Course 13

MATLAB® Interface in Active-HDL

Table of Contents

Introduction

Active-HDL provides a built-in interface that allows the integration of MathWork's intuitive language and a technical computing environment with Aldec's HDL-based simulation environment. The interface included in Active-HDL allows executing MATLAB® commands, calling M-functions, or transferring data to or from the MATLAB workspace. All operations are controlled from your HDL code. Communication with MATLAB is accomplished with a dedicated set of subprograms prepared for both Verilog and VHDL. At any level of a design hierarchy, you can pass commands to MATLAB (e.g. pass an expression to solve or call M-function), transfer HDL variables to MATLAB workspace, perform necessary operations, and transfer the results back to the HDL simulator.

The interface consists of:

- The aldec_matlab_cosim.dll dynamic-link library. The library contains a set of VHDL foreign subprograms and Verilog system tasks that allows execution of commands in the MATLAB environment and transfer data between the HDL simulator and the MATLAB workspace. The library is located in the <Active_HDL_Inst_Dir>/bin/ subdirectory.
- The VHDL `matlab` package compiled to the `aldec` library. The package contains foreign subprograms declarations.

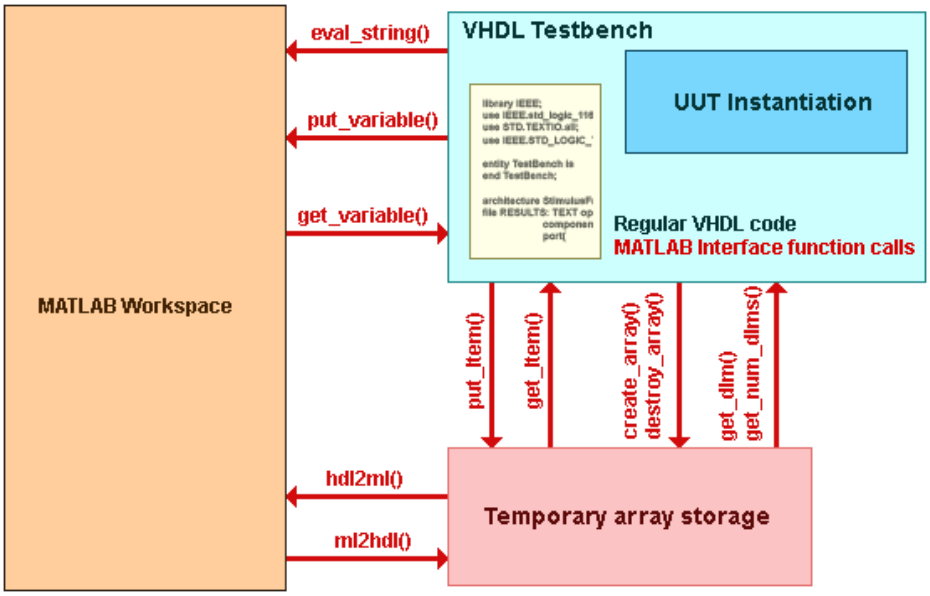

Figure 1 Active-HDL Co-Simulation Interface to MATLAB

Interface Specification

Software Requirements

- MATLAB R14 or newer
- Active-HDL 9.1 or newer

Language Support

- VHDL IEEE Std 1076-1993
- Verilog IEEE Std 1364™-1995

Supported VHDL Data Types

- STD_ULOGIC, STD_ULOGIC_VECTOR (up to 512 bits)
- STD_LOGIC and STD_LOGIC_VECTOR (up to 512 bits)
- BIT and BIT_VECTOR (up to 512 bits)
- SIGNED and UNSIGNED
- INTEGER
- REAL

Supported Verilog Data Types

- logic
- integer
- real

Supported MATLAB Data Types

- double
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32

Class and Cast Mnemonics

The class and cast mnemonics for VHDL are defined in the `matlab` package compiled to the `aldec` library. Class mnemonics are listed in the enumeration mxClassID type and cast mnemonics in the mxCastID enumeration type. When used in the Verilog source code, the mnemonic should be precede with the accent grave (`) character.

Conversion of Fixed Point Numbers

Fixed-point numbers at the HDL side are fully supported. For practical reasons, the number of bits is limited to 512. There are no limitations regarding the location of the decimal point.

