Accelerate Verification of Complex Hardware Algorithms using MATLAB based SystemVerilog DPIs

Samuele Candido, Infineon Technologies Dresden AG & Co. KG

15.10.2025

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

Infineon

Agenda

1. Introduction
2. SystemVerilog DPI
3. RADAR SoC example
4. Translate MATLAB code into DPI components
5. Challenges
6. Next steps and lessons learned

Introduction

- In July 2024, I was assigned the task of integrating MATLAB functions into a SV/UVM testbench
- This Engineering Paper is going through all steps needed to create and use SystemVerilog DPIs generated from MATLAB code
- Case study: RADAR SoC with embedded DSP

SystemVerilog DPI

- Direct Programming Interface (DPI)
  - Interface SystemVerilog with foreign language code
    - Foreign language functions can be imported and called from SystemVerilog
    - SystemVerilog functions can be exported and called from a foreign language
  - Reuse already existing code

- DPI component
  - C files and headers generated from MATLAB code
  - SystemVerilog package with “import DPI” declarations
  - In this context: these files are generated by the MATLAB function *dpigen*

RADAR SoC example – quick overview (1)

- Radar for consumer electronics with embedded DSP
  - Motion, gesture, and target detection
  - Target position, magnitude and velocity

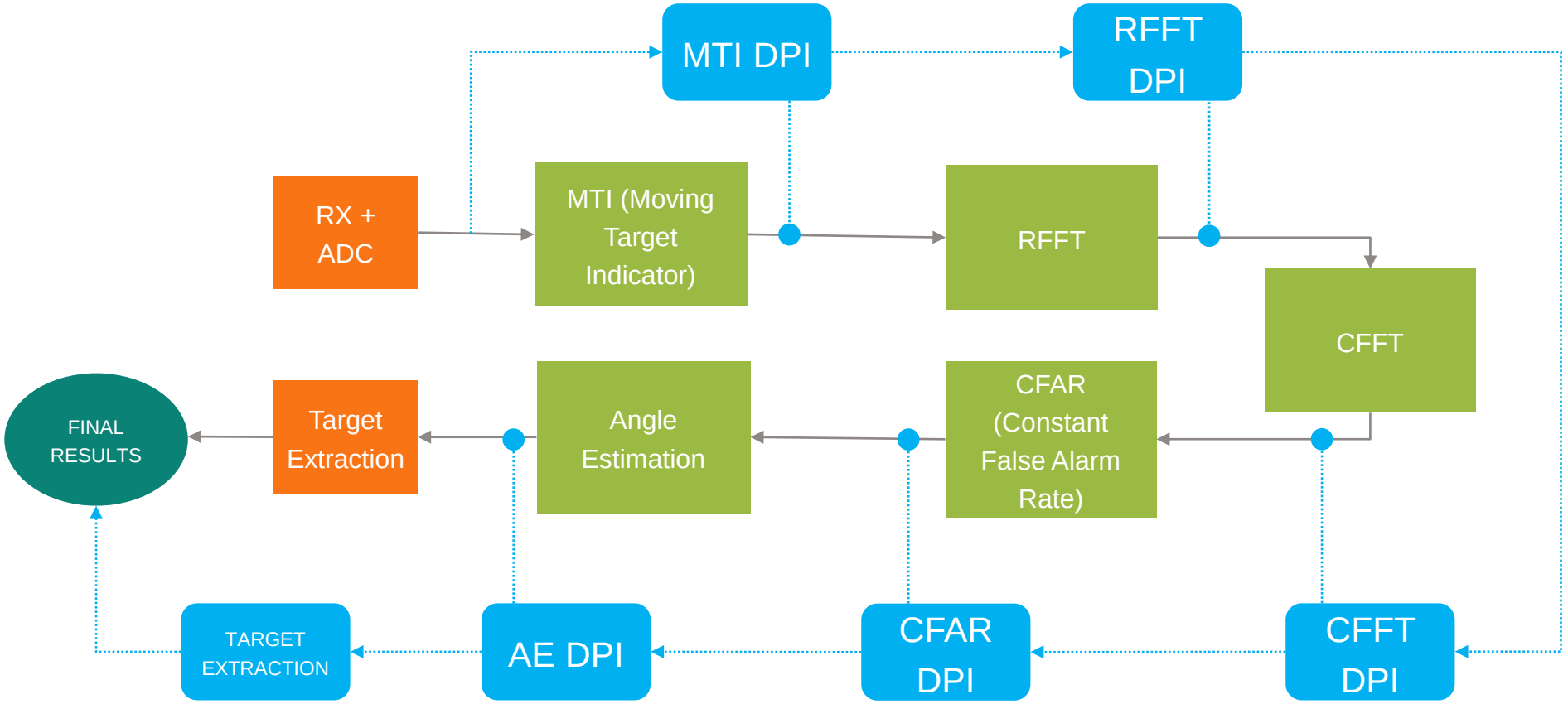
- All operations are performed in fixed-point format (different Q formats used)

- To align between different formats, multiple operations can be configured (sign extension, zero extension, LSB extension, etc.)

