MATLAB2SIMULINK_DPIGEN
User Guide

Description

The script matlab2simulink_dpigen was created to simplify taking a MATLAB function into a Simulink environment and from that use the dpigen function to generate a SystemVerilog DPI-C component. Using dpigen through Simulink instead from MATLAB makes it possible to keep structs in the generated DPI-C component and makes it possible to separate IOs into control ports and standard IOs.

Usage

To use the matlab2simulink_dpigen function you need to set various arguments in the calling of the function. Those arguments control what and how the DPI-C component is built but also sets some options for how the artifacts are zipped together.

The arguments are passed into the function as name value pairs, and the function accepts the following arguments:

`functionname = 'functionname'` : Character string representing the functionname

`ctrlports = {'port1', 'port2',…}` : Cell array containing strings of the portnames that should be set through DPI-C set_param calls instead of being passed into the DPI-C component arguments list. Can be empty.

`inputargs = {int8(0), fi(pi,1,11,5),…}` : Cell array containing the datatypes used in the calling of the original function

`outputstructs = {'name1', struct('a',1'…),…}`: Cell array containing name of output that has a struct as return type and it's definition

`outputfolder = 'nameofoutputfolder'` : Character string setting the name of where the output should be stored after running the matlab2simulink_dpigen

`packtype = 'flat' or 'hierarchical'` : Sets if the resulting zip-file should have a flat hierarchy or if it should be hierarchical.

`packfiles = true or false` : Should there be a zip-file created or not of the resulting C sources / header files

`makeTB = true or false` : If true then the model is only created but the dpigen is not run, as you need to set the simin_<inputsignal> timeseries so the model can be simulated in Simulink, but then you can generate SystemVerilog TB as well as the DPI-C component.

Taking a MATLAB function the script builds a Simulink top-level design that has a subsystem named the same as the MATLAB function, this so that the built DPI-C component name is the same. The Simulink subsystem then contains a c-function block that is the function you specify as input to the matlab2simulink_dpigen function. It also has constant blocks if you specify portnames in the ctrlports input. These constant blocks are set to be of value Simulink.Parameter, these are also set to have their storage class as "Model Default". During the dpigen process they will therefore get their own DPI_<functionname>_setparam_<ctrlportname>_f function that can be used from the testbench to set the value of that data. The other arguments of the function that are not in the

ctrlports array will be part of the DPI-C Components normal arguments, both for the DPI_<functionname>_output_f and DPI_<functionname>_update_f funtions.

If the function returns a struct, then that struct needs to be known for this script to work and needs to be given through the outputstructs option.