Fixed point numbers at the HDL side are cast to floating point, double-precision numbers. Because the mantissa in a double number is 53 bits long, a fixed point number can be accurately translated to a floating point number at the MATLAB side as long as the number of significant bits in the fixed-point number is not greater than 53. If the number of significant digits exceeds 53, only the most significant 53 bits are accounted for.

The value of the fixed-point number can be calculated with the following formula:

$$\boldsymbol{A_{FIX} = 2^{-Bp} * A_{INT}}$$

Where: A_{FIX} *= fixed-point number.*

A_{INT} *= an integer number.*

Bp = the location of binary point.

The values can be scaled by moving the position of the binary point. When Bp is greater than 0, numbers with a fractional part are obtained, for example given AINT = 101; Bp = 1:

A_{FIX} = 2^{-1} * 101(bin) = 10.1(bin) = 2^{-1} * 5(dec) = 2.5(dec)

Note: For a practical example of using Aldec Matlab interface functions please refer to the description of `get_variable, put_variable, get_item` and `put_item` functions *under Transferring Data between Active-HDL and MATLAB* and *Managing Array Values* sections below.

Interface Setup

Registering a COM Server

When you install MATLAB on your computer, it registers a COM server with the operating system. But in some cases you need to register a COM server before you start using the Interface to MATLAB on Windows manually. This can be done with the following command at the operating system command prompt:

```
matlab /regserver
```

The command should be entered once and you do not need to re-enter it after a system restart.

Setup Interface in MATLAB

In order to setup the interface, follow the configuration procedure described below:

1. Start MATLAB.
2. Change the Current Directory in the MATLAB window to the $aldec/Simulink directory.
3. Type 'setup' in the MATLAB Command Window and press Enter. The appearing warning dialog box prevents you from accidental removing previous versions of the Active-HDL Toolbox installed in MATLAB. Press Yes to continue. The following message should be displayed in the Command Window:

```
Welcome to Active-HDL Blockset Setup.

Removing previous version of Active-HDL Blockset from path:

C:\Aldec\Active-HDL 9.1\Simulink

Installing Active-HDL Blockset...

Adding Active-HDL Blockset path:

C:\Aldec\Active-HDL 9.1\Simulink

Active-HDL Blockset has been installed successfully.
```

4. After the setup is finished, you are able to use Active-HDL Toolbox immediately. The Active-HDL Toolbox comprises MATLAB and Simulink Interfaces

FFT Program Example

This training document is based on fft_analysis sample design that comes with Active-HDL installation. By default all the sample designs are installed at "C:/My_Designs". It is advised that you copy it to a separate directory where all user Active-HDL designs are kept.

In Active-HDL load the fft_analysis workspace file from the user directory. Now right click on **compile.do** file and click **Execute**. After **compile.do** file is successfully executed, right click on **run.do** file, click **Execute** and observe the resulting waveforms generated in MATLAB.

Open the VHDL test bench file "top_fft_tb.vhd" and notice how the `eval_string, put_variable, create_array, get_variable, put_item`, and `hdl2ml` commands are used.

Enable Interface Operations for Verilog and VHDL

For Verilog - Declaring PLI Library

MATLAB-related task and functions are located in *aldec_matlab_cosim.dll* library. This library is located in <Active_HDL_Inst_Dir>/bin folder. This library must be added to the list of PLI applications visible to the simulator. Otherwise the simulator will not be able to find (and execute) those tasks and functions.

To add the library to PLI applications, follow the steps below:

1. Go to menu **Design | Design Settings | Verilog PLI Application**.
2. Click on the **Open** box located on the upper right hand corner of the window. Go to <Active_HDL_Inst_Dir>/bin folder and select *aldec_matlab_cosim.dll* and click open.
3. The required PLI application can also be specified with `-pli` argument for the asim command. See Verilog PLI and VPI for details.