- In total, 15 algorithms involved to get the final result (translated into 15 DPI components)

RADAR SoC example – quick overview (2)

- Simplified processing chain:



RADAR SoC example – advantages of MATLAB based SV DPIs

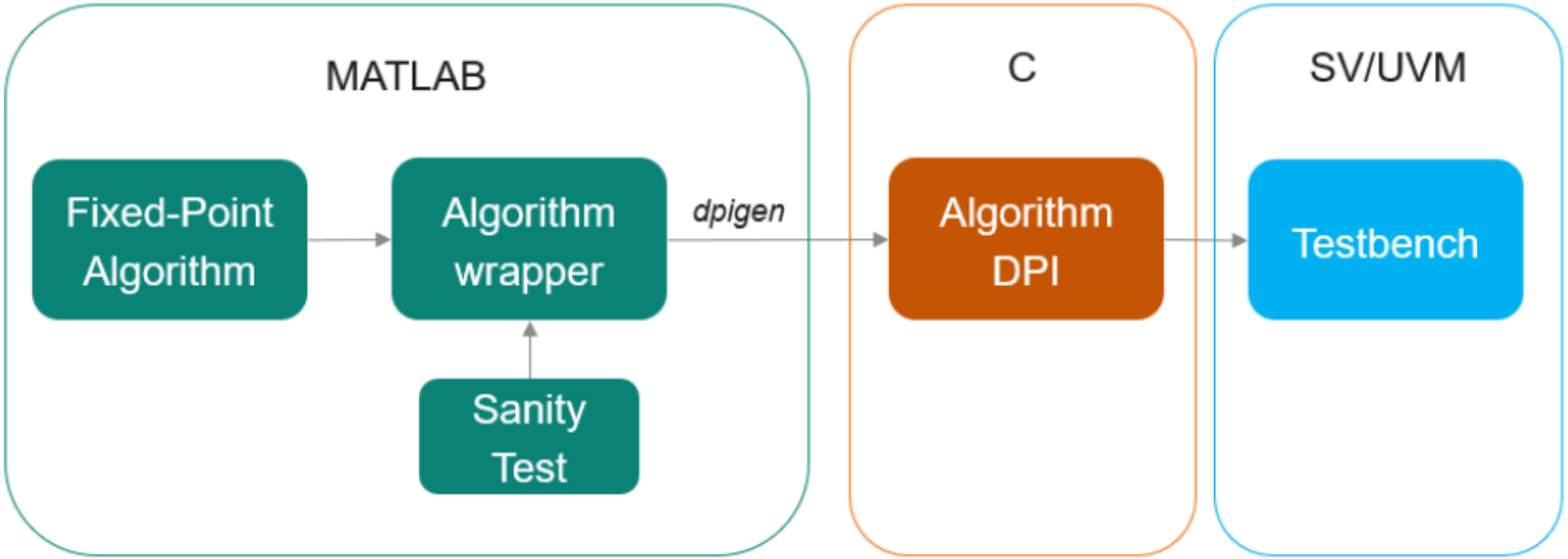
- Transition from SystemVerilog floating point model to MATLAB fixed point models
  - Verification started with a MATLAB floating point model generating expected results (limited set of configuration)
  - To allow randomization, the verification team implemented a SystemVerilog model
  - In the second phase of the project, the SystemVerilog models has been replaced by the MATLAB fixed point model (used to generate the SystemVerilog DPIs)

- RADAR SoC DSP performs many operations, which can be easily implemented with MATLAB
  - Examples: Matrix operations and calculation of trigonometric functions

- MATLAB provides functions and tools for fixed point implementation (high level of abstraction)

- Complexity and number of feature in RADAR SoC continuously incremented over time
  - Effort to extend and maintain the SystemVerilog model significantly increased
  - Verification team was provided with a reference model – reuse of models from concept team

