Best Practice Guide for SystemVerilog DPI Component Generation

Best Practice Guide for SystemVerilog DPI Component Generation

Table of Contents

1 Getting Started

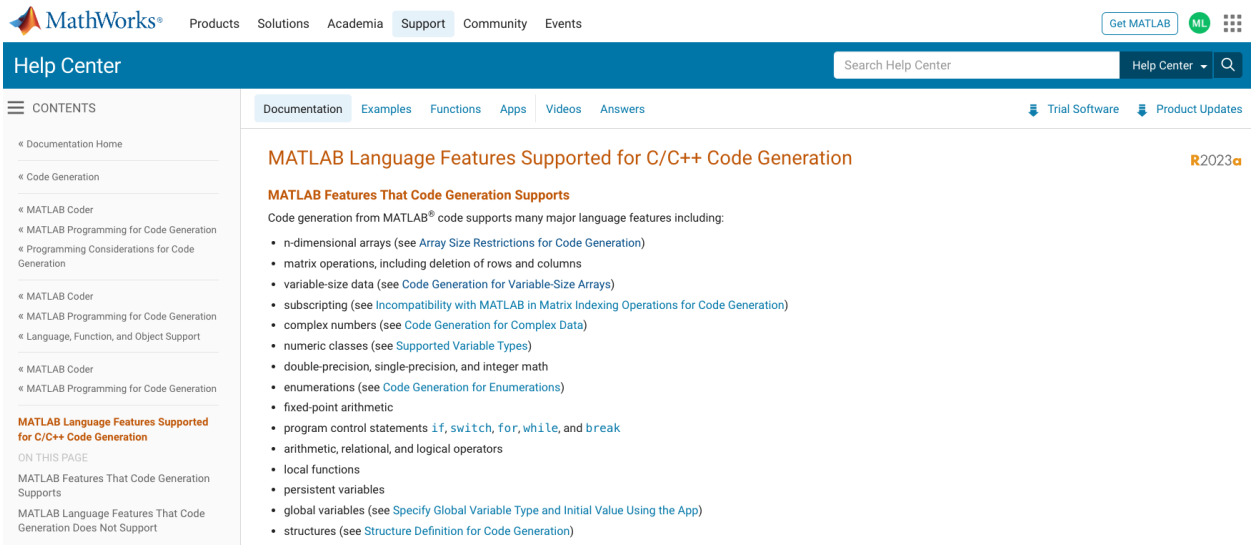
HDL Verifier™ generates SystemVerilog DPI (SVDPI) and Universal Verification Methodology (UVM) test bench components from MATLAB® or Simulink®, connecting algorithm development to your design verification environment. With its focus on the MATLAB workflow, this guide outlines recommended practices for preparing your MATLAB design for SVDPI component generation.

Beginning in R2023b, the DPI and UVM generation features are provided through the ASIC Testbench for HDL Verifier add-on. You can download and install the support package using the MATLAB Add-Ons menu or File Exchange.

1.1 C Code Generation

A MATLAB function must be capable of generating C code before it can be used to generate SVDPI components.

- Always use the `%#codegen` directive to indicate that your MATLAB function is intended for C code generation.
- Utilize `coder.screener` to identify MATLAB functions or features not supported for code generation.
- For more information, refer to MATLAB Language Features Supported for C/C++ Code Generation in the documentation for MATLAB Coder.



Learn the basics of C code generation from MATLAB with MATLAB Coder Onramp.

1.2 SystemVerilog DPI Component Generation

- Generate SVDPI component from MATLAB using the `dpigen` command. The `testbench` option also creates a testbench that you can use to verify the generated SVDPI component in an RTL simulator:

```
>> dpigen -testbench <testbench name> <function name> -args {<args type>}
```

- Ensure the data types of your top-level MATLAB function are supported for SVDPI generation. Consult the documentation Considerations for DPI Component Generation with MATLAB for supported data types.
- You can optionally run software-in-loop (SIL) simulations to verify the generated shared library (.so) using Embedded Coder®.

