2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Accelerate Verification of Complex Hardware Algorithms using MATLAB based SystemVerilog DPIs

Samuele Candido, Infineon Technologies, Dresden, Germany (*samuele.candido@infineon.com*)

***Abstract*— This paper shows how MATLAB can support the verification of complex mathematical algorithms in a SystemVerilog-UVM environment. This case study is based on a real project of a RADAR SoC (System On Chip) with a complex DSP (Digital Signal Processor) being embedded in the device. As part of the verification process, a model to generate the expected results for each of the DSP steps had to be implemented. In the first phase of the project, floating-point models to predict the results have been implemented in SystemVerilog. However, the implementation and the debugging of the models required a significant effort. Moreover, the incrementing complexity and the addition of new DSP features made the SystemVerilog models hard to maintain. In the second phase of the project, the algorithm team provided fixed-point MATLAB models matching the RTL implementation of each algorithm. The MATLAB models were reused by the verification team to replace the SystemVerilog floating-point models, allowing the team to focus on the implementation of testcases, checks, and coverage, rather than the implementation of the model. The transition from a floating-point model to a fixed-point model is also particularly important for this kind of applications, since errors of one LSB could compromise the correctness of the results. The integration of the MATLAB functions was done by using SystemVerilog DPI (Direct Programming Interface), which allowed to easily reuse the MATLAB code from the testbench. This not only shortened the verification time and effort, but it also allowed to reduce to the minimum the error tolerance used in the testbench checks.**

Keywords—verification; MATLAB; hardware algorithms; SystemVerilog; DPI

I. INTRODUCTION

The paper explains the advantages of using MATLAB based SystemVerilog DPIs for the verification of complex mathematical algorithms. An approach to convert the MATLAB code into a DPI component, which consists of C files and a SystemVerilog package, is also presented. The case study is a RADAR SoC with embedded digital signal processing (DSP), which is capable of detecting moving targets and providing information such as position and velocity. For this application, the usage of a bit accurate reference model is fundamental, since small discrepancies can lead to wrong detection and results. Moreover, the availability of the MATLAB reference model, which can be reused by the verification team, can drastically reduce the verification cycles, since the team does not need to invest effort in implementing and debugging the model.

Section II provides a short overview of the algorithms involved in the project, for which the MATLAB modeling is particularly suitable. Section III introduces the SystemVerilog DPI concept and the DPI component definition. Section IV explains the main reasons that lead to the replacement of the SystemVerilog model through DPI components based on MATLAB code. Section V presents the approach. Given a MATLAB function, the steps needed to generate the DPI component, so that the function can be called from the SystemVerilog-UVM testbench, are shown. The few challenges encountered during this process are explained in Section VI.

II. HARDWARE ALGORITHMS

The details of each algorithm involved in the RADAR SoC are not relevant for the scope of this paper. However, a short description of these algorithms can help the reader to understand why MATLAB and SystemVerilog DPIs could increase the verification efficiency.

The RADAR SoC includes an embedded digital signal processing (DSP) unit, allowing motion and target detection. The processing starts when data acquired by the receiver channels are converted to bursts of samples in fixed-point format by analogue to digital converters (ADCs). The consecutive bursts are then compared to observe any change in the time domain. Note that in this step different options can be configured (for example, a simple

arithmetic difference or an infinite impulse response (IIR) filter). If a change is detected, and therefore a potential target, successive processing steps are applied. Real Fast Fourier Transform (RFFT) is used to identify the range of the potential target. Before the RFFT step, a window function can be applied to reduce the effect of spectral leakage of the RFFT. To identify the velocity of the potential target, a Complex Fast Fourier Transform (CFFT) is applied. In order to determine whether the detected motion is a real target or background noise and interference, the Constant False Alarm Rate (CFAR) algorithm is performed. Magnitude and position of the target are also provided in this step. The process continues with the calculation of the azimuth and elevation angles of the target. This is done combining the results from the different receiver channels and by applying CORDIC (Coordinate Rotation Digital Computer) operations.