Example

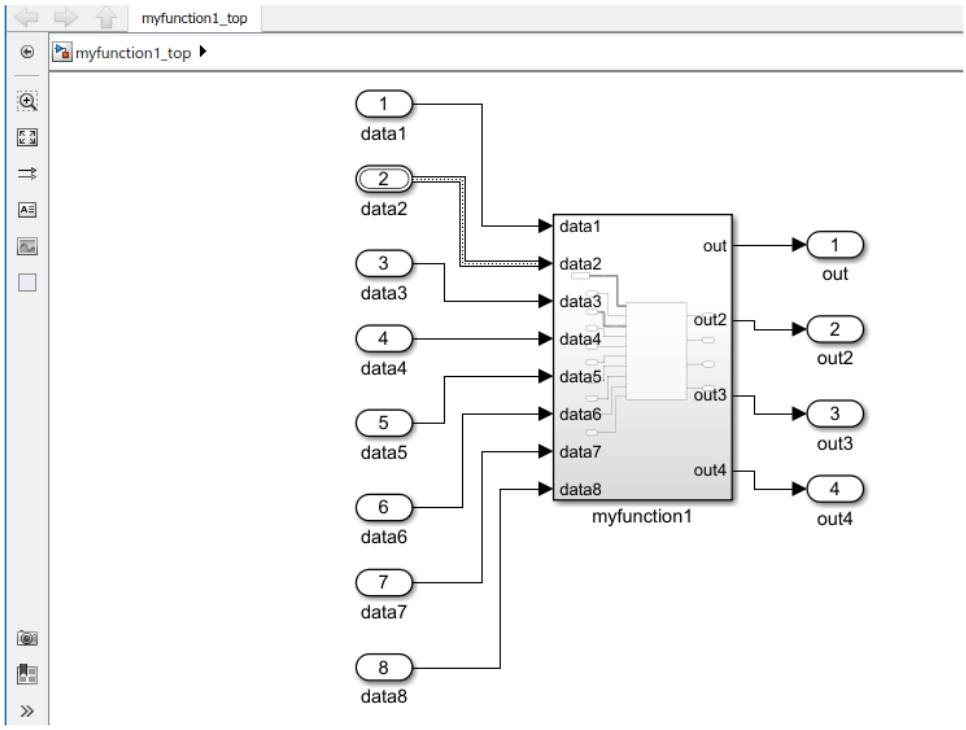
Taking a simple MATLAB function looking like:

```
function out, out2, out3, out4 = myfunction1(ctrl_reg, ...
    data1, data2, data3, data4, data5, data6, data7, ...
    ctrl_reg2, data8)
    tmp = 0;
    if ctrl_reg.valid1
        tmp = data1 + data2.one;
    else
        if ctrl_reg.valid2
            tmp = data1 - data2.one + data2.two;
        else
            tmp = data1 - data2.two;
        end
    end
    out = foo(tmp);
    out2 = data3 + data4;
    if ctrl_reg2>1
        out3 = data5 - data6;
    else
        out3 = data5+data6;
    end
    if data8
        out4 = sum(data7);
    else
        out4 = prod(data7);
    end
end
```

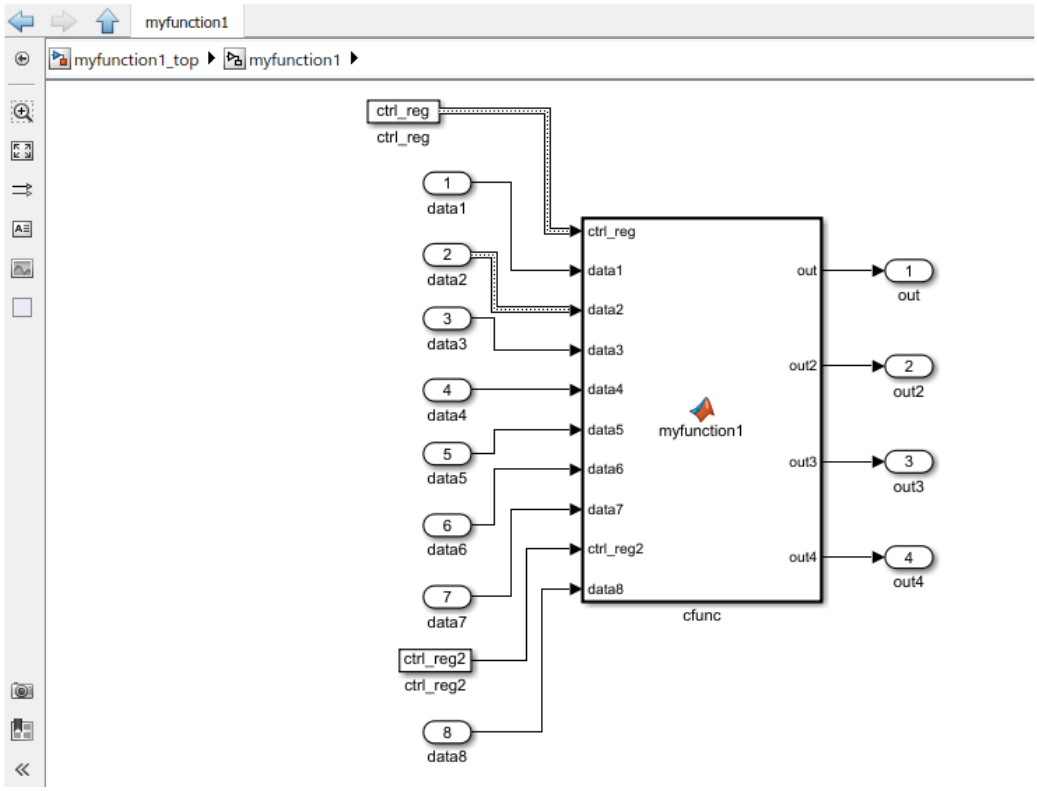
We can build this MATLAB function as a Simulink model and generate DPI-C component by calling the matlab2simulink_dpigen like this:

```
matlab2simulink_dpigen(...
    functionname = 'myfunction1', ...
    ctrlports = {'ctrl_reg', 'ctrl_reg2'},...
    inputargs = {struct('valid1', logical(true), 'valid2', logical(false)), ... ctrl_reg
        int8(0), ...                                    data1
        struct('one', int8(0), 'two', int8(0)), ... data2
        complex(0,0), ...                               data3
        complex(1,1), ...                               data4
        fi(pi, 1, 11, 5), ...                           data5
        fi(pi, 1, 11, 5), ...                           data6
        uint8(zeros(2,3)), ...                          data7
        uint8(7), ...                                   ctrl_reg2
        logical(true) ...                               data8
        }, ...
    DPIFixedPointDataType = 'BitVector', ...
    outputfolder = 'dpic_builddir' ...
    )
```