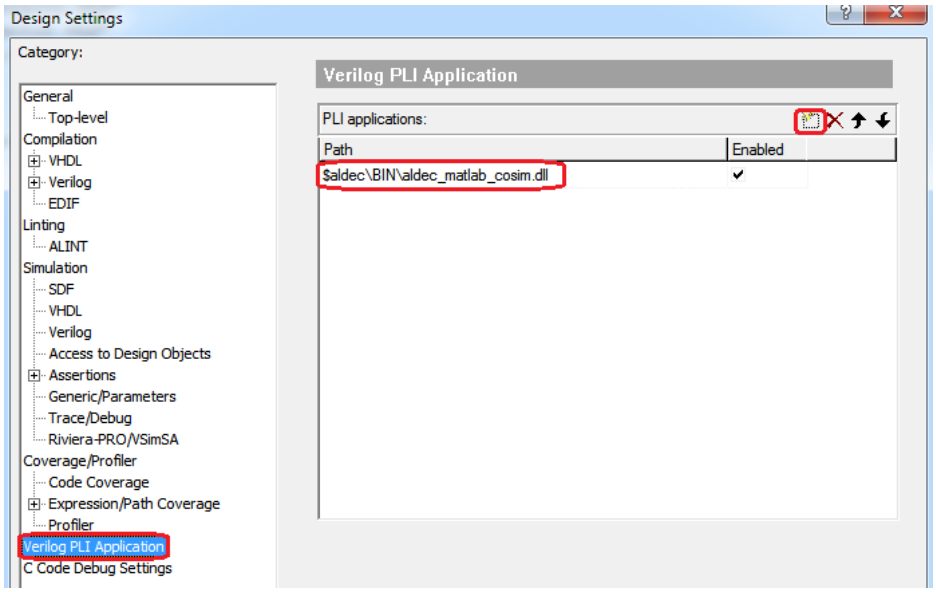

Figure 2 Setting Verilog PLI Application

For VHDL - Declaring MATLAB Packages

In VHDL, the routines for interfacing MATLAB are located in package `matlab` compiled to the `aldec` library. The package must be declared in VHDL source code. To declare the package, do the following:

- Use the following clause in VHDL source code:

```
library aldec;
use aldec.matlab.all;
```

- Make sure that the `aldec` library is visible to the simulator and package `matlab` is compiled into it. This can be done with the following commands:

```
alist
adir -lib aldec
```

Interface Functions

The MATLAB interface operation is based on calling interface functions within the HDL code. The MATLAB along with the co-simulation server is invoked automatically when the simulator executes the first interface subprogram call that is supposed to modify the MATLAB environment. The MATLAB interface is designed mainly to extend the testbench capabilities of HDL languages with the computing power of MATLAB environment; however, it is not limited only to testbenches. For example, design units that perform extensive calculations and impact HDL simulation performance can be partially written in M; an implementation written in M can be used for running functional simulation before RTL is defined.

The routines provided with the interface allow:

Controlling the MATLAB environment

- Example:`ml_setup`
- `ml_quit`

Transferring data between environments

- `get_variable`
- `put_variable`
- `put_simtime`
- `ml2hdl`
- `hdl2ml`

Managing arrays

- `create_array`
- `destroy_array`
- `get_item`
- `put_item`
- `get_num_dims`
- `get_dim`

Controlling MATLAB Environment

Functions in this group can be used to send commands to MATLAB®.

`ml_setup` is use to configure default values for arguments in functions responsible for data transfers between Active-HDL and MATLAB. You can also specify whether MATLAB should bring up its desktop when it starts.

`eval_string` procedure call is used to execute a MATLAB ".m" file as shown by a snippet of VHDL code below. Note that a MATLAB ".m" file is simply a collection of MATLAB commands that may be entered at the MATLAB command prompt. So a simple command that changes the directory in MATLAB is used in `eval_string` too, i.e. any MATLAB command that is executed at the command prompt is permitted in the `eval_string` command as part of the HDL testbench, whether VERILOG or VHDL.