All operations are performed in fixed-point format. More precisely, multiple Q formats are used, which specify the number of bits allocated for the integer part, the number of bits for the fractional part, and also the sign. The format changes between the different computation steps and the alignment between the formats can be configured (for example sign extension, LSB extension, 0 extension, etc.). It is then clear that a reference model shall produce results considering the mentioned fixed-point formats and configuration options. An inaccuracy within any of these steps could lead to a wrong expected result.

To generate the expected results for the described operations, and some other not mentioned steps, 15 DPI components have been generated from MATLAB code and imported into the SystemVerilog-UVM testbench environment. This modular approach allowed to verify the entire processing chain by combining the DPIs, but it also allowed to verify the single algorithms in isolation.

III. SystemVerilog DPI

The SystemVerilog Direct Programming Interface (DPI) allows to interface SystemVerilog with code written in foreign languages. With a DPI, functions in a foreign language can be imported and called from SystemVerilog. DPI also allows to export a SystemVerilog function that can be called from a foreign language. The main advantage of the SystemVerilog DPI is that already existing code can be easily reused [1].

In this work, the term DPI component is often used. A DPI component consists of a set of C files and headers, and a SystemVerilog file. The C files are the implementation in C language of the function originally developed in MATLAB. The SystemVerilog file contains a package with the “import DPI” declarations. An example of the SystemVerilog package and how the imported functions can be called from the SystemVerilog testbench is shown in Section V.

IV. Transitioning from SystemVerilog to MATLAB modeling

In the early phase of the project, the algorithm team used the available floating-point MATLAB models to generate the expected results for the processing steps described in Section II. The results were provided to the verification team in a text format. This allowed to create direct testcases and quickly identify first bugs [2]. However, the expected results were provided for a limited set of defined configurations. Due to the wide configuration space, the previous approach was not enough to verify the RADAR operation in depth. Therefore, to allow the usage of constrained-random tests, the verification team implemented floating-point models in SystemVerilog based on the specification. Although this approach was sufficient for the first phase of the project, it had following drawbacks:

- A significant amount of time and effort had to be invested in the implementation and debugging of the SystemVerilog floating-point models.
- The SystemVerilog models, as well as the expected data from the algorithm team, provided results generated using floating-point arithmetic. The hardware performs all calculations in fixed-point format. Therefore, a tolerance had to be introduced in the testbench checks when comparing the hardware with the reference results. Please note that an error of 1 LSB in one of the processing steps could lead to a different final result. Therefore, the tolerance in the testbench checks shall ideally be 0.

- Later in the project, not only the number of algorithms incremented, but also the complexity of the existing ones increased, making the SystemVerilog implementation more challenging to maintain, time consuming, and error prone.

In the second phase of the project, fixed-point models implemented with MATLAB were provided by the algorithm team. The MATLAB code, reflecting the format used by the hardware, was provided for each algorithm involved in the DSP. At this point, the task of the verification team was to create SystemVerilog DPIs based on the MATLAB code, and to integrate the function calls in the testbench. This allowed to overcome the previous challenges, and it enabled the possibility to verify the bit accuracy of the hardware results.

V. FROM MATLAB TO SYSTEMVERILOG DPI: THE APPROACH

Figure 1 shows an overview of the steps followed to generate a SystemVerilog DPI component (see Section III for more information) from the MATLAB code for the given algorithm.