1.3 Converting Script to Function

Example path: ./examples/1_3_convert_script_to_function

MATLAB scripts are commonly used to setup, initialize, and drive a particular model. However, a script must be converted into functions before it can be used for code generation and in RTL simulation. The provided example demonstrates the best practices for partitioning a script based on the functionality of different DPI components.

- **test_script.mlx** is organized into sections where individual MATLAB functions are called sequentially to implement stimulus, golden reference, and checker logic.
    - **genFrame2Sample.m** creates stimulus for the design. The selection of input type is done through a *mode* input variable. The function further calls **genStim.m** to generate a frame and then performs frame to sample conversion.
    - **dutFFT.m** implements the design behavior using the `dsphdl.FFT` System object™ from the DSP HDL Toolbox™.
    - **fftChecker.m** gets input from the stimulus function and calculates the expected output. It then compares the expected output against the design output for self-checking.
- Selected function variables are driven as function outputs to allow signal debugging in an HDL simulator.
- **gen_dpi.m** provides the commands to generate SVDPI components for the stimulus and the checker.

1.4 Cross-Platform Workflows with Windows and Linux

You can port generated SVDPI components from Windows to Linux. For more information, see Generate Cross-Platform DPI Components in the documentation.

2 MATLAB Modeling Guidelines

2.1 Variable Definition

MATLAB is a dynamically-typed language; the data type, size and complexity of a given variable can change during MATLAB execution. In contrast, C is statically typed. For C code generation, you must explicitly define variable types and sizes before using them in operations or returning them as outputs.

- After a variable is used (read), changing the size of the variable can cause a size mismatch error. Use `coder.varsize` to specify that the size of the variable can change in the generated code.
- Each occurrence of a variable can have only one type. For example, a variable that is defined as double (explicit definition) or assigned a double value (implicit definition) cannot be assigned an integer value later in the MATLAB code.
- You must assign variables explicitly on all execution paths or provide a default value, as shown in the following example:

```
function out = limit(in)
out = in; % default
if in > 10
    out = 10;
end
```

- For additional guidelines, see the documentation on Best Practices for Defining Variables for C/C++ Code Generation.

2.2 Control Loops

Because MATLAB in inherently an untimed language, control loops such as `for` and `while` are commonly used to model time steps in MATLAB. The function below is one such example.

```
function status = badfibLFSR(SEED, LOAD)

persistent SR;
if isempty(SR)
    SR = uint16(SEED);
end

if (logical(LOAD))
    SR = uint16(SEED);
else
    % Fibonacci LFSR
    for n = 1:10   % All timing steps are lost in RTL simulation
        SR = bitshift(SR,1) + ...
            uint16(xor(xor(xor(bitget(SR,16), ...
            bitget(SR,14)), ...
            bitget(SR,13)), ...
            bitget(SR,11)));
    end
end

status = SR;
```

Because SystemVerilog functions take zero simulation time, a for-loop written like this would be completed instantly in the generated SVDPI component rather than over 10 time steps. To model algorithms that retain states

between time steps, you should use persistent variables and move the for-loop outside of the function, as shown below. You can then generate SVDPI from **fibLFSR()** and simulate it in an RTL testbench where the timing information is provided.

```
for T = 1:endoftime
    fibLFSR(SEED, LOAD)
end

function status = fibLFSR(SEED, LOAD)

persistent SR;
if isempty(SR)
    SR = uint16(SEED);
end

if (logical(LOAD))
    SR = uint16(SEED);
else
    SR = bitshift(SR,1) + ...
        uint16(xor(xor(xor(bitget(SR,16), ...
        bitget(SR,14)), ...
        bitget(SR,13)), ...
        bitget(SR,11)));
end

status = SR;
end
```

2.3 Frame to Stream

Example path: ./examples/2_3_frame_to_stream

It is common practice to process data as frames in high-level modeling. When interfacing with a hardware component, frames are typically deconstructed and delivered as scalar elements. The provided example demonstrates a way to convert from frames to samples.