Example:

```
MATLAB_INIT: process
variable dim_constr : TDims(1 to 2) := (1, 1024);
begin
ml_setup (1,0,$PATH/matlab.exe,51921)—- specifies matlab desktop or command window
```

```
                                        --path to executable and TCP port number for
                                        communication
eval_string("cd src"); --you can have any matlab command
    eval_string("prepare_workspace");  -- can pass entire .m file
```

Transferring Data between Active-HDL and MATLAB

The following routines are supported by the Active-HDL interface to pass and manipulate scalar values:

1. **put_variable** procedure call is used to send a VHDL constant, variable, or signal to MATLAB as shown by a snippet of VHDL code below. The put_variable routine passes a variable from the HDL simulator (Active-HDL) to the MATLAB environment. The routine can be used only for scalar values and vectors (i.e. one-dimensional arrays). It allows specifying the positions of the binary point, signed/unsigned and MATLAB object class.

Example 1: In this example variable, indexes and inputs will be passed to Matlab before evaluating windowed_input equation inside Matlab.

```
constant window_type : integer := 4;
signal in_wave : std_logic_vector (15 downto 0);
  MATLAB_WINDOW: process (clk)
variable i : integer;
begin
            put_variable ("w", window_type);
            put_variable ("input", in_wave);
            put_variable ("index", i);
           eval_string ("windowed_input=input*windows1024(w).coeff(index);");
end process;

Inside MATLAB: value of w = 4
```

Note: if "input", "index", or "windowed_input" is entered at the MATLAB command prompt, the value will be displayed as it was for "w".

Example 2:

```
signal Qin_slv : std_logic_vector(15 downto 0);
signal Qin_usgn : unsigned( 15 downto 0 );
signal Qin_bv : bit_vector( 15 downto 0 );


process (clk)
begin
  put_variable( "var1", Qin_slv, 5 );
  put_variable( "var2", Qin_usgn, 0 , mxSIGNED );
  put_variable( "var3", Qin_bv, -10, mxUINT16, mxUNSIGNED );
end process;
```

*The **Qin_slv** vector is interpreted in two's complement notation with the binary point set to 5, so the LSB weight equals 2^(-5) and positive and negative values are transferred. Values beyond the range of a 16-bit vector are saturated.*

*The **Qin_usgn** vector is interpreted in two's complement notation with the binary point set to 0, so values in range <-2^15; 2^15-1> and positive and negative values are transferred. Values beyond the range of a 16-bit vector are saturated. Note that default unsigned cast (as the VHDL variable is of unsigned type) is overridden.*

*The **Qin_bv** vector is interpreted in natural notation (which is default for bit_vector type) with the binary point set to -10. Therefore the LSB weight equals 2^10 and natural values are transferred. Values beyond the range of a 16-bit vector are saturated (zero is assigned for negative numbers). Note that default signed cast (as the VHDL variable is of the bit_vector type) is overridden. The MATLAB variable of the uint16 class is created.*

2. **get_variable** procedure call is used to assign a MATLAB value to a VHDL variable as shown by a snippet of VHDL code below. The get_variable routine allows passing a variable from the MATLAB environment to the HDL simulator (Active-HDL). The target HDL variable must be a scalar value or a vector (i.e. one-dimensional array). The routine supports scalar variables of floating-point (double and single) and integer (int8, uint8, int16, uint16, int32, uint32) types. It automatically recognizes the MATLAB class of the object.

Example 1: In this example new calculated values are taken back to HDL from Matlab environment

```
MATLAB_WINDOW: process (clk)
variable Qin_var : std_logic_vector(15 downto 0);
begin
       eval_string("windowed_input=input*windows1024(w).coeff(index)");
       get_variable("windowed_input", Qin_var);
end process;
```

Note: VHDL variable "Qin_var" from the MATLAB_WINDOW VHDL process is assigned the "windowed_input" MATLAB value.

Example 2:

```
signal Qin_slv : std_logic_vector(15 downto 0);
signal Qin_usgn : unsigned(15 downto 0);
signal Qin_bv : bit_vector(15 downto 0);

      process (clk)
      begin
        get_variable ("var1", Qin_slv, 5);
        get_variable ("var1", Qin_usgn, 0, mxSIGNED);
        get_variable ("var1", Qin_bv,-10, mxUNSIGNED);
      end process;
```

*The **Qin_slv** vector is interpreted in two's complement notation with the binary point set to 5, so the LSB weight is of 2^(-5) and positive and negative values are transferred. Values beyond the range of 16-bit vector are saturated.*

*The **Qin_usgn** vector is interpreted in two's complement notation with binary point set to 0, so values in range <-2^15; 2^15-1> and positive and negative values are transferred. Values beyond the range of 16-bit vector are saturated. Note that default unsigned cast (as the VHDL variable is of unsigned type) is overridden.*

*The **Qin_bv** vector is interpreted in natural notation (which is default for bit_vector type) with binary point set to -10, so the LSB weight is of 2^10 and natural values are transferred. Values beyond the range of 16-bit vector are saturated (zero is assigned for negative numbers).*

Note that default signed cast (as the VHDL variable is of bit_vector type) is overridden.

