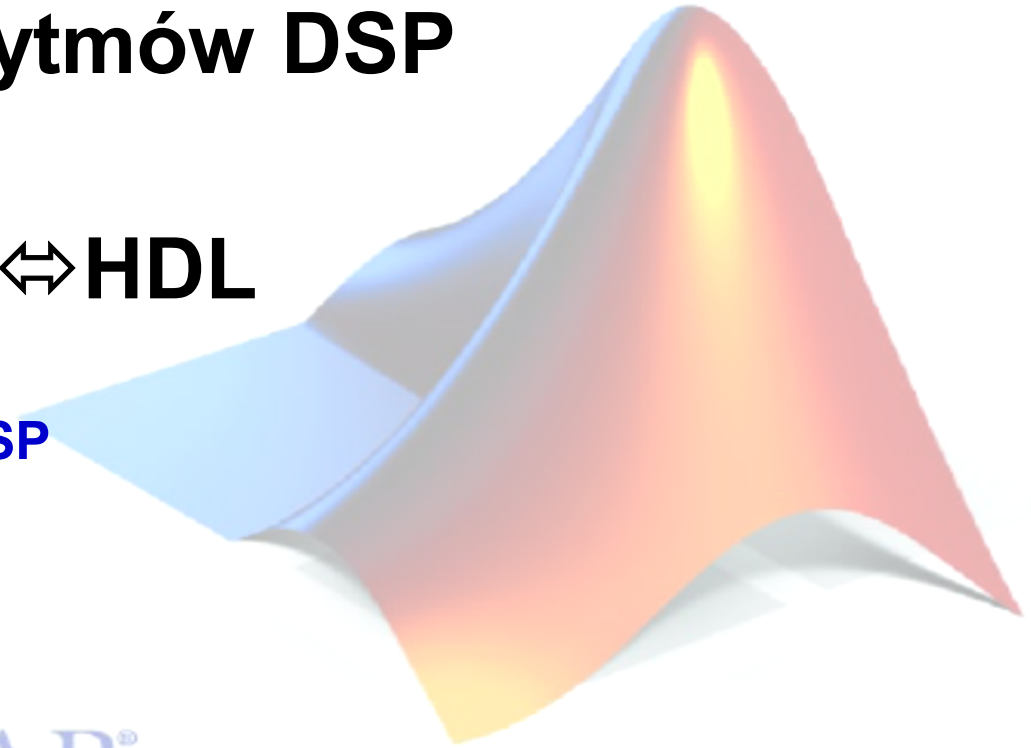
Implementacja algorytmów DSP w układach FPGA

Program wykładu

Implementacja algorytmów DSP w układach FPGA:

kosymulacja Matlab ⇔HDL

- Mathworks
- Xilinx System Generator for DSP
- Aldec AHDL MATLAB



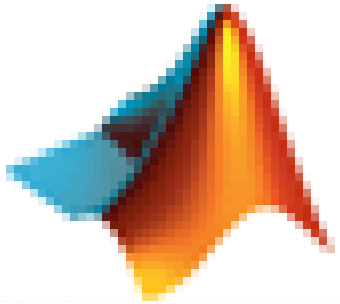
MATLAB®

ALDEC
The Design Verification Company

XILINX®

The MathWorks

Matlab HDL Coder™



Documentation

Search R2021b Documentation

CONTENTS

« Documentation Home
« DSP System Toolbox
« Code Generation

Category

C Code Generation
HDL Code Generation
DSP Algorithm Acceleration
SIMD Code Generation
Code Generation for ARM Cortex-M and ARM Cortex-A Processors

All | Examples | Functions | Blocks | Apps

HDL Code Generation

R2021b

Generate HDL code from MATLAB® and Simulink®

To implement a DSP design on FPGAs or ASICs, you can use either HDL Coder™ or Filter Design HDL Coder™. Both products generate synthesizable and portable VHDL® and Verilog® code, and also generate VHDL and Verilog test benches for quickly simulating, testing, and verifying the generated code.

- HDL Coder — Generate code from Simulink or MATLAB designs. This support includes filters, math and signal operations, and other algorithms optimized for resource use and performance, such as the FFT HDL Optimized, IFFT HDL Optimized, and NCO HDL Optimized blocks. For a basic example of how to generate HDL code using HDL Coder, see Programmable FIR Filter for FPGA.
- Filter Design HDL Coder — Generate code from MATLAB filter designs. You can access code and test bench generation features using the Generate HDL user interface, or by using command-line options. These features are also integrated with the Filter Designer app. For an example of how to generate HDL code using Filter Design HDL Coder, see HDL Butterworth Filter (Filter Design HDL Coder).

To debug your designs in Simulink or MATLAB, use the **Logic Analyzer** waveform viewer.

Simulink Visualization Tool

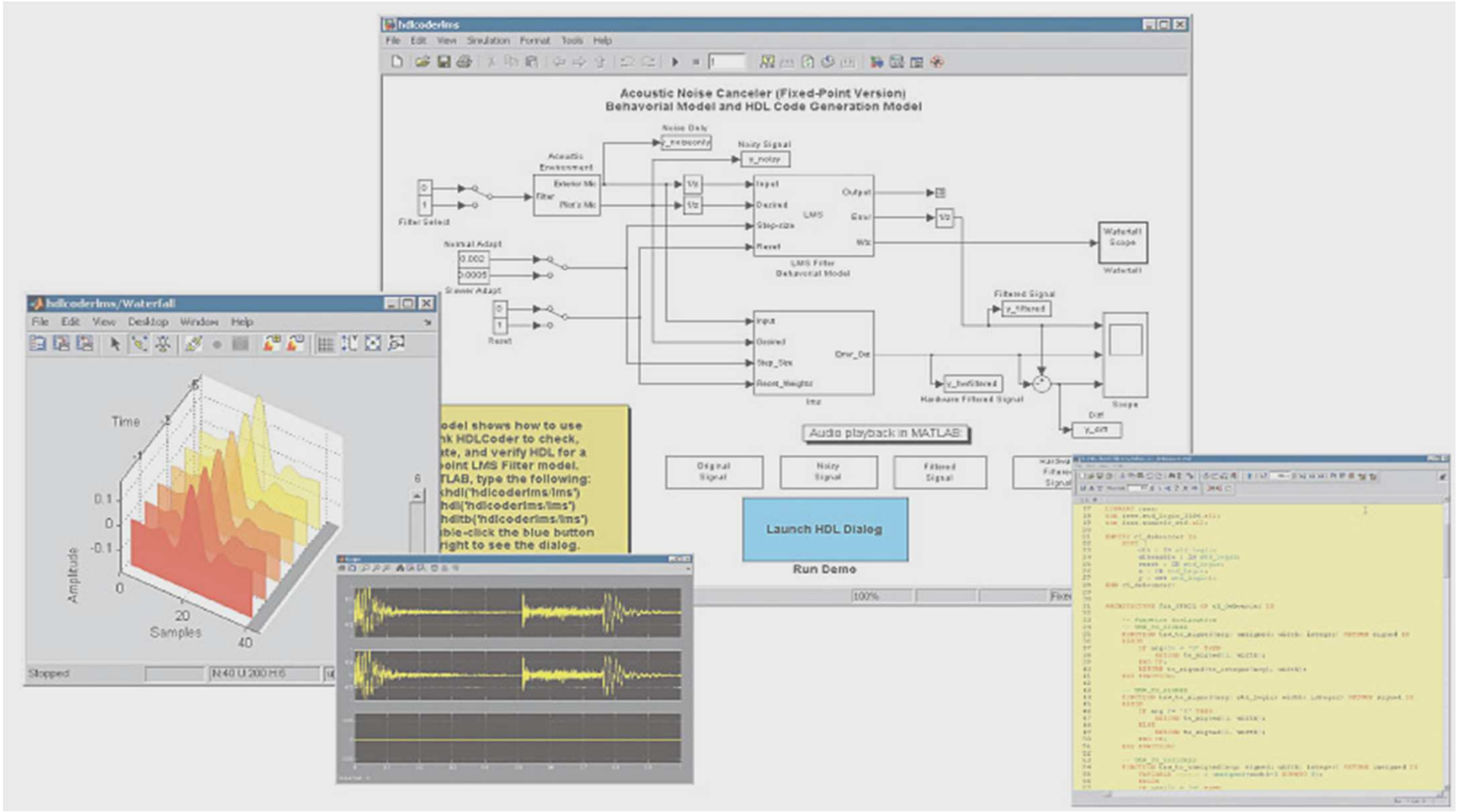
Logic Analyzer	Visualize, measure, and analyze transitions and states over time

Functions

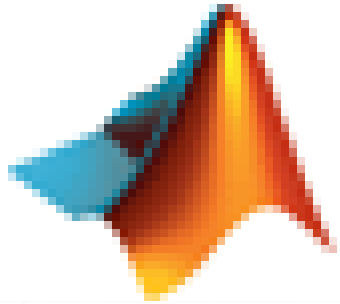
`generatehdl`	Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder)

