**2. EXPERIMENTATION**

**CHAPTER 2.1: GOLDEN REFERANCE**

**Generating original Tx signal:**

Initial setup gives a clear console, clears all the preset values of variables and closes all the figures.

% Initial Setup

clear; clc; close all;

A random pulse has to be generated to visualize an ideal working of the algorithm.

The pulse of signal is designed to be of the length of 64 bauds and the pulse is names *theta.* All the bauds would hold the value between 0 and 1 as double [8 byte] floating point values. The signal *pulse* generates a complex double of *theta* with real and imaginary components.

% Create pulse to detect

rng('default');%random variable will be selected using default algorithm

PulseLen = 64;%64 bits will be the length of the pulse

theta = rand(PulseLen,1);%an array is generated with size 64x1 of values 0 to 1

pulse = exp(1i\*2\*pi\*theta);%the values are converted into complex form

The total length of the pulse to be transmitted is of 5000 bauds which consists of the actual 64 bauds peak of the pulse. The *PulseLoc* variable is randomly generated to add the peak to the generated location. The entire array of *TxSignal* is given the value zero upon which the *pulse* signal is overlaid.

% Insert pulse to Tx signal

rng('shuffle');%random variable will be selected using default algorithm

TxLen = 5000;%total transmition length will be 5000 bits

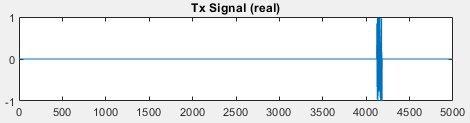
PulseLoc = randi(TxLen-PulseLen\*2);% defining where to insert the pulse

TxSignal = zeros(TxLen,1);% the rest 5000-64 = 4936 bits should be zero while transmitting

TxSignal(PulseLoc:PulseLoc+PulseLen1) = pulse; %merging both pulse to

5000 bits chai

Generated TxSignal: (the peak attained in this diagram is subjected to change for every run).



*Fig 2.1. Original Tx Signal*

The *TxSignal* attained so far is the signal that is required at the receiver end but to mimic the actual transmission of the signal AWGN is added to the signal and will be given a name *RxSignal.*

**Adding noise to the original signal:**

*Noise* signal holds a value random complex numbers which would be added onto the *TxSignal.* The *RxSignal* is scaled to fit the entire range between [-1, 1].

% Create Rx signal by adding noise

Noise = complex(randn(TxLen,1),randn(TxLen,1));%random complex numbers

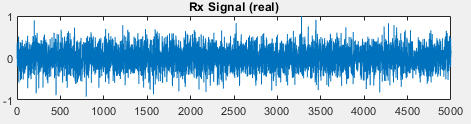
RxSignal = TxSignal + Noise; % addding noise to signal

% Scale Rx signal to +/- one

scale1 = max([abs(real(RxSignal)); abs(imag(RxSignal))]); %converting it for convinience

RxSignal = RxSignal/scale1;%dividing it by max value of signal

Generated RxSignal :



*Fig 2.2. Noise added Rx Signal*

**Retrieving original signal:**

A Correlation filter is used to retrieve back the algorithm. Where the conjugate of the inversed pulse is provided as *CorrFilter* signal. The task is to find the position of the pulse. The *FilterOut* Signal correlates the *RxSignal* against the matched filter to find the location of the signal, which also finds the peak value of the *pulse.*

% Create matched filter coefficients

CorrFilter = conj(flip(pulse))/PulseLen;%step 1 of corelation filter

% Correlate Rx signal against matched filter

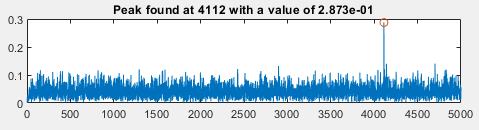
FilterOut = filter(CorrFilter,1,RxSignal);%step 2 of corelation filter

% Find peak magnitude & location

[peak, location] = max(abs(FilterOut));%getting the value of peak at filter

output

The output after the filter.



*Fig 2.3. Reconstructed Rx Signal*

The following commands are used to print the values and the graphs in the output screen.

% Print results

figure(1)%creating a figure

subplot(311); plot(real(TxSignal)); title('Tx Signal (real)');%the signal without noise

subplot(312); plot(real(RxSignal)); title('Rx Signal (real)');%the signal with noise

t = 1:length(FilterOut);%length of signal

str = sprintf('Peak found at %d with a value of %.3d',location,peak);%print the value of peak

subplot(313); plot(t,abs(FilterOut),location,peak,'o'); title(str);%the filtered signal

The above used code is used as the golden reference to stimulate the same using a hardware friendly model using **simulink**.