3. **put_simtime** procedure call transfers the VHDL simulation time to a MATLAB value, and is executed from a VHDL process as shown by a snippet of VHDL code below. It transfers simulation time from Active-HDL to a MATLAB variable.

Example: In this example VHDL simulation time is transferred to the "UPDATED_TIME" Matlab value.

```
ANALYZE_FFT: process (clk)
variable dim_constr : TDims(1 to 2) := (1, 1);
begin
    if rising_edge(clk) then
        if start = '1' then
            dim_constr(2) := 1;
        end if;
        if OE = '1' then
            put_simtime("UPDATED_TIME");
            dim_constr(2) := dim_constr(2) + 1;
        end if;
        if (dim_constr(2) = 1025) then
            dim_constr(2) := 1;
        end if;
    end if;
end process;
FROM MATLAB:
> UPDATED_TIME>
UPDATED_TIME = 0.1409
```

Managing Array Values

The MATLAB interface allows passing and manipulating multi-dimensional arrays with HDL routines. For efficiency, the arrays are created in the interface space and not copied to the HDL simulator. Thus routines are used to create arrays first and then data are transferred.

The following routines are provided:

1. **create_array** function call is used to create a temporary storage array from a VHDL process as shown by a snippet of VHDL code below. The temporary storage array is given an identifier (Id) and is transferred/updated to MATLAB via the hdl2ml procedure call as shown below.

Example:

```
---- Architecture declarations -----
shared variable Qin_Id : INTEGER := 0;
shared variable Iout_Id : INTEGER := 0;
shared variable Qout_Id : INTEGER := 0;
MATLAB_INIT: process
variable dim_constr : TDims(1 to 2) := (1, 1024);
begin
    Qin_Id:= create_array("fft_Qin", 2, dim_constr); -- see Note 1
    Iout_Id:= create_array("fft_Iout", 2, dim_constr); -- see Note 2
    Qout_Id:= create_array("fft_Qout", 2, dim_constr); -- see Note 3
            hdl2ml (Iout_Id);   --fft_Iout is sent to MATLAB
            hdl2ml (Qout_Id); --fft_Qout is sent to MATLAB
            hdl2ml (Qin_Id); --fft_Qin is sent to MATLAB
```

Note 1: fft_Qin is set to be a 2 dimensional array [1 X 1024] in temporary storage

Note 2: fft_Iout is set to be a 2 dimensional array [1 X 1024] in temporary storage

Note 3: fft_Qout is set to be a 2 dimensional array [1 X 1024] in temporary storage

```
FROM MATLAB:
> fft_Qin>
fft_Qin = Columns 1 through 12
               183 183 184 184 184 184 184 184 185 185 185 -186
               Columns 13 through 24
                ...........
               Columns 1021 through 1024
               184 184 183 -184
```

2. **`destroy_array`** procedure call is used to remove a temporary storage array from a VHDL process as shown by a snippet of VHDL code below. The known temporary storage array identifier (Id) must be given as shown below.

Example:

```
---- Architecture declarations -----
shared variable Qin_Id : INTEGER := 0;
MATLAB_INIT: process
variable dim_constr : TDims(1 to 2) := (1, 1024);
begin
    Qin_Id:= create_array("fft_Qin", 2, dim_constr); -- see Note 1
             hdl2ml (Qin_Id); --fft_Qin is sent to MATLAB
          destory_array(Qin_Id);  -- see Note 2
```

Note 1: fft_Qin is set to be a 2 dimentional array [1x1024] in temporary storage

Note 2: fft_Qin has been deleted / removed from temporary storage and therefore no updates possible to Matlab

Note: "fft_Qin" still remains in MATLAB but can't be updated

3. **get_item** procedure call reads a cell from MATLAB array copy stored in the interface memory and writes it to an HDL variable. The array must be first copied to interface memory from MATLAB using ml2hdl routine. Floating point (single and double) and integer (int8, uint8, int16, uint16, int32, uint32) arrays are supported. It automatically recognizes the MATLAB class.