The Simulink model that is created will then be named myfunction1_top.slx and looks like this:



As can be seen here the two input arguments that were specified as ctrlports are not present in the top-level Simulink model. Going into the myfunction1 subsystem we can see the following:



Here both ports specified in the ctrlports are visible and set to be driven by constant blocks. All signals are then driving the cfunc block called myfunction1.

What also can be seen in the Simulink model is that the inputs that are specified as structs have been converted into Simulink buses.

When the DPI-C generation is done we can in the code that was generated see that the Simulink buses, are still structs in the myfunction1_dpi_pkg.sv file:

```
/*Simulink signal name: 'data2' Bus Object Name: 'data2_typeBus'*/
typedef struct{
             /* Simulink signal name: 'one' */
             bit signed [7:0] one;
             /* Simulink signal name: 'two' */
             bit signed [7:0] two;
}data2_typeBus;

/*Simulink signal name: 'ctrl_reg' Bus Object Name: 'ctrl_reg_typeBus'*/
typedef struct{
             /* Simulink signal name: 'valid1' */
             bit [0:0] valid1;
             /* Simulink signal name: 'valid2' */
             bit [0:0] valid2;
}ctrl_reg_typeBus;
```

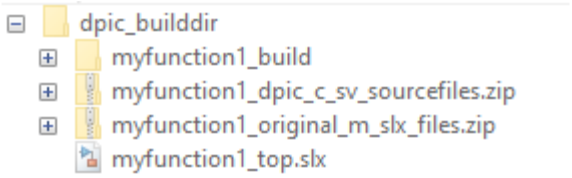
And for the DPI-C cfunctions that have been generated we see the following:

```
// Declare imported C functions
import "DPI-C" function chandle DPI_myfunction1_initialize(chandle existhandle);
import "DPI-C" function chandle DPI_myfunction1_reset_f(input chandle objhandle,
/*Simulink signal name: 'data1'*/
input bit signed [7:0] data1,
...
inout real out4 [3]);
import "DPI-C" function void DPI_myfunction1_output_f(input chandle objhandle,
/*Simulink signal name: 'data1'*/
input bit signed [7:0] data1,
...
inout real out4 [3]);
import "DPI-C" function void DPI_myfunction1_update_f(input chandle objhandle,
...
input bit [0:0] data8);
import "DPI-C" function void DPI_myfunction1_terminate(input chandle objhandle);

import "DPI-C" function void DPI_myfunction1_setparam_ctrl_reg_f(input chandle objhandle,
/*Simulink parameter name: 'valid1'*/
input bit [0:0] RTWStructParam_ctrl_reg_valid1,
/*Simulink parameter name: 'valid2'*/
input bit [0:0] RTWStructParam_ctrl_reg_valid2);
```