Topics

Simulink® HDL Coder™



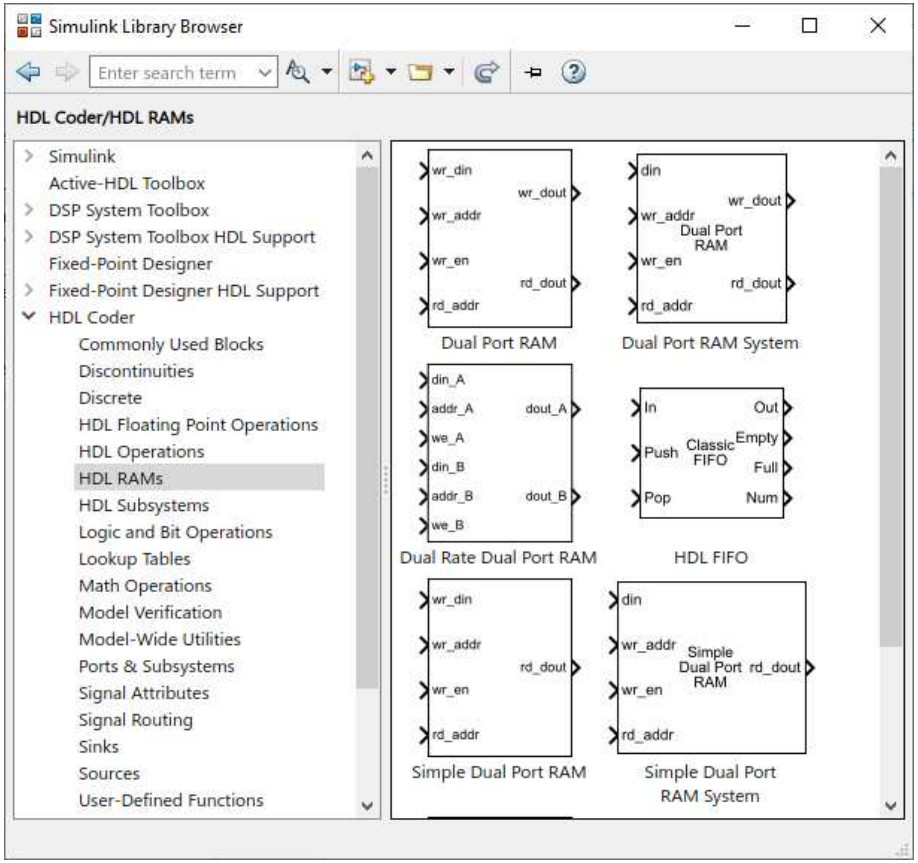
Simulink® HDL Coder™

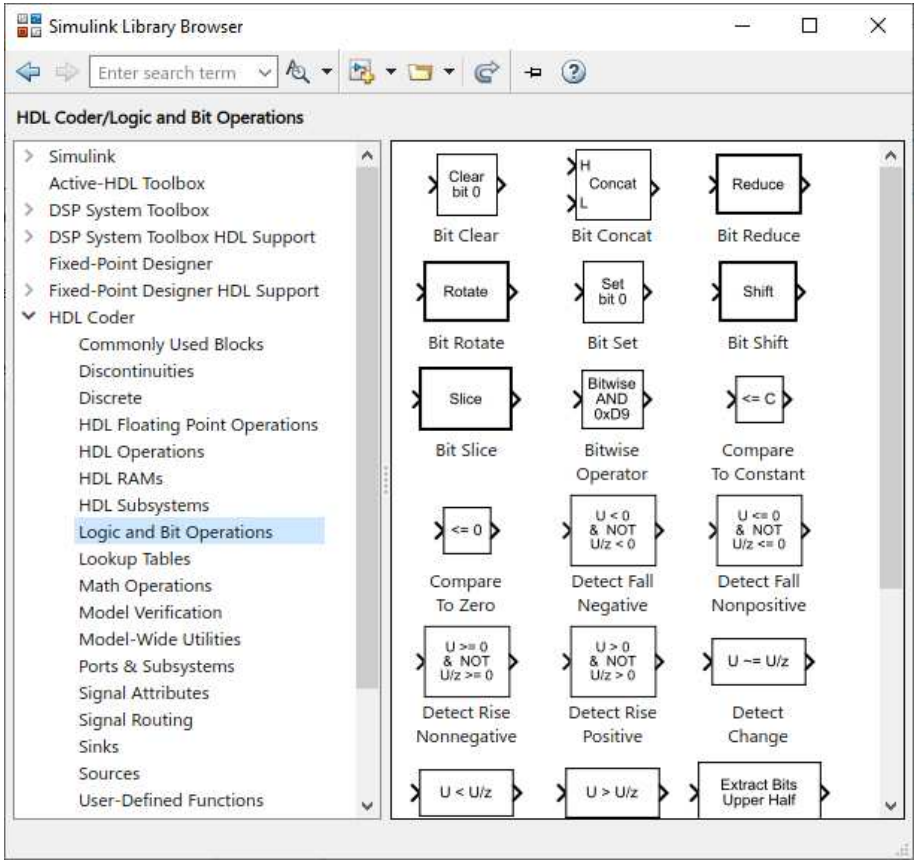


Key Features

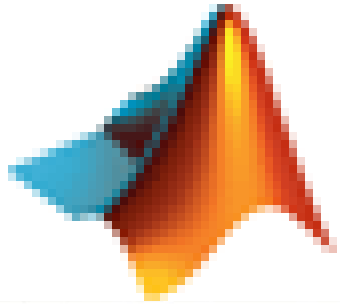
- *Generates synthesizable HDL code from Simulink models and Embedded MATLAB™ code for datapath implementations*
- *Generates synthesizable HDL code from Stateflow charts for Mealy and Moore finite-state machines and control logic implementations*
- *Generates VHDL code that is IEEE 1076 compliant and Verilog code that is IEEE 1364-2001 compliant*
- *Lets to create bit-true and cycle-accurate models that match your Simulink design specifications*
- *Lets to select from multiple HDL architectural implementations for commonly used blocks*
- *Lets you specify the subsystem for HDL code generation*
- *Enables you to reuse existing IP HDL code (with EDA Simulator Link products)*
- *Generates simulation and synthesis scripts*

Simulink® HDL Coder™





MATLAB Filter Design HDL Coder



- ***Support for the following discrete-time filter structures:***

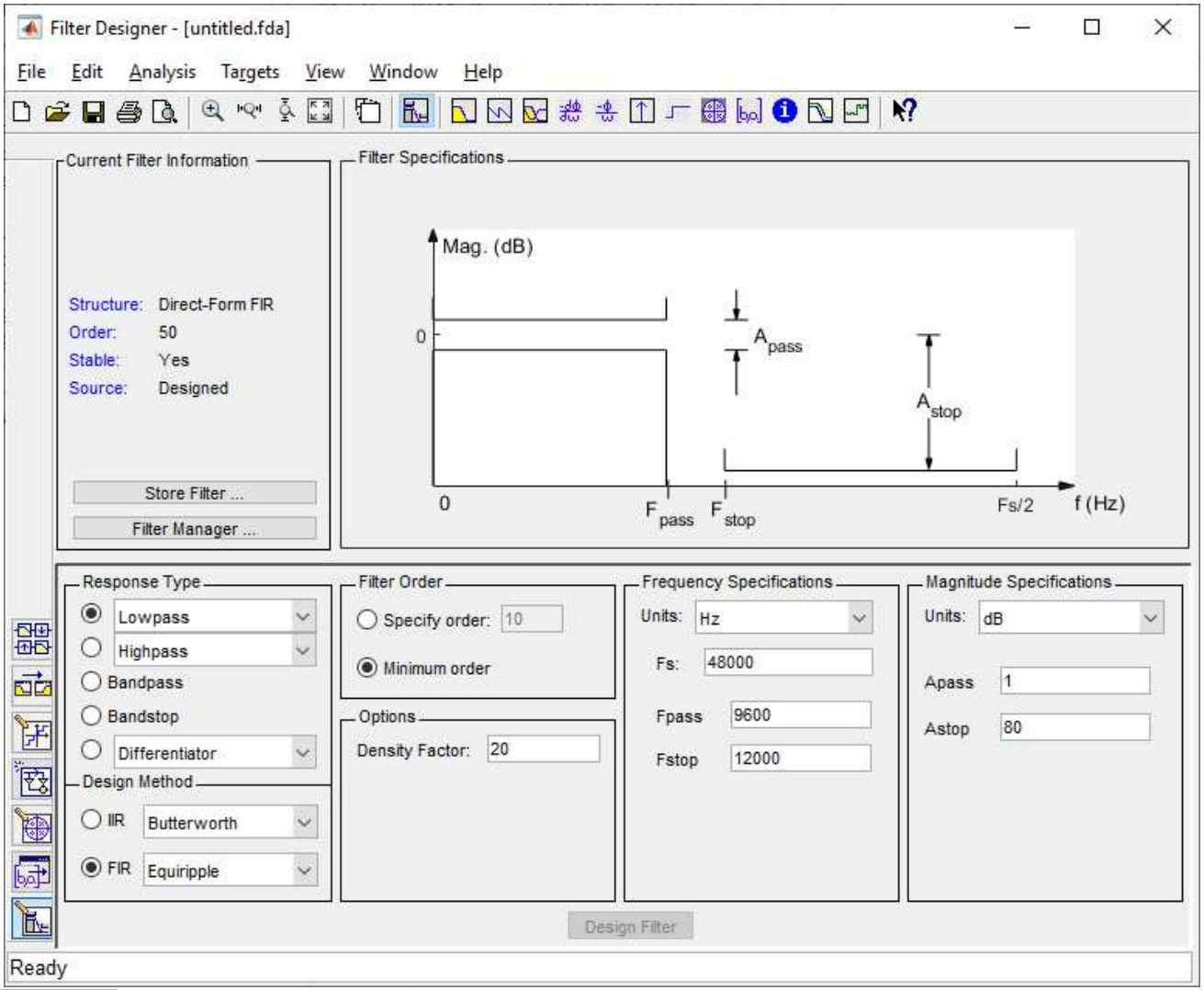
  Finite Impulse Response (FIR) – Antisymmetric FIR – Transposed FIR – Symmetric FIR Second-order section (SOS) – Infinite Impulse Response (IIR) Direct Form I – SOS IIR Direct Form I transposed – SOS IIR Direct Form II – SOS IIR Direct Form II transposed – Discrete-Time Scalar – Delay filter – Farrow (fractional delay) filter

- ***Support for the following multirate filter structures:***

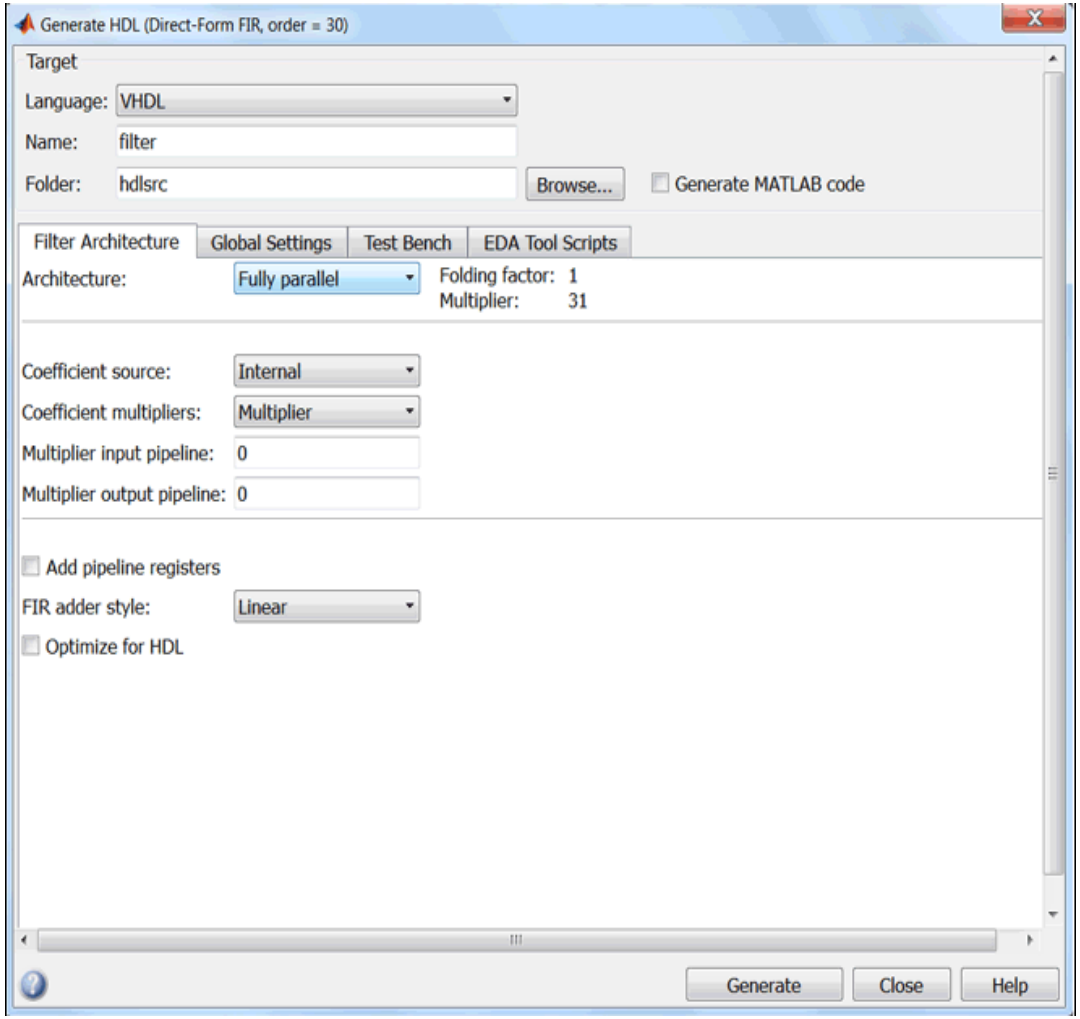
  Cascaded Integrator Comb (CIC) interpolation – Cascaded Integrator Comb (CIC) decimation – Direct-Form Transposed FIR Polyphase Decimator – Direct-Form FIR Polyphase Interpolator – Direct-Form FIR Polyphase Decimator – FIR Hold Interpolator – FIR Linear Interpolator – Direct-Form FIR Polyphase Sample Rate Converter