Example:

```
process(clk)
  variable cell_loc : TDims(1 to 3) := (1, 1, 2);
  variable Qin_slv : std_logic_vector(15 downto 0);
  variable Qin_usgn : unsigned(15 downto 0);
  variable Qin_bv : bit_vector(15 downto 0);
begin
  get_item( Qin_slv, mxUNSIGNED, Qin_Id, cell_loc );
  get_item( Qin_usgn, 5, mxSIGNED, Qin_Id, cell_loc );
  get_item( Qin_bv, -10, Qin_Id, cell_loc );
  cell_loc(2) := cell_loc(2) + 1;
end process;
```

*The **Qin_slv** vector is interpreted in natural notation with the binary point set to 0, so all values in range <0; 2^16-1> are transferred. Values greater than 2^16-1 are saturated to a vector of ones and negative values are set to 0.*

*The **Qin_usgn** vector is interpreted in two's complement notation with the binary point set to 5, so that the LSB weight is of 2^(-5) and positive and negative values are transferred. Values beyond the range of 16-bit vector are saturated. Note that default unsigned cast (as the VHDL variable is of unsigned type) is overridden.*

*The **Qin_bv** vector is interpreted in two's complement notation (which is default for bit_vector type) with the binary point set to -10, so that the LSB weight is of 2^10 and positive and negative values are transferred. Values beyond the range of 16-bit vector are saturated.*

In all cases value is read from cell indexed with (1,1, 2) where i increments with every clock edge.

4. **put_item** procedure call is used to store a VHDL constant, variable, or signal in a temporary storage array and is executed from a VHDL process as shown by a snippet of VHDL code below. The known temporary storage array identifier (Id) must be given as shown below. The pointer or reference element must be given in order to properly populate the matrix.

Example:

```
---- Signal declarations ---
signal Qin : std_logic_vector (15 downto 0);
DISPLAY_INPUT: process (clk,reset)
variable dim_constr : TDims(1 to 2) := (1, 1); --see Note:1
begin
    if reset = '1' then
```

```
            dim_constr(2) := 1;
    end if;
            put_item(Qin, 0, Qin_Id, dim_constr); --see Note:2
            dim_constr(2) := dim_constr(2) + 1;
        if (dim_constr(2) = 1025) then
            hdl2ml (Qin_Id);   -- see Note 3
            eval_string("display_input");-- see Note 4
            dim_constr(2) := 1;
        end if;
end process;
```

Note 1:(row,column) of an (M X N) matrix, in this example M = 1 and N = 1024 and begin populating matrix at [1, 1] i.e. [row,col]

Note 2: 1 the VHDL signal "Qin" is pointed to "Qin[0]" as the starting point. It has each element assigned to the "fft_Qin" matrix as shown in the previous example. The temporary storage "fft_qin" 2 dimensional array reference element is given in the "dim_constr" value and starts at [1,1] --and goes to [1, 1024] in populating the matrix, so it is a row vector of 1024 elements.

Note 3: fft_Qin is sent to MATLAB, i.e. "fft_Qin" is updated so all previous values in MATLAB are replaced with new ones

Note 4: file "display_input.m" exists in the source directory.

5. **ml2hdl** function call transfers an array from MATLAB to a temporary storage array and is executed from a VHDL process as shown by a snippet of VHDL code below. `get_item` procedure call is then used to obtain a given element of the array and store it in a VHDL variable. The known temporary storage array identifier (Id) is obtained from the ml2hdl procedure call within the VHDL process. The pointer or reference element must be given in order to obtain the correct value from the MATLAB matrix or array.