- The **genFrame.m** function generates a frame of data where the size of the output vector depends on the input parameter *nsamples*.
- The **genSample.m** function serves as a wrapper to **genFrame**. It uses a state machine and a counter (modeled as a persistent variable) to loop through the frame elements and output the data as a stream.

2.4 Vectors and Matrices

Please consider these guidelines for working with vectors and matrices when performing SVDPI code generation:

- Define a matrix before assigning value to any of its elements.
- Avoid matrix resizing, as this results in delete/copy in C code. Use bounded worst-case, instead of unbounded dimensions (e.g. `[:5,:5]` instead of `[:inf,:inf]`). This avoids the need for C-functions like *malloc* and *calloc* for resizing operations.
- Use element wise operators (`.*`, `.+`) as much as possible, and avoid looping through the matrix.
- In many cases, you can do in-place operations in MATLAB by using the `x = foo(x)` syntax. That is, you specifically use the same variable name as both the return value and the incoming argument. Where

possible, MATLAB takes the original `x` as an argument by reference, and just changes its elements, as opposed to making a separate copy as the return variable.

- Disable the **preserve array dimensions** option in your configurations, as shown below. The option results in more readable code by using nested for-loop for each dimension. Disabling the option generates faster code with fewer operations associated with loop iterations.

```
cfg = coder.config('dll');
cfg.PreserveArrayDimensions = false;
```

2.5 Variable-Sized Vectors and Matrices

Example path: ./examples/2_5_variable_sized_vectors_matrices

Whenever possible, use only fixed-size vectors and matrices. If a variable-sized vector is necessary, see the example Use Variable-Sized Vector in SystemVerilog DPI Component. Variable-sized matrices are not supported; instead, you can write a wrapper function to convert a matrix to a variable-sized vector. For example:

```
function res = innerFcn(matrixA, matrixB)
res = horzcat(matrixA, matrixB);
end
```

Wrap the **innerFcn()** function to create a variable-sized vector.

```
function [vectorRes, sizeRes] = outerFcn(vectorA, sizeA, vectorB, sizeB)

res = innerFcn(reshape(vectorA, sizeA), reshape(vectorB, sizeB));
sizeRes = size(res);
vectorRes = reshape(res, [1 prod(sizeRes)]);

end
```

2.6 Logical Bit Shifting

The `bitshift()` function is commonly used for shifting bits, where fixed-point data type overflows are automatically handled.

Logical shift operations require manual handling. For efficient code generation, use `bitsll(a,k)` for logical shift-left operations and `bitsrl(a,k)` for logical shift-right operations on fixed-point data types.

2.7 Floating- and Fixed-Point Data Types

The use of floating-point (double) vs fixed-point data types has different considerations for SVDPI generation. Double data types simulate faster than fixed-point datatypes. There are also functional differences, such as cycle accuracy or threshold check in the generated code.

- For bit-accurate RTL simulations, use fixed-point data types – i.e. `fi()` – throughout the function. This eliminates quantization-related errors.
- To accelerate MATLAB function execution, use the `fiaccel()` function or use a MEX function.
- For tolerance-based comparison, generate the SVDPI component using a floating-point data type.

- If a fixed-point interface is required, apply the `fi()` function. For example:

```
function result = foo(A, B)

p = 3.14;

t = A + B;
t = p * t;

result = fi(t,1,32,30); % Convert only at output
```

2.8 Providing Visibility into SVDPI Components

Whether a SVDPI component is serving as a behavioral reference model or a UVM component, providing internal visibility makes testbench development faster and debugging efforts easier. Points of interest within the MATLAB function that is used to generate SVDPI component must be brought out as function output arguments. Follow these guidelines on driving the output arguments:

- Assign value at the end of the function.
- Changing or setting of data type also at the end of the function.
- Avoid passing char or string as outputs.