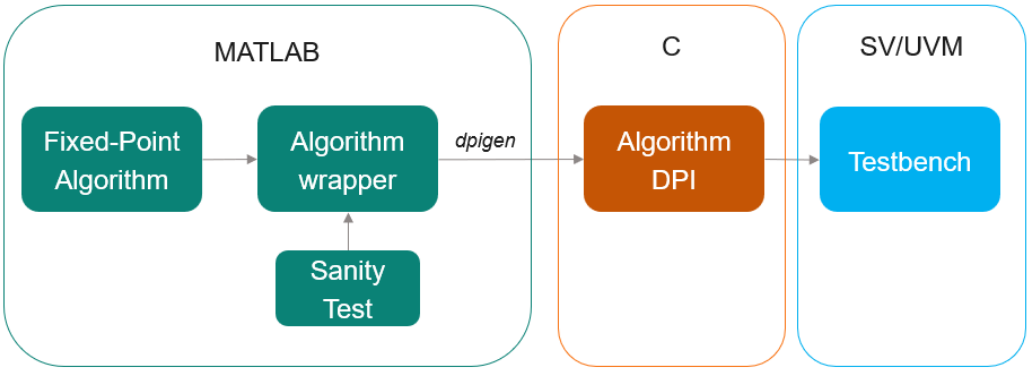

Figure 1. Conversion of MATLAB code into a SystemVerilog DPI

1. First of all, the MATLAB code, in this project the fixed-point MATLAB implementation of a given algorithm, is embedded in a wrapper function. This step is mainly needed to align the format of the arguments expected by the testbench with the arguments provided by the MATLAB function. For example, real and imaginary part of complex numbers are handled in SystemVerilog as different arguments, while MATLAB is expecting one single complex argument. Moreover, SystemVerilog logic data types of a generic number of bits, had to be adapted to the data types available in MATLAB. A very simple example for an algorithm and a wrapper function is shown in (1) and (2).

```
% Implementation of an algorithm in MATLAB
function theta = alg_matlab(phase_difference)
    % body of the function
    …
end
```

(1)

```
% Wrapper for the MATLAB function
function theta = alg_matlab_wrapper(phase_diff)
    % reinterpret input (from int32 to Q format)
    phase_diff_q_format = reinterpretcast(phase_diff,numerictype(1,32,23));
    % call MATLAB function
    theta = alg_matlab(phase_diff_q_format);
end
```

(2)

2. A simple sanity test is executed to verify that no error has been introduced in the wrapper code. The test generates results with the original MATLAB function based on random input data, and it compares them with the results provided by calling the wrapper function.
3. The wrapper function is then converted into the DPI component, generated in C code, using the *dpigen* function provided by MATLAB. Besides the basic arguments related to input and output files location,

the *dpigen* function needs a configuration object (*EmbeddedCodeConfig*) as argument, as well as the data type of the input arguments (the data type of the output is normally automatically derived). The configuration objects allow to customize many aspects of the generated DPI component, such as target language (C or C++), speed and memory characteristics, code appearance, debugging options, replacements with custom code, etc. The *dpigen* command not only generates the C code reflecting the functionality implemented in the original MATLAB function, it also provides a SystemVerilog package with the import declarations. Please note that *dpigen* requires the MATLAB add-on "*ASIC Testbench for HDL Verifier*". An example of the usage of the *dpigen* command is the following (3):

```
dpigen -args {int32(0)} -c alg_matlab_wrapper.m -d alg_dpi -config cfg_dpi
```
(3)

4. Now the DPI component is ready to be used within the testbench environment, and the C functions can be called from the SystemVerilog code. Note that the generated C files and the SystemVerilog package need to be compiled by the simulator tool, which is creating and linking the object libraries. With this approach, no interaction with MATLAB during simulation is needed. An example of the SystemVerilog package with the "import DPI" declarations is shown in (4), while (5) shows how the imported C functions can be used within the SystemVerilog-UVM testbench environment.

```
package alg_dpi_pkg;

// Declare imported C functions
import "DPI-C" function chandle DPI_alg_initialize(input chandle existhandle);
import "DPI-C" function chandle DPI_alg_reset(input chandle objhandle,input int
phase_diff, output int theta);
import "DPI-C" function void DPI_alg_output(input chandle objhandle,input int
phase_diff, output int theta);
import "DPI-C" function void DPI_alg_terminate(input chandle existhandle);

endpackage : alg_dpi_pkg
```
(4)

```
chandle objhandle = null;

objhandle = DPI_alg_initialize(objhandle);
DPI_alg_output(objhandle,phase_diff_tb,theta_tb);
DPI_alg_terminate(objhandle);
```
(5)

The previous steps have been applied for each algorithm. As a result, the generated DPI components could be combined together, and enabled and disabled based on the configuration, to generated the expected results. Moreover, they could also be singularly used, for example to generate data in testcases focusing more on one of the specific algorithms.