Example:

```
---- Architecture declarations -----;
shared variable Iin_Id : INTEGER := 0;
ANALYZE_FFT : process (clk)
variable dim_constr : TDims(1 to 2) := (1, 1); -- see Note 1
variable Iin_test : STD_LOGIC_VECTOR(15 downto 0); -- see Note 2
    if rising_edge(clk) then
        if start = '1' then
            dim_constr(2) := 1;
        end if;
        if OE = '1' then
            Iin_Id:= ml2hdl("fft_Iin"); -- see Note 3
get_item(Iin_test,0,Iin_Id,dim_constr); -- see Note 4
            dim_constr(2) := dim_constr(2) + 1;
        end if;
     end if;
```

```
end process;
```

Note 1: (row,column) of (M x N) matrix, in this example M = 1 and N = 1024

Note 2: variable required to store begin element of array obtained from Matlab.

Note 3: fft_Iin is an array in MATLAB and that array is transferred to a temporary storage space and is accessed using the Identifier Iin_Id

Note 4: VHDL variable "Iin_test" is pointed to "Iin[0]" as the starting point. It has its single value assigned to the "fft_Iin" matrix element pointed to by "dim_constr". The temporary storage "fft_Iin" 2 dimensional array reference element is given in the "dim_constr" value and starts at [1,1] and goes to [1,1024] which is the MATLAB matrix, i.e. a row vector of 1024 elements.

6. **`get_num_dims`** function call obtains the dimensions of a MATLAB matrix (array) in temporary storage space, i.e. if given an (M X N X K) it will return a value of 3 within the VHDL process. `get_dim` function call will return the value for K if passed a 3, M if passed a 1, and N if passed a 2. Both function calls are executed from a VHDL process as shown by a snippet of VHDL code below.

Example:

```
---- Architecture declarations -----;
shared variable Iin_Id : INTEGER := 0;
ANALYZE_FFT: process (clk)
variable dim_constr : TDims(1 to 2) := (1, 1); --see Note 1
variable Iin_test : STD_LOGIC_VECTOR(15 downto 0); --see Note 2
variable Iin_total_dimensions : INTEGER;
variable Iin_size_last_dimension : INTEGER;
begin
    if rising_edge(clk) then
        if start = '1' then
            dim_constr(2) := 1;
        end if;
        if OE = '1' then
                Iin_Id := ml2hdl("EKF"); --see Note 3
       Iin_total_dimensions:= get_num_dims(Iin_Id); -- see Note 4
   Iin_size_last_dimension:= get_dim(Iin_Id,Iin_total_dimensions); -- see Note 5
       dim_constr(2) := dim_constr(2) + 1;
       get_item(Iin_test,0,Iin_Id,dim_constr);
        end if;
        if (dim_constr(2) =  Iin_size_last_dimension + 1) then
            dim_constr(2) := 1;
        end if;
    end if;
end process;
```

Note 1: (row,column) of an (M x N) matrix, in this example M = 1 and N = 1024

Note 2: variable required to store element of array obtained from MATLAB

Note 3: EKF is an array in MATLAB and that array is transferred to a temporary storage space and is accessed using the Identifier Iin_Id

Note 4: EKF is a [1 X 1024] matrix, so this procedure call will return the value of 2.

Note 5: the procedure call will return the value of 1024.

7. **get_time** procedure call obtains the MATLAB simulation time and is executed from a VHDL process as shown by a snippet of VHDL code below.

```
ANALYZE_FFT: process (clk)
variable dim_constr : TDims(1 to 2) := (1, 1);
begin
    if rising_edge(clk) then
        if start = '1' then
            dim_constr(2) := 1;
        end if;
        if OE = '1' then
            get_time;                          --Benchmark procedure
            dim_constr(2) := dim_constr(2) + 1;
        end if;
        if (dim_constr(2) = 1025) then
            dim_constr(2) := 1;
        end if;
    end if;
end process;
```

Co-Simulation & Result

When Simulation is initialized and run through Active-HDL, the 'MATLAB Command Window' opens, the Workspace Window is displayed, and the a set of plots visualize the analyzed data.

The connection to MATLAB is established automatically when any of the following subprograms is reached in VHDL code:

- `eval_string()`
- `put_variable()`
- `get_variable()`
- `put_item()`
- `get_item()`
- `hdl2ml()`
- `ml2hdl()`

All messages reported by MATLAB during co-simulation are displayed in the Active-HDL console window.