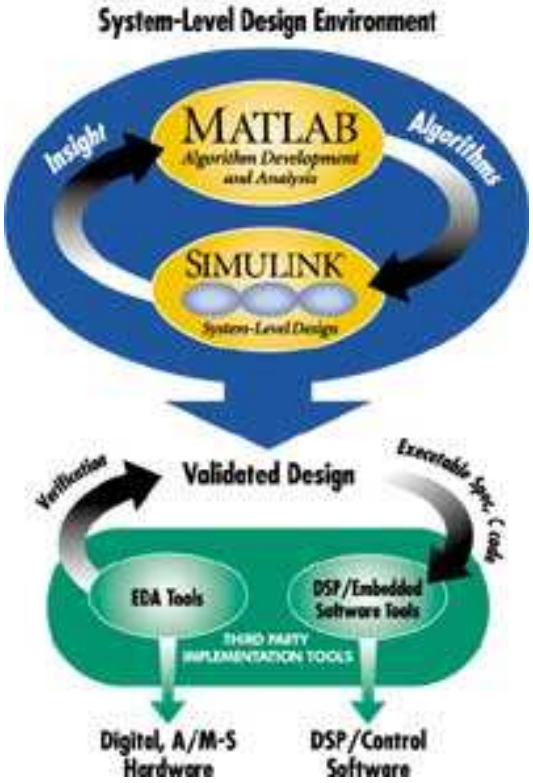
- ***Support for cascade filters (multirate and discrete-time)***
- ***Generation of code that adheres to a clean HDL coding style***
- ***Options for optimizing numeric results of generated HDL code***
- ***Options for specifying parallel, serial (fully, partly or cascade), or distributed arithmetic architectures for FIR filter realizations***
- ***Options for controlling the contents and style of the generated HDL code and test bench***
- ***Test bench generation for validating the generated HDL filter code***
- ***VHDL, Verilog, and ModelSim Tcl/Tk DO file test bench options***
- ***Automatic generation of scripts for third-party simulation and synthesis tool***

MATLAB Filter Design HDL Coder



MATLAB Filter Design HDL Coder





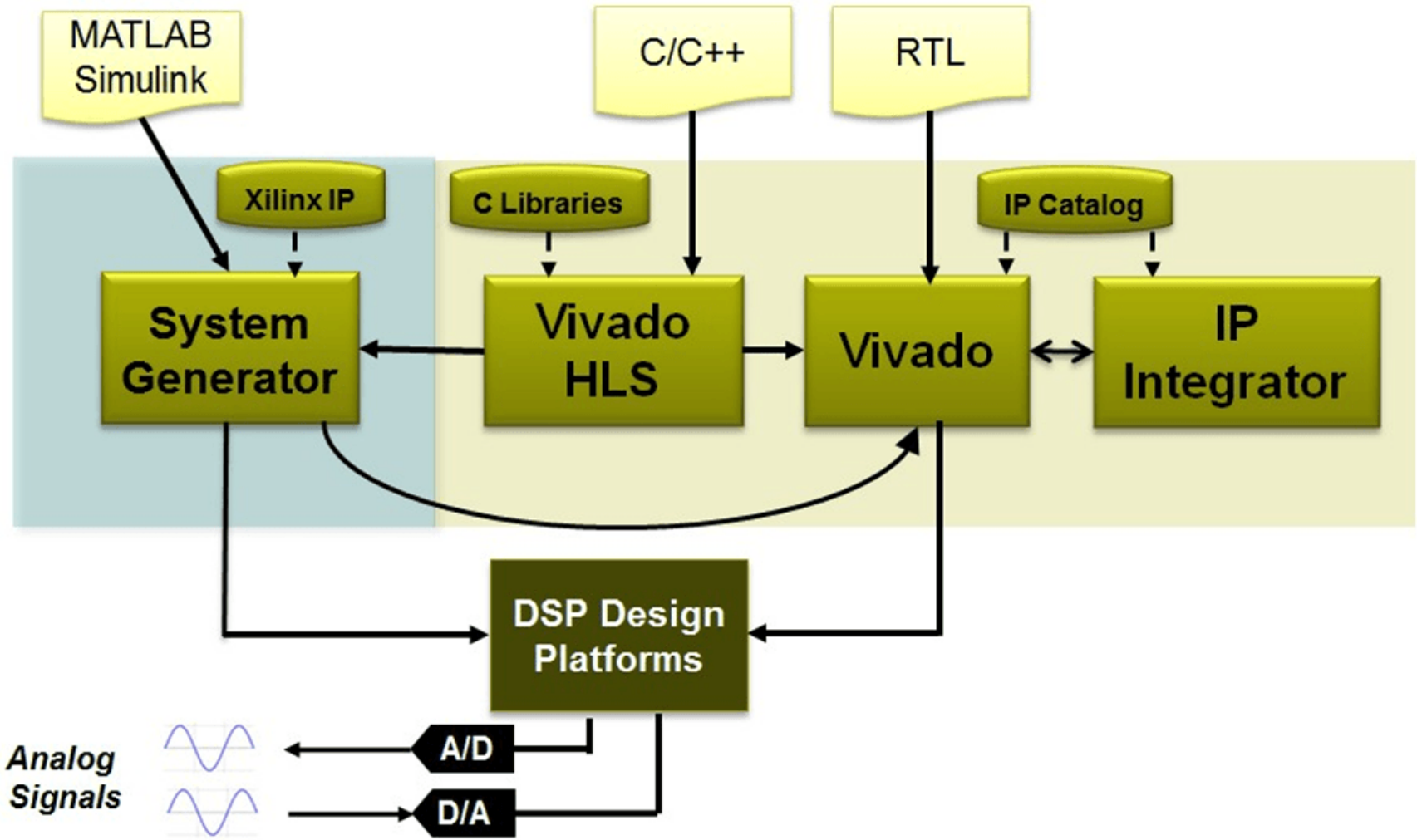
Graphical user interface (GUI) accessible from Filter Design and Analysis

Xilinx:
System Generator for DSP

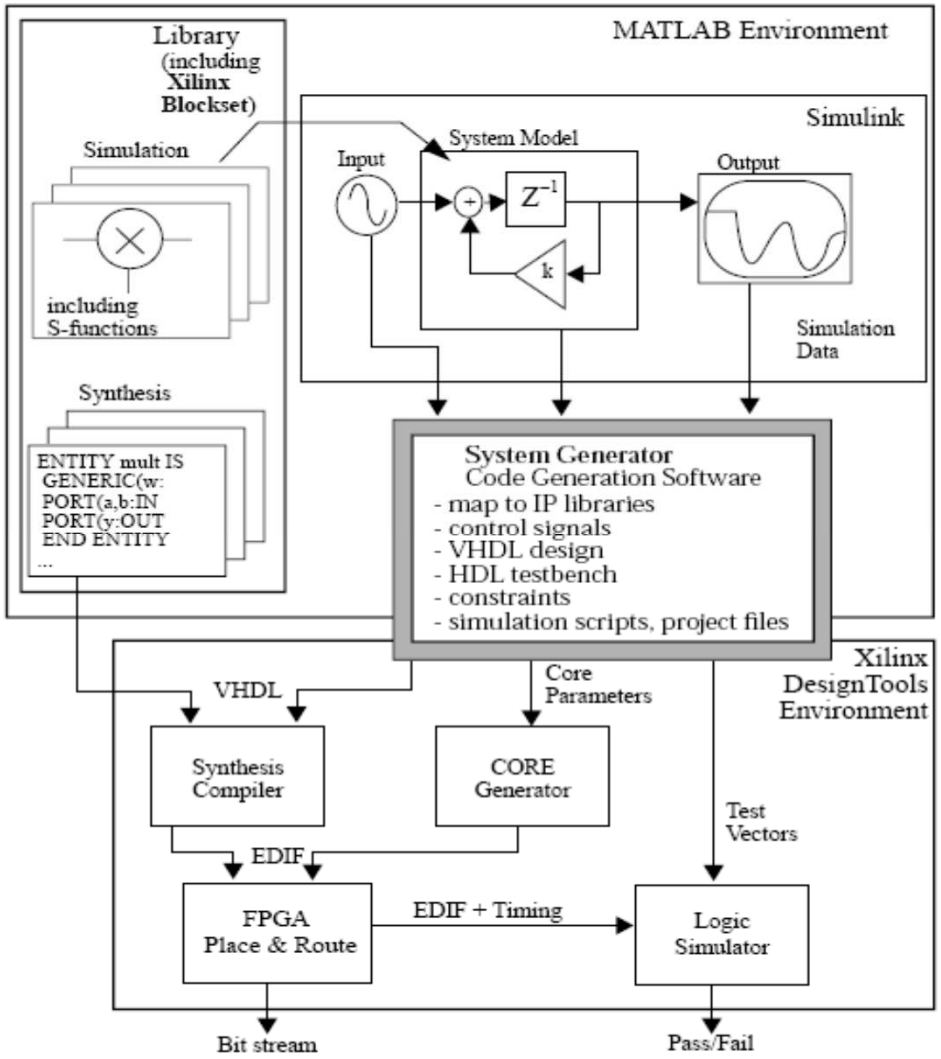
XILINX
SYSTEM
GENERATOR™
For DSP

System Generator provides a system integration platform for the design of DSP FPGAs that allows the RTL, Simulink, MATLAB and C/C++ components of a DSP system to come together in a single simulation and implementation environment.