**CHAPTER 2.2: HARDWARE FRIENDLY MODEL**

**Window Creation:**

Measuring 5000 bauds can be done with ease by compiler at Matlab but to make it hardware applicable a window method is used where, a window of 11 bauds will be taken at a time and the middle baud (the 6th one) is checked to exceed the threshold value.

% Implementing the same above mentioned algorithm in a hardware friendly manner.

WindowLen = 11;%comparing the 5000 bits with 11 bits at a time serially

MidIdx = ceil(WindowLen/2);%gets the value of 6th bit in the series

threshold = 0.03;%only the peak would be more than this value.

%note: for convenience the threshold value was adjusted from experimental

       %trial and error method

Attaining a square root of a complex variable could increase the complexity of the circuit, hence *MagSqOut* signal is used such that the absolute value of the signal is squared hence making calculations easier.

% Compute magnitude squared to avoid sqrt operation

MagSqOut = abs(FilterOut).^2;

The Window of 5000 bauds is let to slide comparing the threshold value at each iteration. The *MidSample* and *CompareOut* confirms that the middle value satisfies the conditions only at the start of the actual pulse.

% Sliding window operation

for n = 1:length(FilterOut)-WindowLen %1 to 4989

% Compare each value in the window to the middle sample via subtraction

    DataBuff = MagSqOut(n:n+WindowLen-1);%stream of 11 bits

    MidSample = DataBuff(MidIdx);%bit 6 of the sample

    CompareOut = DataBuff - MidSample; % this is a vector

 % if all values in the result are negative and the middle sample is

    % greater than a threshold, it is a local max

    if all(CompareOut <= 0) && (MidSample > threshold)

        peak\_2 = MidSample;

        location\_2 = n + (MidIdx-1);%gives the peak location

    end

end

**Fixed point insertion:**

A fixed point is much feasible to use in a hardware model than a floating point. Hence tools from Fixed point designer toolbox is used to convert the values into fixed point sacrificing precision for bandwidth.

% Simulate model in fixed-point or floating-point

fxpt\_mode = true;

if fxpt\_mode  % fixed-point

    DT\_input = fixdt(1,16,14);%fixdt(signed/unsigned',wordlength,fractionlength)

    DT\_filter = fixdt(1,18,15);

    DT\_power = fixdt(1,18,11);

else  % floating-point

    DT\_input = 'double';%64 bit floating point

    DT\_filter = 'double';

    DT\_power = 'double';

end

DT\_coeff = fixdt(1,18); % coeff is treated as double if input is double

Converting column vector to row.

if iscolumn(CorrFilter)

    CorrFilter = transpose(CorrFilter); % need row vector for filter block

end

SimTime fixes the time for which the model will run. And slout instantiate the project.

%stimulation time to be length of the signal plus 30 time units

%30 time units= 10 in front end + 10 in back end + 10 buffer

SimTime = length(RxSignal) + WindowLen + 30;

% Simulate model

slout = sim('project\_detect');

Two functions have been instantiated which will be called multiple times during simulation.

% Correlation filter output

FilterOutSL = getLogged(slout,'filter\_out');

FilterValid = getLogged(slout,'filter\_valid');

FilterOutSL = FilterOutSL(FilterValid);%signal through filter

%to compare and print logged signal values

compareData(real(FilterOut),real(FilterOutSL),{2 3 1},'ML vs SL correlator output (re)');

compareData(imag(FilterOut),imag(FilterOutSL),{2 3 2},'ML vs SL correlator output (im)');

% Magnitude squared output

MagSqSL = getLogged(slout,'mag\_sq\_out');

MagSqSL = MagSqSL(FilterValid);%magnitude square of filter

%to compare and print logged signal values

compareData(MagSqOut,MagSqSL,{2 3 3},'ML vs SL mag-squared output');

% Peak value

MidSampleSL = getLogged(slout,'mid\_sample');

Detected = getLogged(slout,'detected');

PeakSL = double(MidSampleSL(Detected>0));

The output of the code is displayed in the figure using print commands.

