by Mintz, Mike, Ekendahl, Robert ISBN: 978-0-387-71738-8, Publisher:Springer
15. Writing Testbenches using SystemVerilog, 2006 by Bergeron, Janick ISBN: 978-0-387-29221-2.
16. The Art of Verification with SystemVerilog Assertions, 2006 ISBN-13: 978-0-9711994-1-5