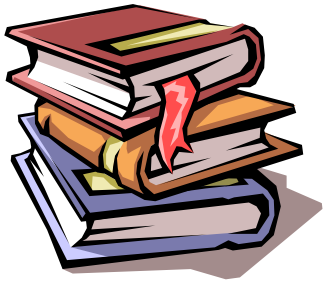
Implementacja algorytmów DSP – *Design Flow*



Xilinx:
System Generator for DSP



1. ***Describe the algorithm in mathematical terms***
2. ***Realize the algorithm in the design environment, initially using double precision***
3. ***Trim double precision arithmetic down to fixed point***
4. ***Translate the design into efficient hardware***



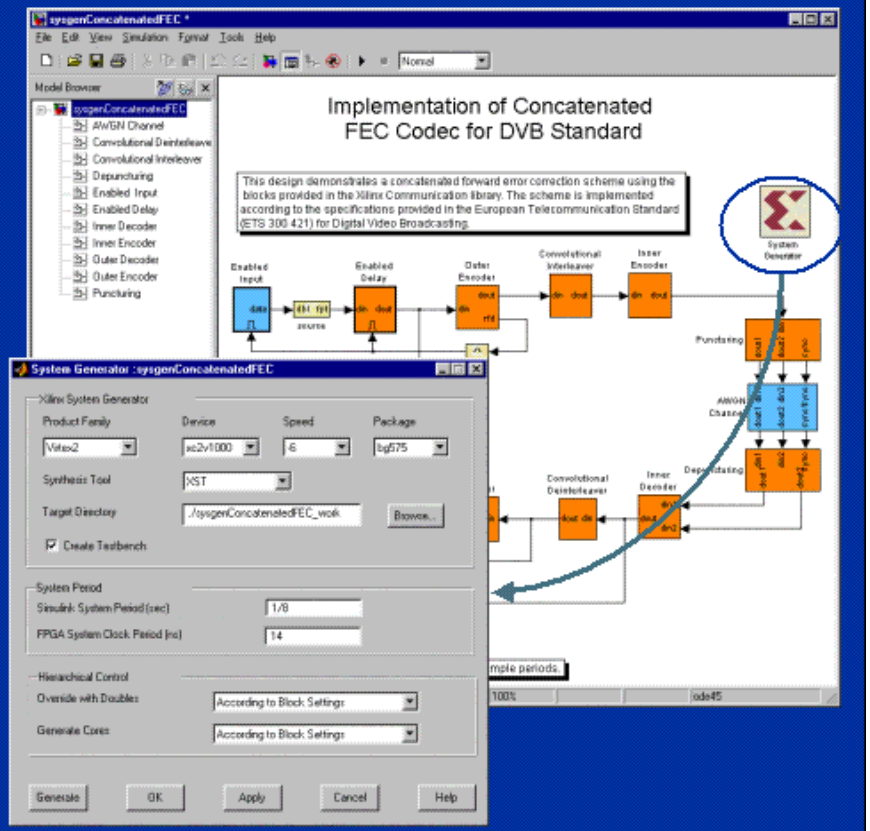
Xilinx:
System Generator for DSP

Introduction
Introduction to System Generator
Vivado Design Suite Tutorial: Model-Based DSP Design Using System Generator
Vivado Design Suite Tutorial: Designing with IP
Vivado Design Suite Tutorial: Creating and Packaging Custom IP
Key Concepts
Generating Vivado HLS block for use in System Generator for DSP
Using Vivado HLS C/C++/System C block in System Generator
Working with System Generator for DSP and Platform Design Flows from IP Integrator
Using Hardware Co-Simulation with Vivado System Generator for DSP
System Generator Multiple Clock Domains
Specifying AXI4-Lite Interfaces for your Vivado System Generator Design
MathWorks - FPGA Design and Codesign

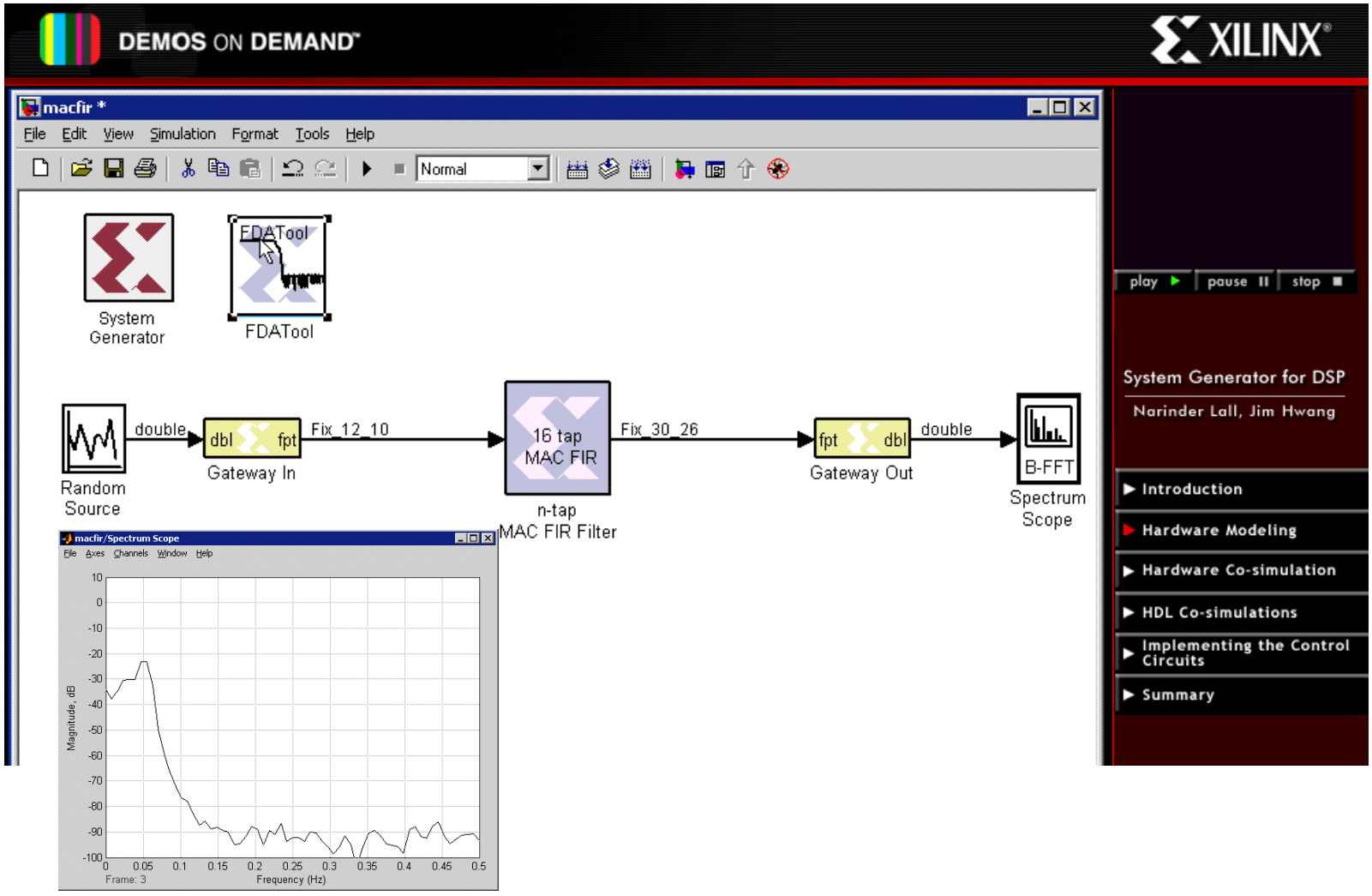
Xilinx:
System Generator for DSP

System Generator for DSP

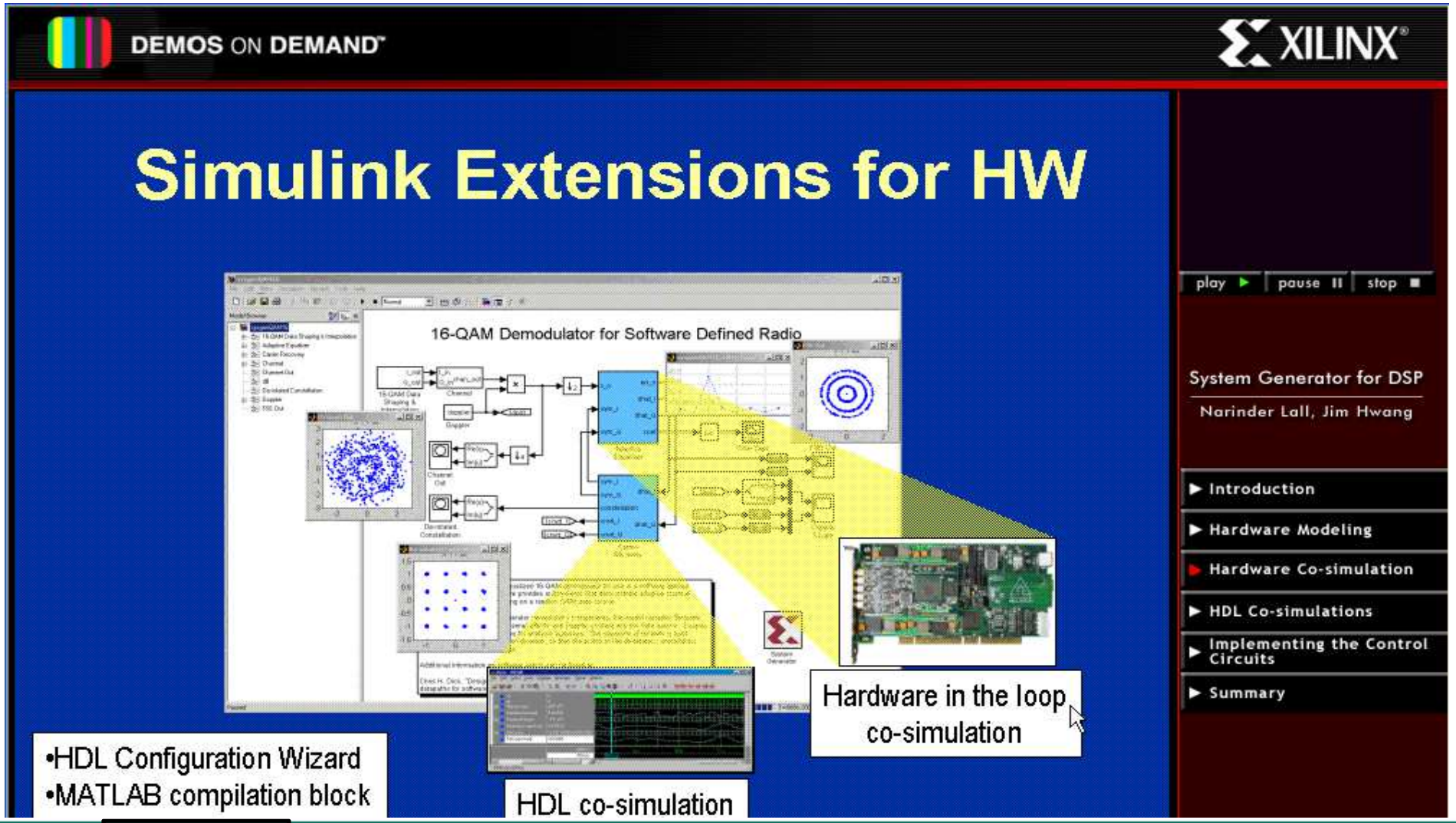
- Library-based, visual data flow
- Polymorphic operators
- Arbitrary precision fixed-point
- Bit and cycle true modeling
- Multi-rate signal processing
- Seamlessly integrated with Simulink and MATLAB
  - Type and rate propagation
  - Test bench and data analysis
- Automatic code generation
  - Synthesizable VHDL
  - IP cores
  - HDL test bench
  - Project and constraint files



Xilinx:
System Generator for DSP



Xilinx: Simulink



Interfejsy MATLAB

Recorded Webinar: HDL Functional Verification with MATLAB & Simulink

Contact Sales | Request a Trial | Request Training | Webinar Feedback | Exit Webinar

The MathWorks — MATLAB&SIMULINK

Old Pain

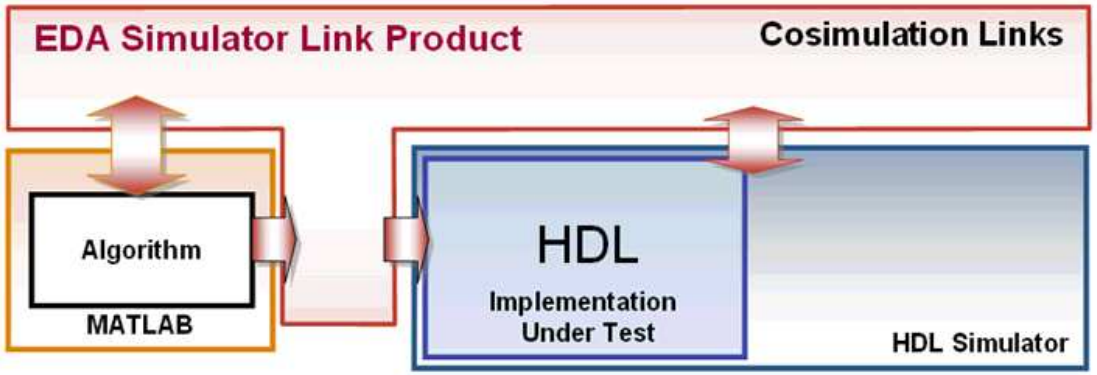
Disjointed development flow

- Manual
- Difficult to share knowledge
- Slow to debug
- Chronic extra effort required