In the _output_f, and the update_f the control ports are not part of that interface but for those two control ports there are separate setparam functions generated:

```
 // Define SystemVerilog wrapper functions for tunable parameter native struct support for parameter:
ctrl_reg
function void DPI_myfunction1_setparam_ctrl_reg(input chandle objhandle,
/*Simulink parameter name: 'ctrl_reg'*/
input ctrl_reg_typeBus ctrl_reg);
             DPI_myfunction1_setparam_ctrl_reg_f(objhandle,ctrl_reg.valid1, ctrl_reg.valid2);
endfunction


import "DPI-C" function void DPI_myfunction1_setparam_ctrl_reg2_f(input chandle objhandle,
/*Simulink parameter name: 'ctrl_reg2'*/
input bit [7:0] RTWStructParam_ctrl_reg2);
```

In the output folder there are also two zip-files created:



The myfunction1_dpi_c_sv_sourcefiles.zip contains all files needed to compile the DPI-C functions on a machine without a MATLAB installation.

The myfunction1_original_m_slx_files.zip contains a copy of all m-files that were needed to generate the dpigen but also the Simulink top level slx file as well as the workspace variables needed for using the Simulink top level, they contain the bus definitions and sets the Simulink parameters for the control ports. There is also a runme.m script that can be used to build the dpi-c component from that Simulink toplevel system.

outputstructs

Say that your function returns a struct with two fields as:

```
function returnname = myfunction(in1, in2)
...
    returnname.a = uint8(in1);
    returnname.b = uint16(in2):
end
```

Then a call to matlab2simulink_dpigen needs to provide this outputstructs argument, example:

```
matlab2simulink_dpigen(...
    functionname = 'myfunction',...
    inputargs = {uint8(1), uint16(1)},...
    outputstructs = {'returnname', struct('a', uint8(1), 'b', uint16(1))}, ...
    DPIFixedPointDataType = 'BitVector', ...
    outputfolder = 'myfunction_builddir', ...
    makeTB = false);
```

makeTB

With this option there is no DPI-C component generated when you run this script, but in the output folder there is a script prepared, named “runme_sim_build.m”, that needs some manual edits before being run. The edits that need to be done is to add stimuli so the Simulink model can be simulated. When that has been added the script can be run and the DPI-C component is built but also a testbench is built that exercises the DPI-C component and compares the results with the results from the Simulink model.

As an example take this simple add_sub function:

```
function [outputArg1] = add_sub(mode, inputArg1, inputArg2)

    %UNTITLED Summary of this function goes here
    %   Detailed explanation goes here
    if mode
        outputArg1 = inputArg1 + inputArg2;
    else
        outputArg1 = inputArg2 - inputArg1;
    end
end
```

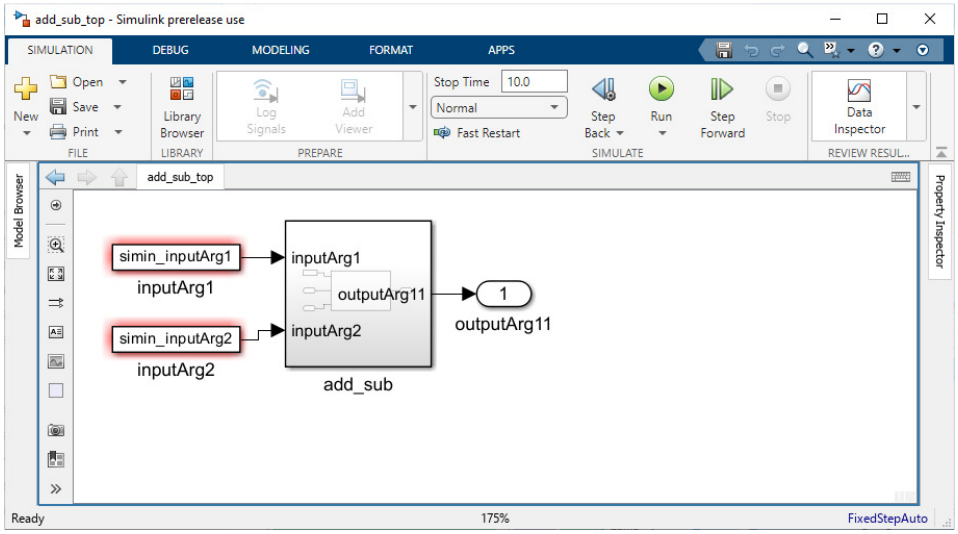
If we define the mode as a ctrl port when running the matlab2simulink_dpigen command with this:

```
matlab2simulink_dpigen(...
    functionname = 'add_sub',...
    ctrlports = {'mode'}, ...
    inputargs = {true, uint8(1), uint8(0)},...
    DPIFixedPointDataType = 'BitVector', ...
    outputfolder = 'add_sub_builddir', ...
    makeTB = true);
```