VI. CHALLENGES

Although the steps shown in Section V seem straightforward, the verification team had some minor challenges during the development phase:

- Not all MATLAB code can be translated into a DPI component. As a result, in such cases, multiple iterations with the algorithm team were needed to update the code and make it suitable for the translation.
- Matrices are not supported as a DPI argument. In the present case study, matrix arithmetic is heavily used and many conversions from matrix to vector (and vice versa) were needed. However, these conversions can be easily implemented in the MATLAB wrapper function. The wrapper function accepts vectors as arguments (and the information regarding the number of rows and columns), and it creates the corresponding matrix. The matrix is then passed to the MATLAB function.

- Differences in the data types between MATLAB and SystemVerilog required some attention, adaptions and reinterpretation of arguments, to align between the SystemVerilog data types and the integer MATLAB data types.
- The need of a wrapper function can be a source of issues, since errors can be introduced (due to the input/output and vector/matrices manipulation for example).
- The successful and efficient usage of the DPI components is strongly dependent on the maturity of the MATLAB model.
- The debugging time is affected by the limited visibility into the DPI component. While it is straightforward to add debug information within SystemVerilog-UVM models, for example in form of macros such as `uvm_info, it is not possible to directly access variables and temporary results in the DPI component, without having to modify the MATLAB code and regenerate the DPI component. In this project, the debugging of particular cases was done by exporting the input data from the SystemVerilog-UVM testbench into a MATLAB test. This process was time consuming and error prone. Another considered option was to add the interesting internal variables to the function outputs, so that they could be easily accessed from the SystemVerilog-UVM testbench. However, this approach does not scale, particularly with complex algorithms including many calls of MATLAB functions.
- One last aspect to consider is that a small amount of glue code on SystemVerilog side had to be implemented. The glue code allowed to connect all the single DPIs and to enable and disable them based on the configuration. The operations performed in the glue code shall be reduced to the minimum.

VII. CONCLUSIONS

Although the usage of the fixed-point MATLAB models created a dependency between verification and algorithm team, this strategy provided a number of advantages. The verification team was provided with a golden reference from the algorithm team, avoiding the effort for the implementation, debugging and maintenance of the model in SystemVerilog. Due to the heavy usage of matrix arithmetic with complex numbers and trigonometry operations, MATLAB was a good candidate for the implementation of the mathematical models. Following the approach presented in Section V, each MATLAB function could be transformed in a DPI component with a limited amount of effort. Each function could then be easily called from the SystemVerilog-UVM testbench. All DPI components could be combined to form the processing chain, they could be enabled or disabled based on the configuration, and they could also be used singularly to generate the expected results for only one of the algorithms. Moreover, with the fixed-point MATLAB models, the verification team reduced the tolerance of the checks to zero, which was needed when using the SystemVerilog floating-point model. From an application point of view, this allowed to verify the bit accuracy of the results provided by the hardware, which is fundamental for RADAR motion and target detection.

In order to facilitate the integration of MATLAB based SystemVerilog DPI components, some aspects shall be improved. One of these is the reporting and the visibility from within the DPI component. Debugging any issue with the root in the MATLAB code is challenging. Besides the inputs and the outputs of the DPI components, the reporting of intermediate values is not possible without additional effort. Another aspect to consider is that the developer of the MATLAB function should be aware of the limitations and challenges mentioned in VI. With best practices and guidelines, the number of iterations between developer and verification engineer can be reduced. This would also make the MATLAB wrapper function unnecessary.

REFERENCES

[1] "SystemVerilog DPI Tutorial," *Doulos.com*, 2024. https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-dpi-tutorial/ (accessed Jun. 22, 2025).

[2] M. Litterick, A. Ivankovic, B. Arsov, and A. Kumar, "Hard Math -- Easy UVM: Pragmatic solutions for verifying hardware algorithms using UVM," DVcon Europe 2024.