2.9 Considerations for Constrained Randomization

Randomization is not equivalent between MATLAB and RTL simulation. A random sequence with the same seed value does not produce the same random value. Avoid randomizing any internal variables in the MATLAB function. Instead, you should adhere to these methods:

- Replace the internal variable with an input argument to the function with a default initial value.
- Randomize the input argument in your MATLAB testbench.
- Generate a SVDPI component from the MATLAB function.
- Create a random constraint block in your SystemVerilog testbench to drive the inputs to the SVDPI component.

2.10 Generating UVM components

Beginning in R2023b, you can use MATLAB to generate standalone UVM component such as *uvm_sequence*, *uvm_scoreboard* and *uvm_predictor* blocks, and customize them through UVM templates. See the following documentation sections for more details.

- Use Templates to Create SystemVerilog DPI and UVM Components
- SystemVerilog and UVM Template Engine

3 Additional Guidelines

3.1 Generating SVDPI Component for MATLAB Code with MEX Function Calls

Example path: ./examples/3_1_generate_SVDPI_with_MEX

MEX functions are sometimes used in MATLAB code to import external C/C++ programs or to accelerate fixed-point simulation. Follow the guidelines below when generating SVDPI components for MATLAB code that includes MEX function calls.

To import external C code:

- Use `coder.ceval()` to call the external C code in your MATLAB function.
- Generate a MEX file for the MATLAB function that calls the external C code. See Generate MEX Functions by Using the MATLAB Coder App for more information.
- Create a wrapper MATLAB function that uses `coder.target()` to check if the code is executing in MATLAB. Then, generate an SVDPI component from this wrapper function.

```
function y = wrapper_moving_avg(x)

if coder.target('MATLAB')
    y = call_moving_avg_mex(x);
    disp('Function running in MATLAB');
else
    disp(coder.target());
    y = call_moving_avg(x);
    fprintf('Function running in generated code \n');
end
```

Using MEX in MATLAB code to speed up fixed-point simulation:

- Use the `fiaccel()` function to create a MEX function for the fixed-point code you wish to accelerate.
- Use the `coder.target()` function to call the MEX function in MATLAB, or include the original MATLAB code when generating SVDPI component.

```
function y = wrapper_function(x)

if coder.target('MATLAB')
    % call MEX function
else
    % call original MATLAB function
End
```

3.2 Modeling UVM Driver and Monitor

Make use of Simulink for modeling UVM driver and monitor components where time and cycle accuracy are important. MATLAB can be used within the Simulink subsystem for non-time dependent tasks such error injection, type conversion, or refactoring data from frames to streams.

- MATLAB can be used to translate transactions into forms required for the interface within the Driver and Monitor. This MATLAB code can be imported in Simulink as MATLAB function blocks.
- Make use of Simulink to apply the translated transaction at the signal level as Simulink is better equipped to handle time and perform multi-rate modeling.
- UVM driver and monitor code can then be generated as shown in Include Driver and Monitor in UVM Test Bench
- In case there is flow control between the DUT and drivers or monitors, it is recommended to extend the generated code for driving/monitoring the bus interface transactions.

3.3 Parameterizing Generated Code

Example path: ./examples/3_3_parameterize_generated_code

A MATLAB function's input argument may represent dynamic data that varies during run time or a static parameter that is set during the configuration phase. Given their indistinguishable nature, the `dpigen` function processes them identically while generating SVDPI components.

Alternatively, implementing the MATLAB function in Simulink via a MATLAB Function block enables clear differentiation between parameters and regular inputs. This is achieved by driving parameters with Constant blocks that use `Simulink.Parameter` variables. HDL Verifier converts these into constructs that are readily parameterizable in the generated SVDPI test bench.

The **matlab2simulink_dpigen.m** example demonstrates an automated approach to convert a MATLAB function into a Simulink model. If a struct is used as an input in the MATLAB function, it is implemented as a non-virtual bus in the Simulink model and preserved as a struct in the generated SVDPI component. For more information on utilizing **matlab2simulink_dpigen.m**, refer to the **matlab2simulink_dpigen_userguide.pdf** document.