New Solution

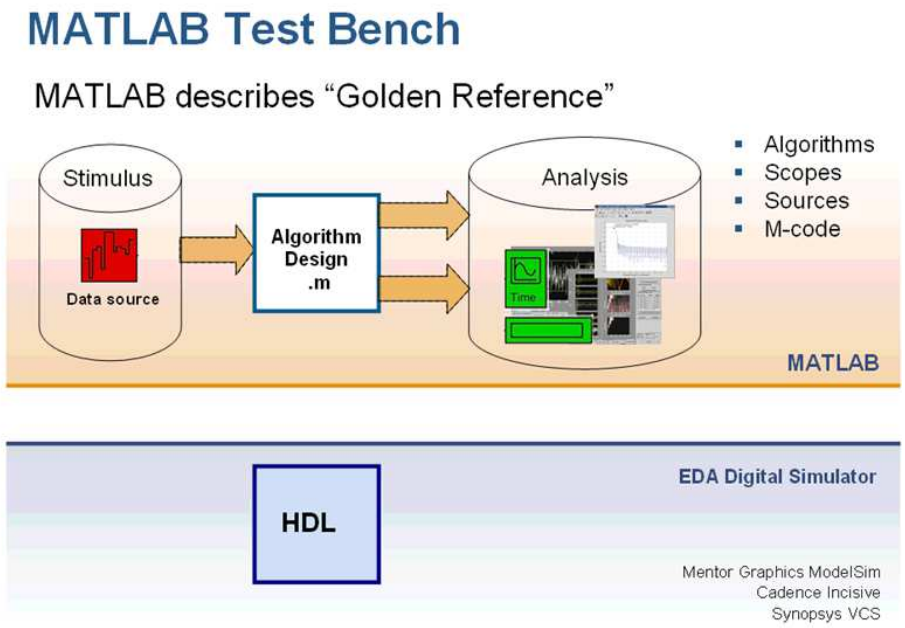
Integrated automated flow

- Leverages existing MATLAB
- Enables cross-team effort
- Interactive debug
- "Plugs into" existing flow

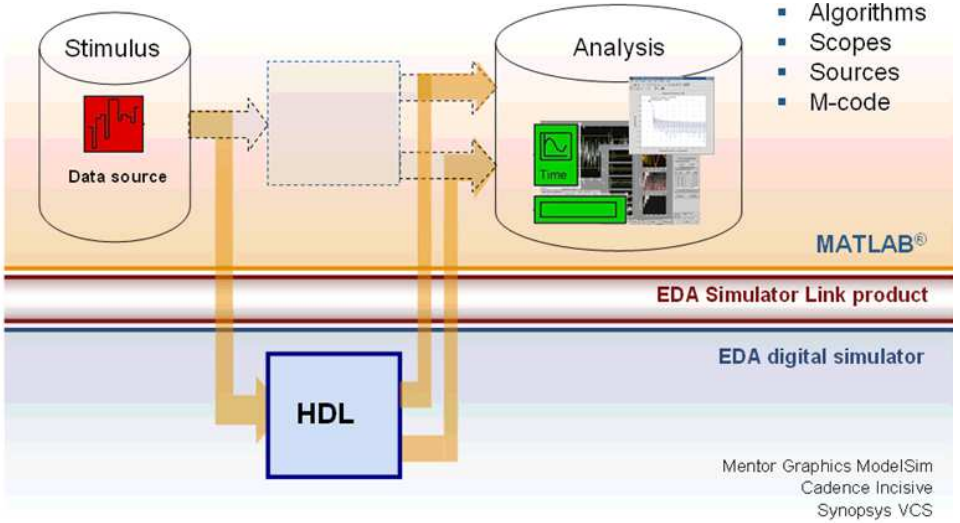


Interfejsy MATLAB

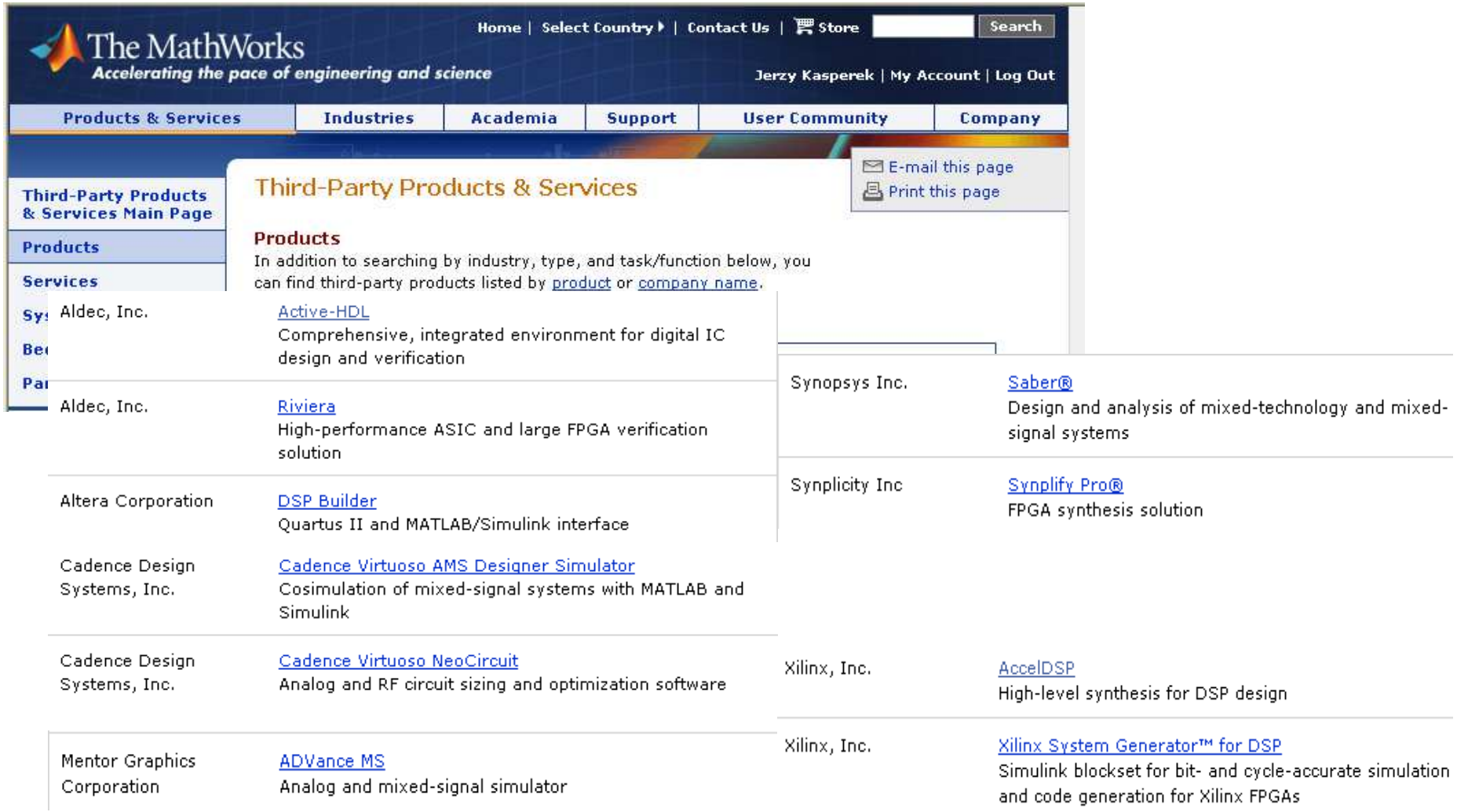
Symulacja funkcjonalna w środowisku MATLAB