As the co-simulation advances the MATLAB Workspace window and a plot figure displaying subsequent contents of FFT input buffer (the plot title applied shows the type of window function) can be observed.

Right before the simulation ends a set of plots with analyzed data for selected window function is displayed.

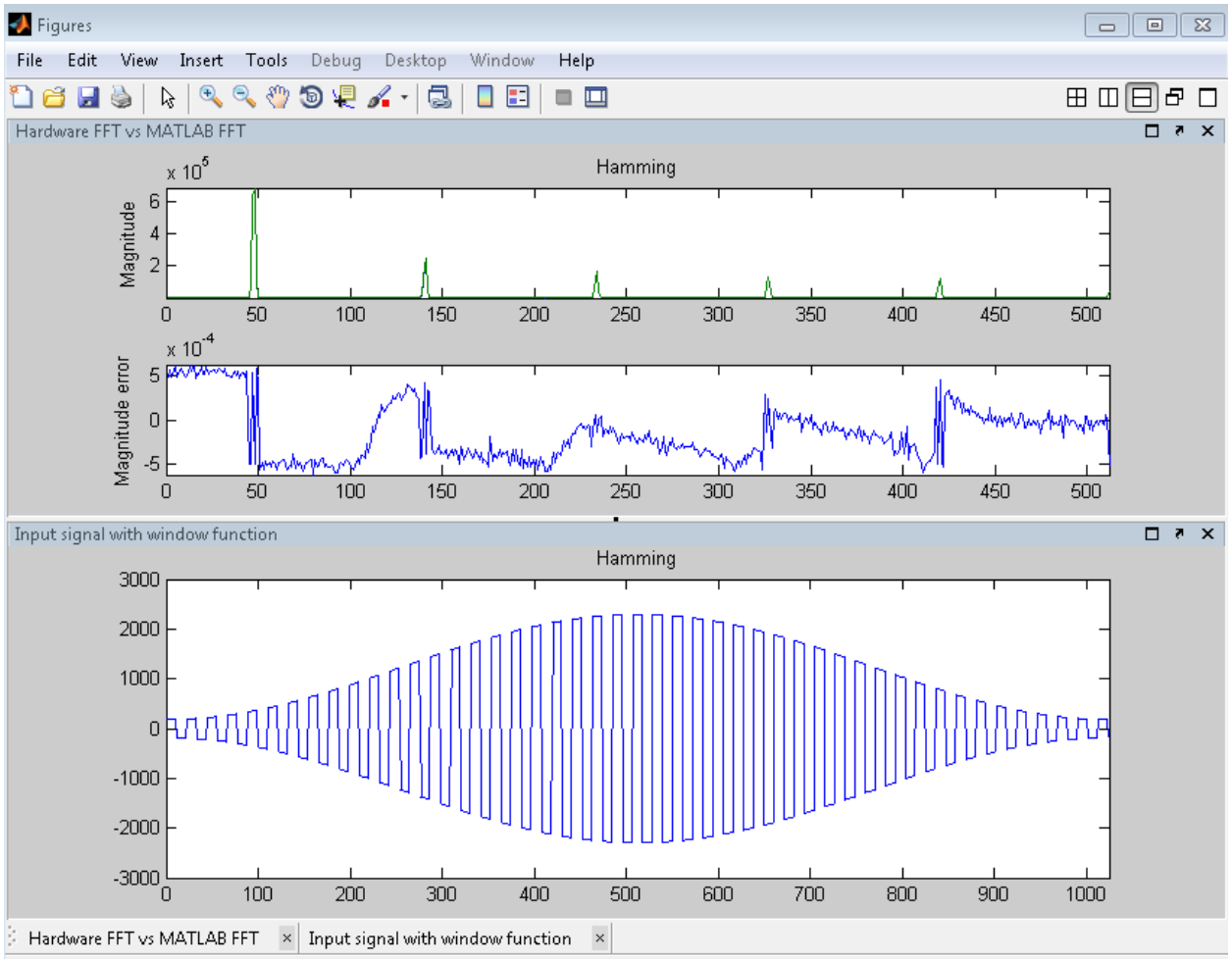

Figure 3 FFT output in MATLAB