Then in the add_sub_builddir folder we can open the runme_sim_build.m script that looks like this:

```
load('baseWorkspace.mat')
open_system('add_sub_top')
simStopTime = 10; % TODO: Set to the number of inputs you have

% Here are the inputs that needs to be set before running this script
simin_inputArg1 = timeseries(0, 1:simStopTime); % TODO: Change to your data in
timeseries format
simin_inputArg2 = timeseries(0, 1:simStopTime); % TODO: Change to your data in
timeseries format

% Renaming add_sub.m so the generated DPI-C gets correct name
[folder, filename, ext] = fileparts(which('add_sub'));
org_file = fullfile(folder,[filename ext]);
tmp_file = fullfile(folder,[filename ext '.tmp']);
movefile(org_file, tmp_file)

cs = getActiveConfigSet('add_sub_top');
cs.set_param('StartTime', '0')
cs.set_param('StopTime', num2str(simStopTime))
cs.set_param('FixedStep', '1')


simout = sim('add_sub_top');
slbuild('add_sub_top/add_sub')

% Renaming add_sub.m.tmp to it's original name
movefile(tmp_file, org_file)
```

The Simulink model add_sub_top.slx that was built looks like this:



Here we can see that the model only has two inputs driven by two From Workspace blocks that reads from the workspace variable simin_inputArg1 and simin_inputArgs2 resepctively. The mode control port has been turned into a Simulink.parameter and is tied to a constant block inside the add_sub subsystem.

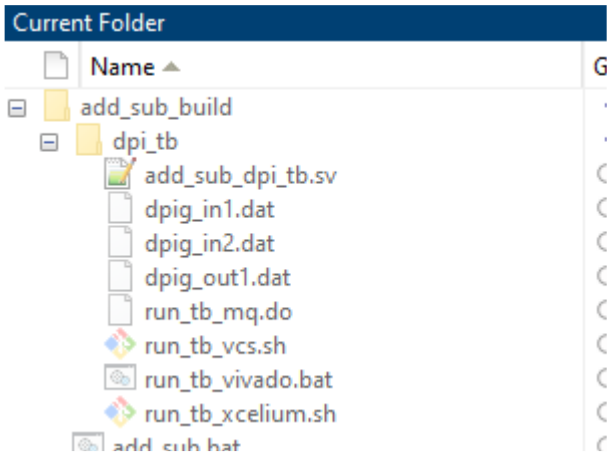
If we modify the runme_sim_build to define some input stimuli for simin_inputArg1 and simin_inputArg2 like this:

```
simin_inputArg1 = timeseries(uint8(1:simStopTime), 1:simStopTime); % TODO: Change to your
data in timeseries format
simin_inputArg2 = timeseries(uint8(randi(20,1,simStopTime)), 1:simStopTime); % TODO: Change
to your data in timeseries format
```