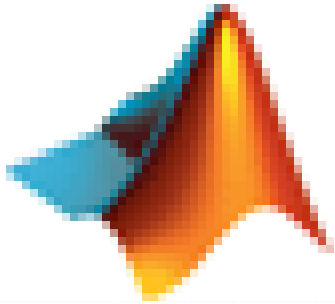
Symulacja w środowisku EDA

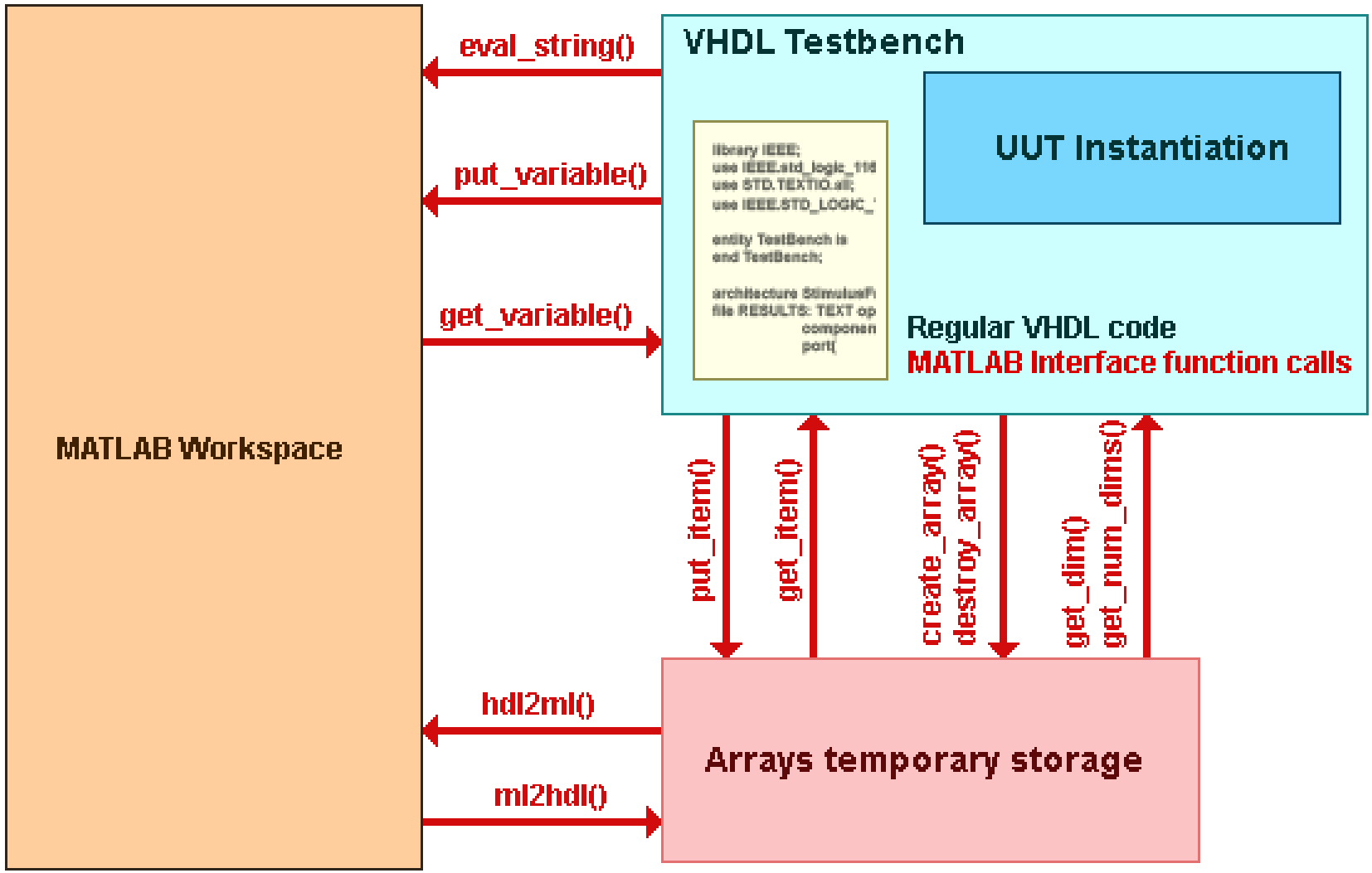


Interfejsy MATLAB

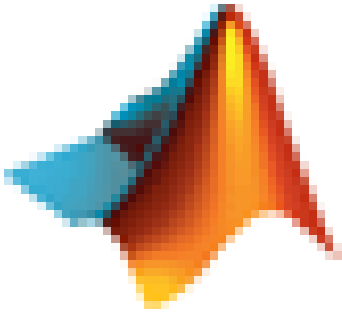


Interfejs AHDL ⇔ MATLAB



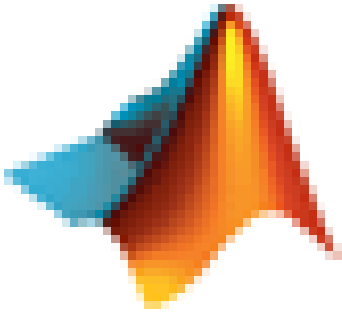


Interfejs AHDL ⇔ MATLAB



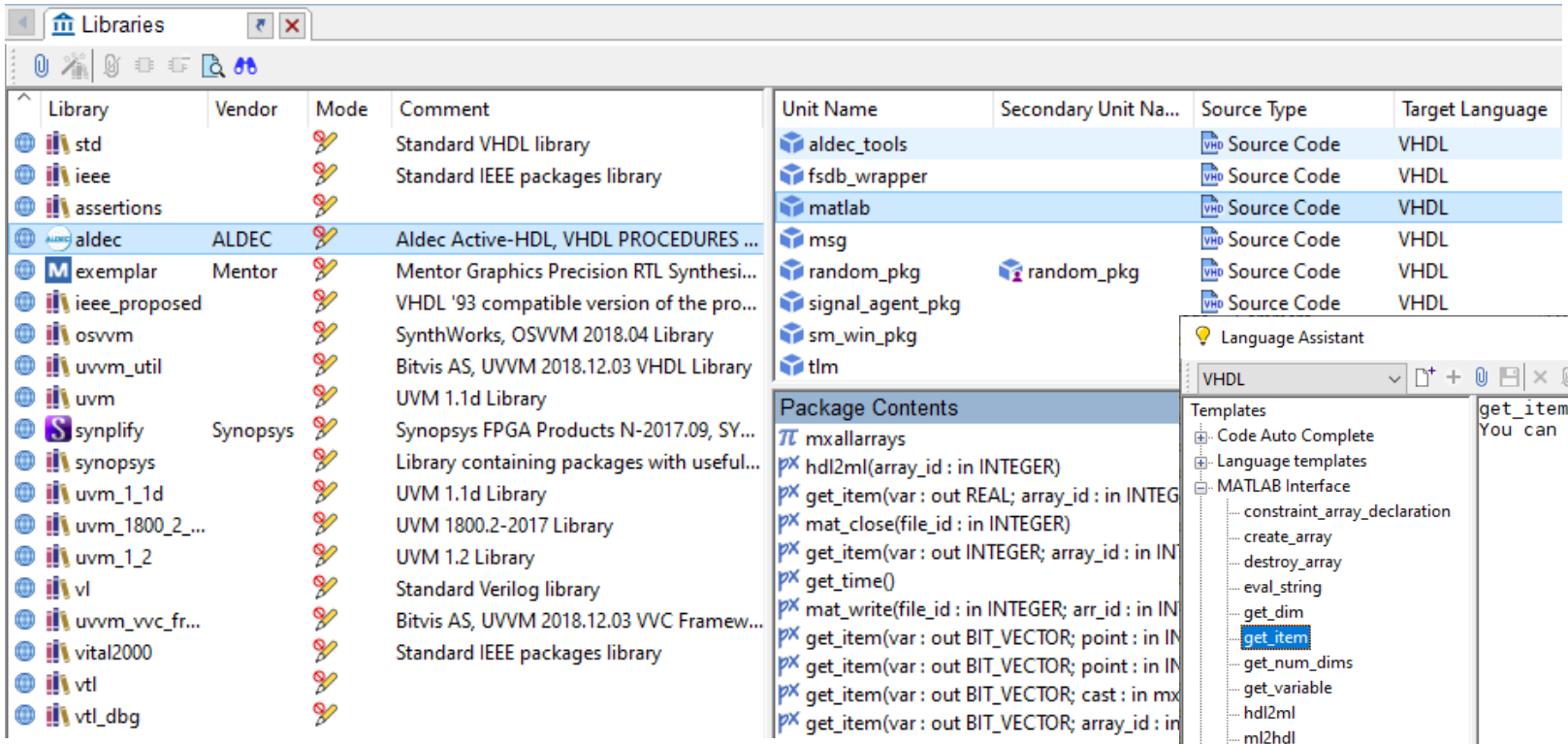
- *Comprehensive model verification, data analysis, and results visualization*
- *Ability to compare theoretical algorithm with hardware implementation*
- *Effective use of mathematical formulas that may be difficult to implement with pure HDL*
- *Simple and efficient way of integration of HDL hardware models with the high-level model of the system in MATLAB*
- *Capability of generating sophisticated test vectors*
- *Execution of numerical computations that are not performed by designed hardware*
- *Extending the testbench with parts developed in MATLAB for faster algorithm execution and/or data visualization,*
- *An effective bidirectional data transfer between HDL simulator and MATLAB environment (millions of samples can be passed at ease).*
- *Ability to request any service from MATLAB directly from the HDL code, including sophisticated calculations and data visualization performer by the MATLAB graphical tools.*

Interfejs AHDL ⇔ MATLAB



- **Software Requirements**
  MATLAB R14 or higher
- **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 (System)Verilog Types**
  – Logic
  – Integer
  – Real

Interfejs AHDL ⇔ MATLAB



W projekcie wykorzystującym interfejs MATLAB należy zadeklarować:

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

Interfejs AHDL ⇔ MATLAB

eval_string

Description

The eval_string routine allows you to pass commands from the HDL Simulator (Active-HDL) to the MATLAB environment. MATLAB output is printed to the Console window (not to the MATLAB Command Window).

NOTE: Placing a semicolon after a MATLAB command disables the echo.
This helps to limit excessive console output that could impair simulation performance.

Syntax

VHDL:

```
eval_string("ml_cmd");
```

Verilog:

```
$eval_string("ml_cmd");
```

ml_cmd

The command or expression to be sent to the MATLAB environment.
It is equivalent to typing a string in the MATLAB Console or the MATLAB Command Window.
Supported argument types (VHDL): string
Supported argument types (Verilog): string
(i.e. a sequence of characters enclosed by double quotes).

Note that string variables are not supported.

Interfejs AHDL ⇔ MATLAB

Passing Scalar Values put_variable

Description

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).

Syntax

For an HDL variable (hdl_var) expressed as a scalar, a vector treated as an integer, an integer or a floating point number:

VHDL:

```
put_variable("var_name", var);
```

For an HDL variable (hdl_var) expressed as a vector treated as fixed-point number:

VHDL:

```
put_variable("var_name", var, point);
```

Interfejs AHDL ⇔ MATLAB

Passing Scalar Values put_variable

```
put_variable("var_name", var);
put_variable("var_name", var, point);
```

var_name

The name of the variable in the MATLAB environment. If the variable already exists, its value will be overwritten. If variable does not exist, it will be created and assigned.
Supported argument types (VHDL, Verilog): string

var

The HDL variable to be transferred to the MATLAB environment. The variable is read-only. The ***put_variable*** *routine call will not change its value.*
Supported argument types (VHDL): std_logic, std_logic_vector, signed, unsigned, bit, bit_vector, integer, real
Supported argument types (Verilog): reg (scalar or vector), net (scalar or vector), integer, real

point

The location of the binary point, starting from the least significant position. If set to 0, the number shall be treated as an integer value.

Interfejs AHDL ⇔ MATLAB

Passing Scalar Values put_variable

```
package Matlab is
type TDims is array(POSITIVE range <>) of integer;

------------------------------------------------------------------
-- procedure declaration
------------------------------------------------------------------
procedure put_variable(var_name: in string; var: in std_logic);
attribute foreign of put_variable: procedure is
        "VHPI $ALDEC/BIN/aldec_matlab_cosim.dll; put_variable_s";
...

------------------------------------------------------------------
-- procedure body
------------------------------------------------------------------
procedure put_variable(var_name: in string; var: in std_logic) is
begin
end procedure;


      -- VHPI – VHDL Programming Language Interface
```

Interfejs AHDL ⇔ MATLAB

Passing Scalar Values get_variable

Description

The get_variable routine allows you to pass 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.

Syntax

For an HDL variable (hdl_var) expressed as a scalar, a vector treated as an integer, an integer or a floating point number:

VHDL:

```
get_variable("var_name", var);
```

Verilog:

```
$get_variable("var_name", var);
```

For an HDL variable (hdl_var) expressed as a vector treated as fixed-point number:

VHDL:

```
get_variable("var_name", var, point);
```

Verilog:

```
$get_variable("var_name", var, point);
```

Interfejs AHDL ⇔ MATLAB

Passing Scalar Values get_variable

```
get_variable("var_name", var);
get_variable("var_name", var, point);
```

var_name

The name of a variable in the MATLAB environment. Floating point and integer types are supported. If no variable is defined by the specified name in MATLAB, an error message will be printed to the Console window. Supported argument types (VHDL, Verilog): string

var

The name of the target HDL variable where the value should be stored.
Supported argument types (VHDL): std_logic, std_logic_vector, signed, unsigned, bit, bit_vector, integer, real
Supported argument types (Verilog): reg *(scalar or vector),* integer, real

point

The location of the binary point, starting from the least significant position. If set to 0, the number shall be treated as an integer value.

Interfejs AHDL ⇔ MATLAB

Manipulating Array Values create_array

Description

Creates an array with the specified dimensions and returns the array identifier (array_id). If the array creation fails, 0 is returned. The number of array dimensions is currently limited to 6. It is recommended to remove the array from the memory with the destroy_array *routine if it is no longer needed for calculations.*

Syntax

VHDL:

```
array_id = create_array("name", ndim, dims);
```

Verilog:

```
array_id = $create_array("name", dim_0, ... , dim_5);
```

name

The name of the variable that will be used in the MATLAB environment.
Supported argument types (VHDL, Verilog): string
(Note that string variables are not supported)

ndim

The number of dimensions. Supported argument types (VHDL): integer

dims

An array of integers specifying the number of elements for each dimension.

Interfejs AHDL ⇔ MATLAB

Manipulating Array Values destroy_array

Description

Removes an array from the memory.