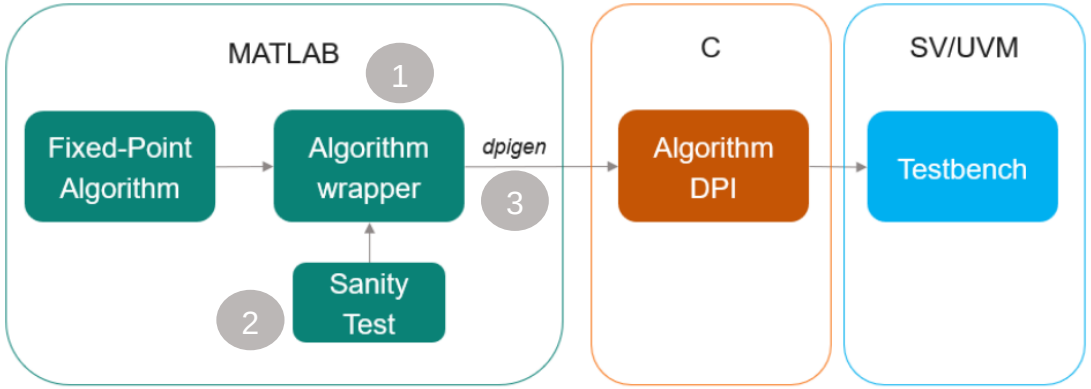
Translate MATLAB code into DPI components (1)



Translate MATLAB code into DPI components (2)

1. The given fixed point algorithm was embedded in a wrapper function
   - Align the arguments datatypes/format, create complex numbers, etc.
2. Sanity test to check that no errors have been introduced in the wrapper
3. Wrapper function converted into a DPI component using the MATLAB *dpigen* function
   - Some arguments: Input and output file locations, type of the input arguments, and a configuration object

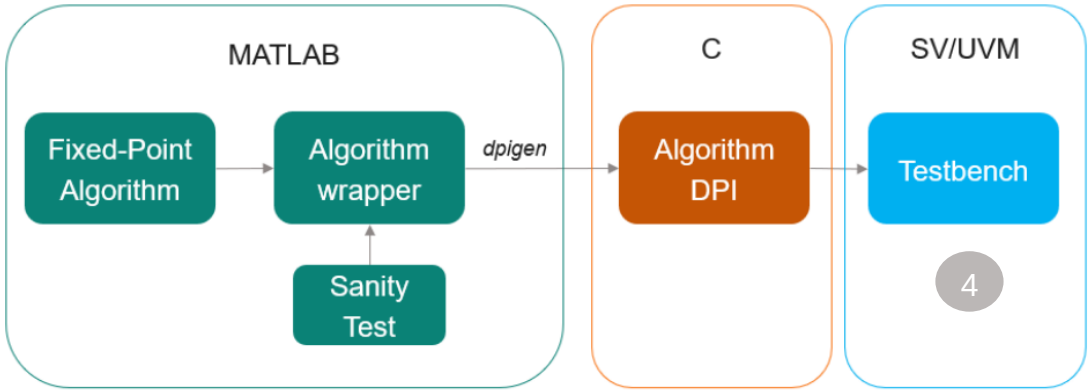
```
dpigen -args {int32(0)} -c alg_matlab_wrapper.m -d alg_dpi -config cfg_dpi
```



Translate MATLAB code into DPI components (3)

- The configuration object allows to customized a number of aspects of the generated DPI (target language, code appearance, debugging options, replacement with custom code, etc.)
- Output of the dpigen call:
  - C code and header files reflecting the functionality implemented in the MATLAB function
  - SystemVerilog package containing the *import "DPI-C"* declaration

4. Integrate the DPI call in the testbench
   - Compile C files and SystemVerilog package with Xcelium
   - Call the functions from the testbench



Translate MATLAB code into DPI components (4)

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

```
% 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
```

TEST

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

```
chandle objhandle = null;

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

```
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
```

Challenges

1. Successful and efficient usage of the DPI components is strongly dependent on the maturity of the MATLAB model
    - If the root of the issue is in the MATLAB code itself, debugging is complex
2. Very low visibility into the DPI component
    - Logging and reporting intermediate results from within the DPI is not easy
3. Not all MATLAB code can be translated into a DPI component (e.g. changing types through assignments, limited support for FOR loop indexes with unknown size)
    - Multiple iterations with the algorithm team to solve these issues and update the code
4. Matrices are not supported as DPI argument
    - In the RADAR SoC, most algorithms were working with matrices and many conversions matrix <-> vector were needed
5. Datatype difference between MATLAB and SystemVerilog
    - Many adaptions and reinterpretation of the arguments needed

Next steps and lessons learned

- Start early in the project with the reference model (high confidence that the model is correct)
  - Avoid the question “Is the bug in the RTL, in the MATLAB code, or in the testbench?”

- Provide guidelines to the developer of the MATLAB function
  - Align on the datatypes of the arguments, provide information about how to generate the DPI (avoid iterations due to unsupported code), etc.

- Improve visibility from within the DPI component. Examples:
  - Create log files with input data which can be easily used in a MATLAB test
  - Add interesting variables to the function outputs -> only feasible for simple functions


Infineon