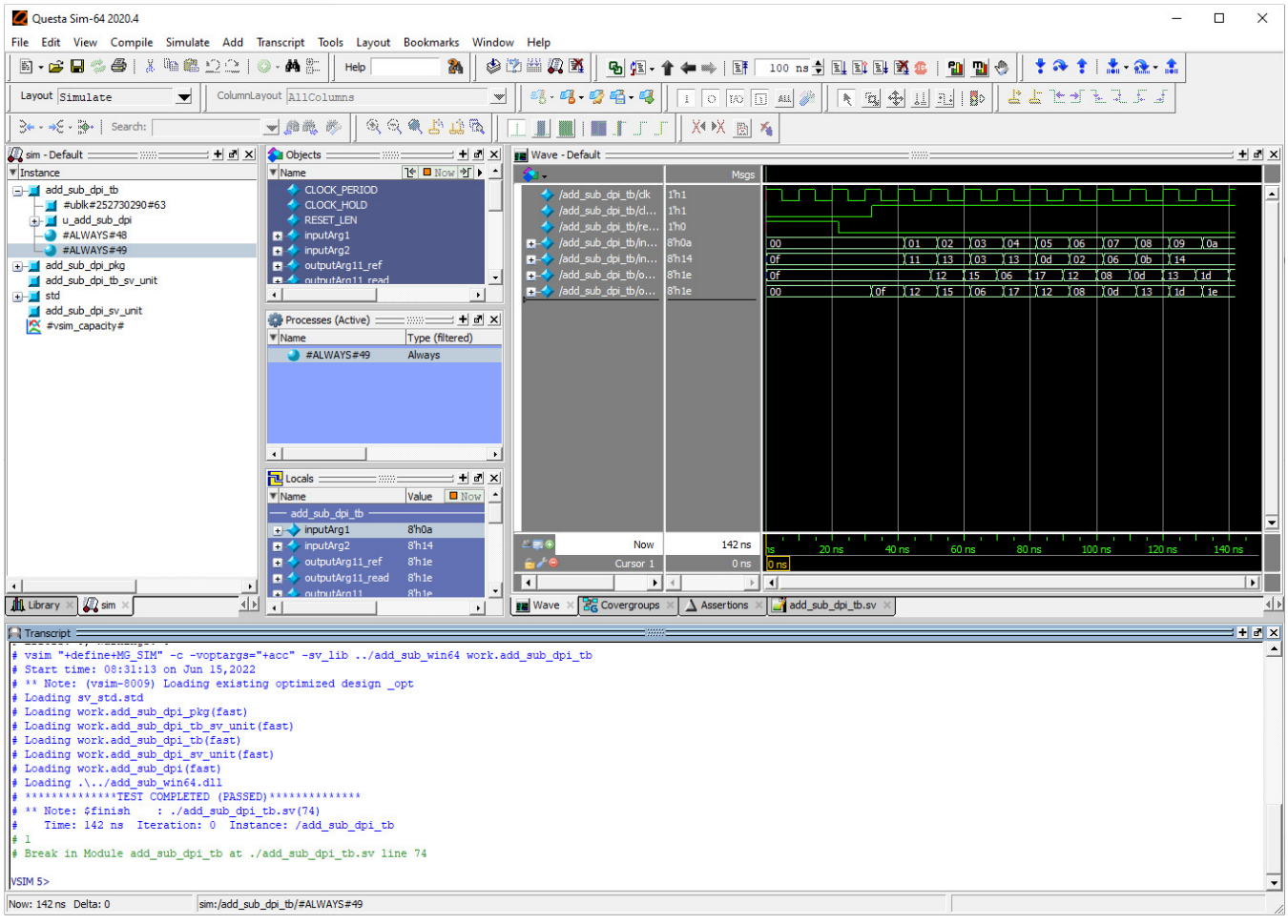
Then we can run this script and when being run it builds the DPI-C component but it also simulates the Simulink model as well as captures all data going in and coming out of the model and produces a testbench for us that can be run in a RTL simulator:

```
### Starting build procedure for model: add_sub

### Generating code and artifacts to 'Model specific' folder structure

### Generating code into build folder:
C:\designs\ericsson\matlab2simulink_dpigen\workdir\add_sub_builddir\add_sub_build

…

### Starting SystemVerilog DPI Component Generation

### Saving binary information cache.

### Generating DPI H Wrapper add_sub_dpi.h

### Generating DPI C Wrapper add_sub_dpi.c

### Generating SystemVerilog module package add_sub_dpi_pkg.sv

### Generating SystemVerilog module add_sub_dpi.sv

### Starting test vector capture

### Generating SystemVerilog test bench add_sub_dpi_tb.sv

### Running Simulink simulation to capture inputs and expected outputs

### Generating test bench simulation script for Mentor Graphics QuestaSim/Modelsim
run_tb_mq.do

### Generating test bench simulation script for Cadence Xcelium run_tb_xcelium.sh

### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh

### Generating test bench simulation script for Vivado Simulator run_tb_vivado.bat
```