Syntax

VHDL:

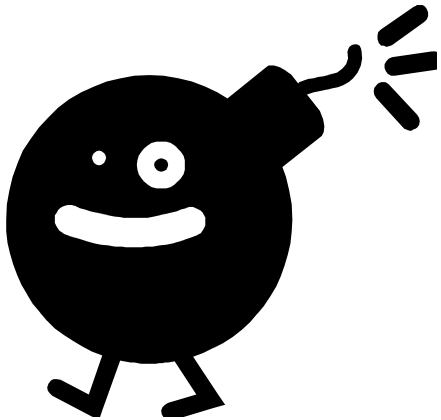
```
destroy_array(array_id);
```

Verilog:

```
$destroy_array(array_id);
```

array_id

The array identifier.



Interfejs AHDL ⇔ MATLAB

Manipulating Array Values get_item / put_item

***Description* get_item**

Reads a floating-point number from the array and stores it in an HDL variable.

***Description* put_item**

Changes an HDL value to a floating-point number and stores it in the specified element of the array.

Syntax

VHDL:

```
get_item(var, array_id, index);
put_item(var, array_id, index);
```

Verilog:

```
$get_item(var, point, array_id, sel_0, ..., sel_5);
$put_item(var, point, array_id, sel_0, ..., sel_5);
```

var - *The name of an HDL variable to be loaded from / stored in the selected array element.*

array_id - *Array identifier.*

index - *An array of integers specifying the location of the element in the array. Array indexing starts with 1.*

Interfejs AHDL ⇔ MATLAB

Manipulating Array Values ml2hdl / hdl2ml

***Description* ml2hdl**

Transfers an array specified by the name parameter from the MATLAB environment to the HDL simulator and returns the identifier that points to the array stored in the HDL simulator.

***Description* hdl2ml**

Transfers an array specified by the array identifier to the MATLAB environment and stores it under the name previously specified with the create_array *or* ml2hdl *routine.*

Syntax

VHDL:

```
array_id := ml2hdl("name", array_id);
hdl2ml(array_id);
```

Verilog:

```
array_id = $ml2hdl("name", array_id);
$hdl2ml(array_id);
```

name - *The name of a variable in the MATLAB environment. Supported argument types (VHDL, Verilog):* string

array_id - *Array identifier (obtained with create_array or the previous call of the ml2hdl function). Supported argument types (VHDL, Verilog):* integer

Interfejs AHDL ⇔ MATLAB

Manipulating Array Values get_num_dims, get_dim

***Description* get_num_dims**

Reads the number of dimensions in the selected array. The returned value is an integer.

***Description* get_dim**

Returns the number of elements in the specified dimension.

Syntax

VHDL:

```
ndims := get_num_dims(array_id);
elem := get_dim(array_id, dim_sel);
```

Verilog:

```
ndims = $get_num_dims(array_id);
elem = $get_dim(array_id, dim_sel);
```

array_id - *Array identifier (obtained with create_array or the previous call of the ml2hdl function). Supported argument types (VHDL, Verilog):* integer

dim_sel - *The number of array dimensions. Dimensions are numbered starting with 1. Supported argument types (VHDL, Verilog):* integer

Interfejs AHDL ⇔ MATLAB
przekazywanie bieżącego czasu symulacji put_simtime

Description **put_simtime**
Transfers simulation time to a MATLAB variable.

Syntax
VHDL:
```
put_simtime (var_name);
put_simtime (var_name_t, var_name_u);
put_simtime (var_name_t, var_name_u, class);
```
Verilog:
```
$put_simtime (var_name);
$put_simtime (var_name_t, var_name_u);
$put_simtime (var_name_t, var_name_u, class);
```

var_name
The name of a MATLAB variable where the current simulation time will be stored. The time value will be expressed in seconds.
Supported argument types (VHDL): string *(Verilog):* string literal
var_name_t
The name of a MATLAB variable where the current simulation time will be stored. The time value will be expressed in simulation time units.
Supported argument types (VHDL): string *(Verilog):* string literal

Interfejs AHDL ⇔ MATLAB
przekazywanie bieżącego czasu symulacji put_simtime

Przykład

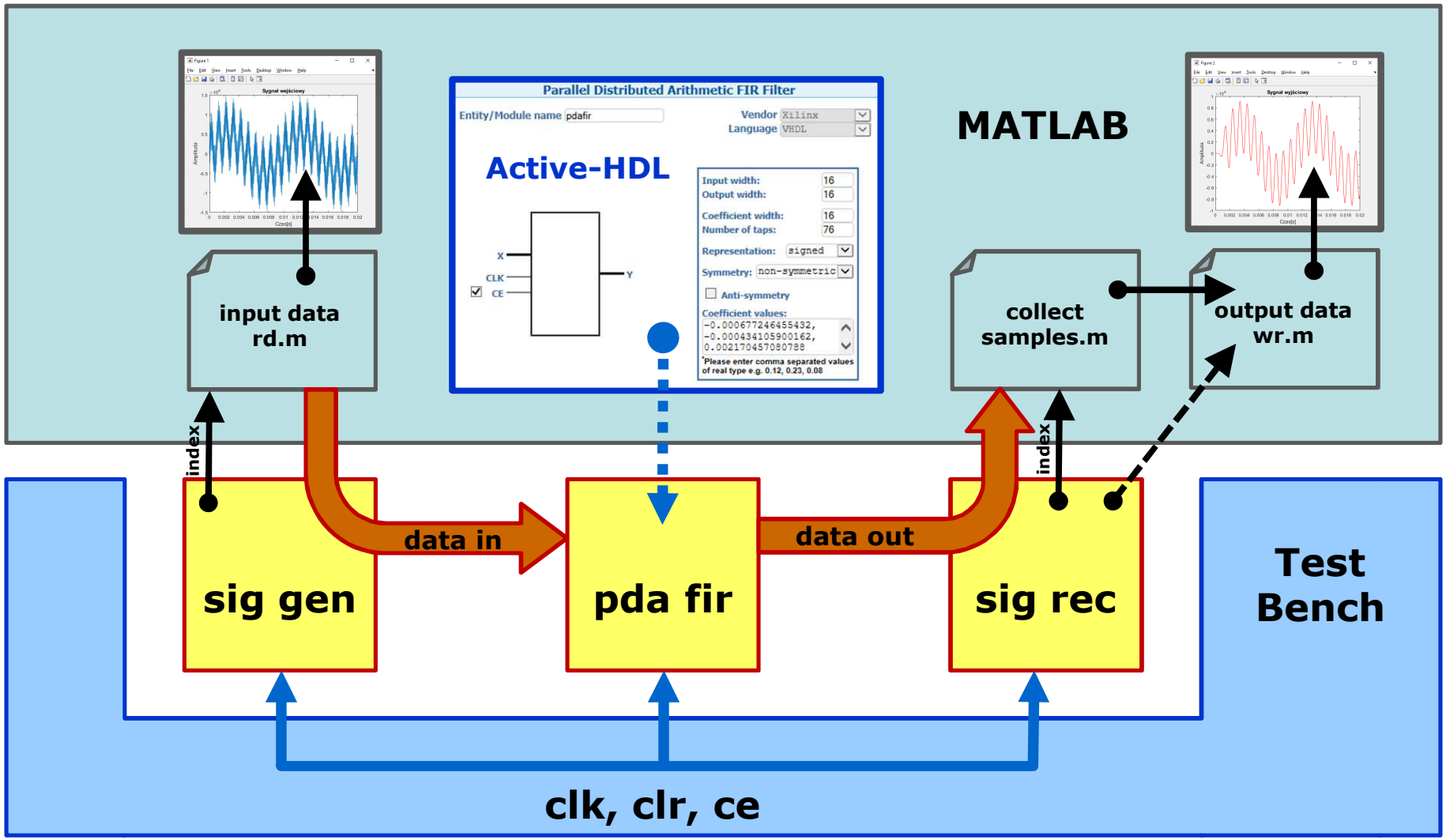
Gdzieś w kodzie VHDL:

```
put_simtime ("HDL_sim_time", "ps");
eval_string (matlab_data_script);
get_variable ("output_sample", data_sample);
```

W skrypcie matlab_data_script

```
% send back to AHDL output sample - time is send in picoseconds
% values -1.0....1.0 are converted to U2 signed 14 bits
amplitude_scaling_factor = 2^13;
simulation_time_unit = 1e-12;
% get sample
how_many_samples_V = size(V_rx,2);
how_many_time_units_V = param_rx_duration/simulation_time_unit;
sample_index_V = round(HDL_sim_time*(how_many_samples_V/how_many_time_units_V));
if (sample_index_V < how_many_samples_V && sample_index_V > 0)
    output_sample = amplitude_scaling_factor * V_rx(sample_index_V);
else
    output_sample = 0;
    disp('ADC_V data index out of range');
end
```

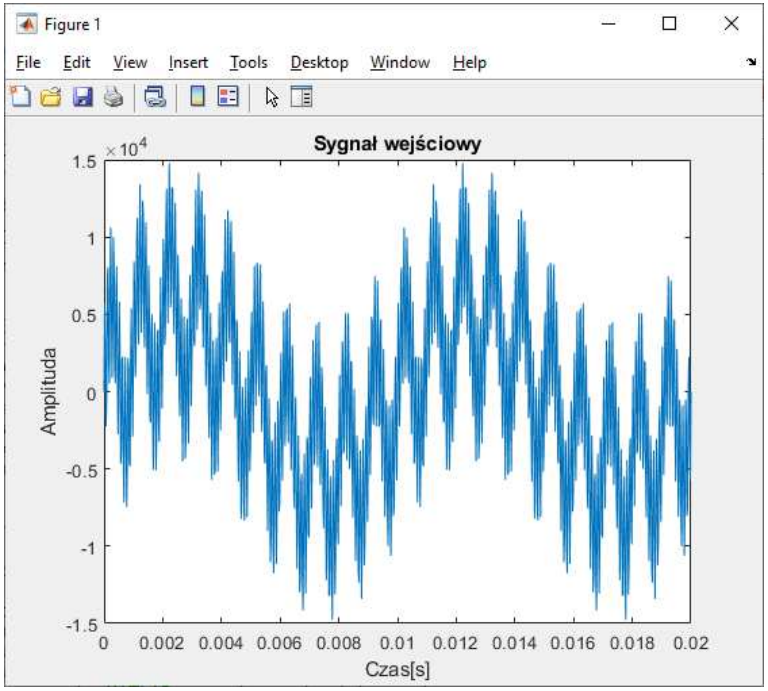
Projekt filtru AHDL ⇔ MATLAB



Projekt filtru AHDL ⇔ MATLAB

Plik **input_data_rd.m**

```
fp = 44100;                          % częstotliwość próbkowania
t = (0:1/fp:0.02);                   % wyznaczenie czasów próbek od 0 do 20 msek
s1 = sin(2*pi*t*100);                % generacja trzech sinusoid ...
s2 = sin(2*pi*t*1000);               %
s3 = sin(2*pi*t*10000);              %
in_data_matrix = 50000*(s1+s2+s3);   % ...i ich połączenie dodając amplitudy

% plot
figure(1)
plot(t, in_data_matrix);
xlabel('Czas[s]');
ylabel('Amplituda');
title('Sygnał wejściowy');
```



Projekt filtru AHDL ⇔ MATLAB
PUT_samples: process

```
PUT_samples: process
  variable init_MATLAB: boolean:=false;
  variable array_ID: integer;                          -- identyfikator tablicy
  variable samples: integer;                           -- rozmiar tablicy
  variable pointer: TDims (1 to 2) := (1,1);           -- wskaźnik elementu tablicy
  variable sample: std_logic_vector(15 downto 0); -- próbka

begin
  -- MATLAB init, read how many samples in data input file
  -- run this part of the code just once...
  if init_MATLAB = false then
    init_MATLAB := true;
    -- execute MATLAB script
    eval_string("input_data_rd");              -- wykonanie "input_data_rd.m"
    array_ID := ml2hdl("in_data_matrix");      -- pobranie tablicy
    samples := get_dim(array_ID, 2);           -- pobranie rozmiaru tablicy
  end if;
```