The testbench is build into the folder



Running the RTL simulator with the run_tb_<RTLSimulator>.do/sh/bat will run this testbench and that runs the DPI-C component and compares the outputs with the outputs from the Simulink model and then gives a PASSED or failed result:



Now the testbench is built using the default setting for the mode control port. If you want to excersize different modes then you need to add a few lines into the generated testbench:

```
    parameter MODE_CHANGE_TIME = RESET_LEN + 2*CLOCK_PERIOD + 10*CLOCK_PERIOD; // Number of samples in mode1

    chandle objhandle=null;
```

```
initial begin
    objhandle = DPI_add_sub_initialize(objhandle);
                    #MODE_CHANGE_TIME;
                    DPI_add_sub_setparam_mode(objhandle, 1'b0);
end
```

In this part we have 10 samples for default setting of mode which is 1'b1 and after that we switch to mode being 1'b0.

And you also need to update the files dpig_in1.dat / dpig_in2.dat and dpig_out.dat so that they contain the correct data for input stimuli vs expected outputs.

Doing this for our add_sub we can see the following waveform:

dpig_in1.dat is changed:

From: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (one per each line without commas) to:

To: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (one per each line without commas)

And dpig_in2.dat is changed:

From: 0F, 11, 13, 03, 13, 0D, 02, 06, 0B, 14, 14 (one per each line without commas)

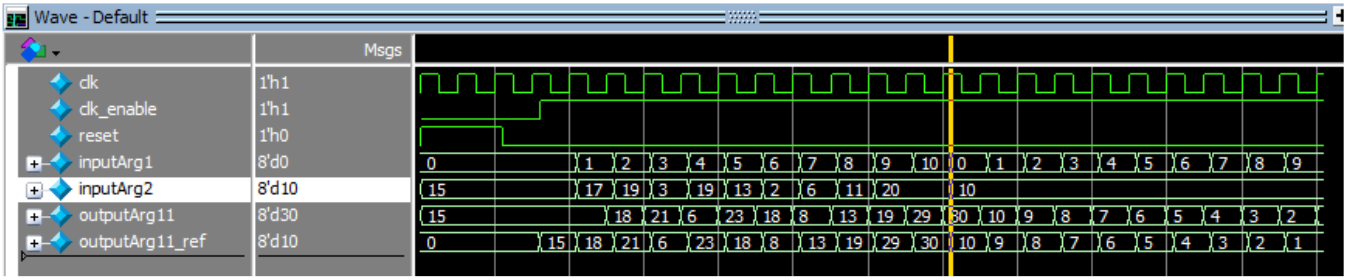
To: 0F, 11, 13, 03, 13, 0D, 02, 06, 0B, 14, 14, A, A, A, A, A, A, A, A, A, A (one per each line without commas)

And dpig_out.dat is changed as:

From: 0F,12, 15, 06, 17, 12, 08, 0D, 13, 1D, 1E (one per each line without commas)

To: 0F,12, 15, 06, 17, 12, 08, 0D, 13, 1D, 1E, A, 9, 8, 7, 6, 5, 4, 3, 2, 1 (one per each line without commas)

Then when we run the generated testbench



At time 142ns we change the mode and the adder becomes a subtractor, and we also get a passing result:

```
# Loading work.add_sub_dpi_tb(fast)
# Loading work.add_sub_dpi_sv_unit(fast)
# Loading work.add_sub_dpi(fast)
# Loading .\../add_sub_win64.dll
# **************TEST COMPLETED (PASSED)**************
# ** Note: $finish    : ./add_sub_dpi_tb.sv(82)
#    Time: 242 ns  Iteration: 0  Instance: /add_sub_dpi_tb
# 1
```