Projekt filtru AHDL ⇔ MATLAB
PUT_samples: process

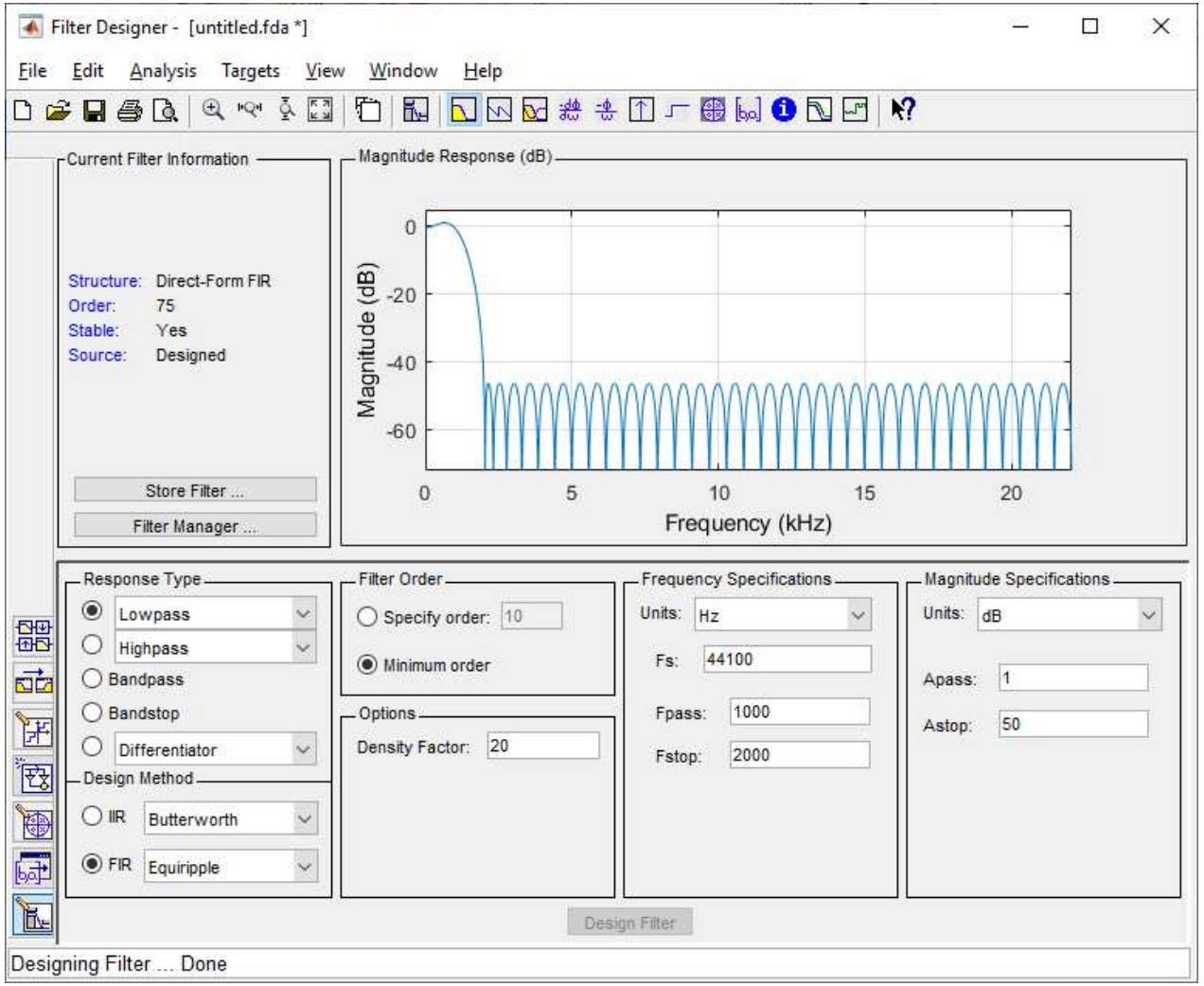
```
  -- if no reset get data
  if rst = '1' then
    sample  := (others => '0');
    pointer := (1,1);
  else
    get_item(sample, array_ID, pointer);  -- pobranie próbki z tablicy
    pointer(2):= pointer(2)+1;            -- zwiększenie wskaźnika
    data_in <= sample;                    -- wystawienie próbki na magistralę
  end if;

  -- data end?
  if (pointer(2) > samples and init_MATLAB = true) then
    report "End input data";
    destroy_array(array_ID);
    -- process end...
    wait;
  end if;

  data_in <= sample;
  -- run on every rising clock edge
  wait until clk'event and clk = '1';
end process;
```

Projekt filtru AHDL ⇔ MATLAB

Przykładowy projekt studencki:

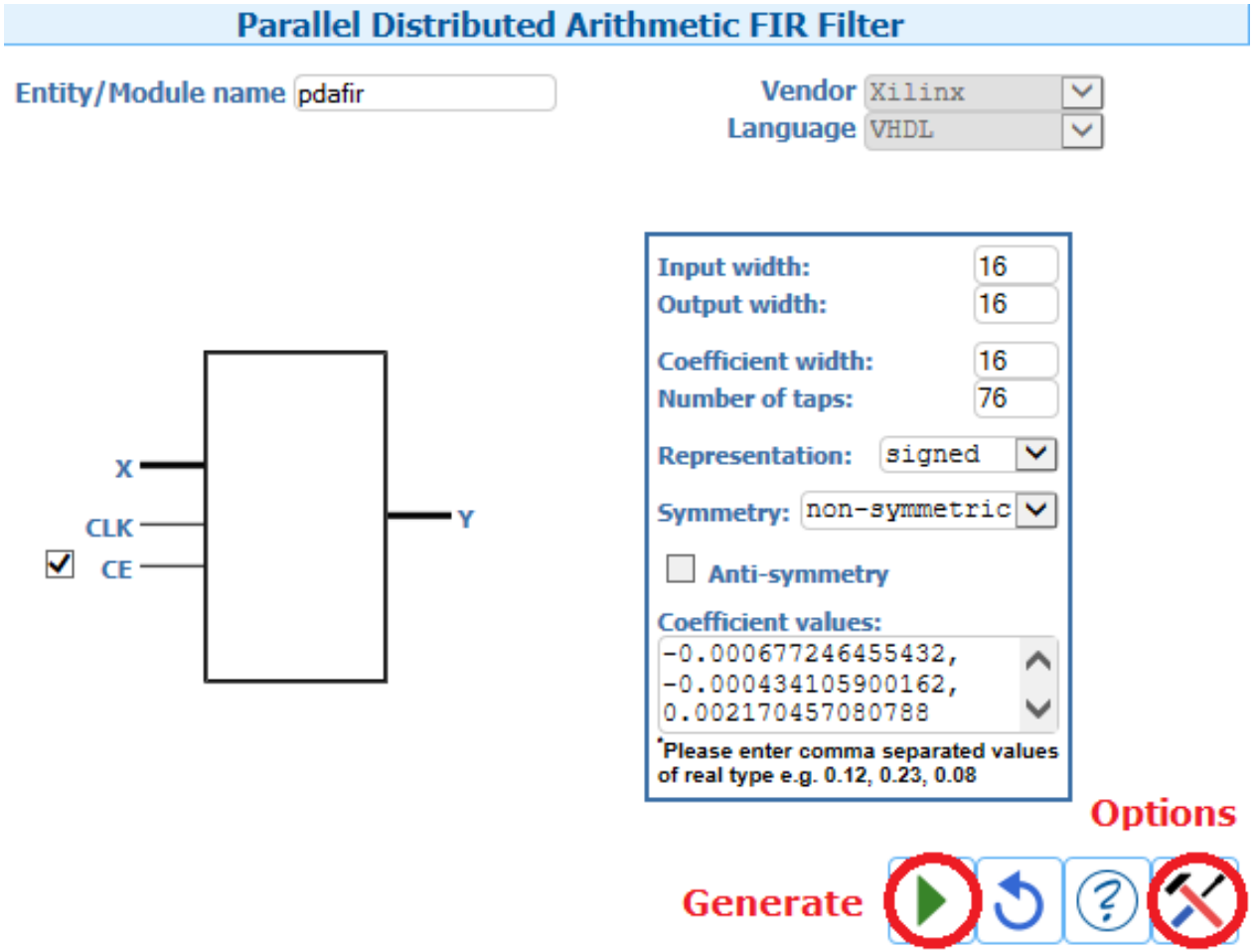
1. W środowisku MATLAB przy wykorzystaniu narzędzia **filterDesigner** zaprojektować filtr audio

2. Ze środowiska MATLAB wziąć współczynniki zaprojektowanego filtru



Projekt filtru AHDL ⇔ MATLAB

3. W środowisku AHDL przy wykorzystaniu narzędzia **IP Core Generator** zaprojektować filtr
4. Wpisać współczynniki pobrane ze środowiska MATLAB
5. W środowisku AHDL wygenerować syntezowalny plik źródłowy *.vhd z komponentem filtru
6. W środowisku AHDL wygenerować Testbench



Projekt filtru AHDL ⇔ MATLAB
GET_samples: process

```
  signal sample: std_logic_vector(15 downto 0); -- próbka

begin
  GET_samples: process
    variable init_MATLAB: boolean := false;
    variable array_ID: integer;                  -- identyfikator tablicy
    variable samples: integer;                   -- rozmiar tablicy
    variable counter: integer := 1;              -- licznik próbek
  begin
    -- MATLAB init, read how many samples in data input file
    -- run this part of the code just once...
    -- of course can be also passed via testbench
    if init_MATLAB = false then
      init_MATLAB := true;
      -- execute MATLAB script
      eval_string("input_data_rd");              -- wykonanie "input_data_rd.m"
      array_ID := ml2hdl("in_data_matrix");      -- pobranie tablicy
      samples := get_dim(array_ID, 2);           -- pobranie rozmiaru tablicy
    end if;
```

Projekt filtru AHDL ⇔ MATLAB
GET_samples: process

```
  -- if no reset put samples
  if (rst = '1' or init_MATLAB = false) then
    data_counter := 1;
  else
    if counter < (samples + 1) then
      sample <= data_out;
      put_variable("i", counter);
      put_variable("signal_sample", sample, 0);
      -- execute MATLAB script simple data capture
      eval_string("collect_samples");
      counter := counter + 1;
    else
       -- execute MATLAB script
       eval_string("output_data_wr");    -- wykonanie „output_data_wr.m"
       -- process end...
       wait;
     end if;
   end if;
   -- run on every rising clock edge
   wait until clk'event and clk = '1';
end process;
```

Plik **collect_samples.m**

```
out_data_matrix(i,1) = signal_sample;
```

Projekt filtru AHDL ⇔ MATLAB

Plik **output_data_wr.m**

```
fp = 44100;
t = (0:1/fp:0.02);
n = length(t);
outputs(1,:) = out_data_matrix(1:n, 1);

% plot
figure(2);
plot(t,outputs,'red');
xlabel('Czas[s]');
ylabel('Amplituda');
title('Sygnał wyjściowy');
disp('Done!');
```