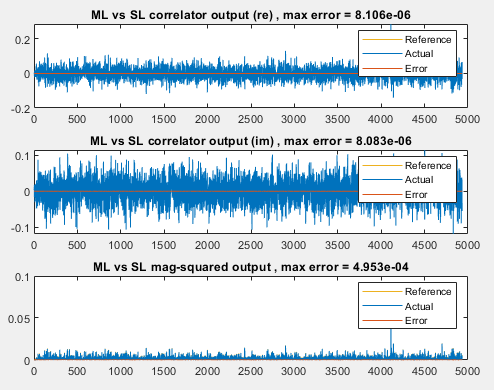
fprintf('\nPeak location = %d, magnitude = %.3d using global max\n',location,peak);

fprintf('Peak location = %d, mag-squared = %.3d using local max\n',location\_2,peak\_2);

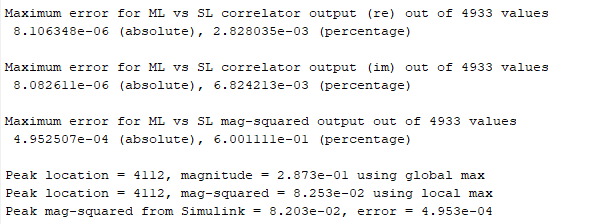
fprintf('Peak mag-squared from Simulink = %.3d, error = %.3d\n',PeakSL,abs(peak\_2-PeakSL));

**Output:**

* The red legend in the output is the measure of deviation of the simulink model’s output to that of the Matlab’s golden reference, which stays at zero indicating perfect working of the hardware model.
* Yellow legend is hidden in the graph as the blue legend overlaps the reference plot.
* The error is in terms of 10-6 is caused due to the fixed point conversion, which could be neglected.



*Fig 2.4. Rx and Tx Signals through hardware friendly code*



*Fig 2.5- Command Window Output*

**CHAPTER 2.3: FUNCTION BLOCKS**

**The two functions used in the above programs:**

**1. Getlogged:**

Squeeze removes a dimension form the array. This function is used to log all the values in the simulink model.

Empty signal is also eliminated and it triggers an error while that happens. logsout, getElement are key triggers used to transfer the signal for one sub block to other.

%%Defining the functions used in the code above

%getLogged function

function signal\_val = getLogged(simout\_obj,signal\_name)%function instantiation

logsout = simout\_obj.logsout;%load logged signal

%checking for possible error in outputs

if isempty(logsout)

    error('No logged signal found. Make sure ''%s'' is logged in the model',...

        signal\_name);

end

sig = logsout.getElement(signal\_name);%read elements of logged signal

%checking for possible error in outputs

if isempty(sig)

    error('Signal ''%s'' not found. Make sure it is logged and named correctly.',...

        signal\_name);

end

signal\_val = squeeze(sig.Values.Data);

%squeeze returns an array with the same elements as the input,...

%but with dimensions of length 1 removed.

end

**2. Compare data:**

This function doesn’t affect the hardware model but is used to find the differences between both, hence would help us solve the bugs.

%compareData function

%function instantiation

function err\_vec = compareData(reference,actual,figure\_number,textstring)

% Vector input only

if ~isvector(reference) || ~isvector(actual)

    error('Input signals must be vector');%check for inputs to be vector

else

    if isrow(reference)

        reference = transpose(reference);%convert to column vector

    end

    if isrow(actual)

        actual = transpose(actual);%convert to column vector

    end

end

% Make signals same length if necessary

if length(reference) ~= length(actual)

%     warning(['Length of reference (%d) is not the same as actual signal (%d).'...

%         ' Truncating the longer input.'],length(reference),length(actual));

    %balance length of both the vectors to be equal

    len = 1:min(length(reference),length(actual));

    reference = reference(len);%length modification of reference

    actual = actual(len);%length modification of actual

end

% Turn complex into vector

if xor(isreal(reference),isreal(actual))%checking for nature of both signals

    error('Input signals are not both real or both complex');

elseif ~isreal(reference)

    ref\_vec = double([real(reference) imag(reference)]);%covert to double

    act\_vec = double([real(actual) imag(actual)]);%covert to double

    tag = {'(Real)','(Imag)'};

else

    ref\_vec = double(reference);%covert to double

    act\_vec = double(actual);%covert to double

    tag = {''};

end

% Configure figure

if iscell(figure\_number)

    if size(ref\_vec,2) > 1 % complex

        error('Cannot yet subplot multiple complex inputs');

    else

        figure(figure\_number{1})%defines figure(2)

    end

else

    figure(figure\_number)

end

c = get(groot,'defaultAxesColorOrder');%used at plotting

% Compute error

err\_vec = ref\_vec - act\_vec;%error vector to be printed on output

max\_err = max(abs(err\_vec));%max error to be printed on output

max\_ref = max(abs(ref\_vec));%max reference to be printed on output

The calculated output is sent back to the main screen using print commands.

%output printed on command window

fprintf('\nMaximum error for %s out of %d values\n',textstring,length(actual));

for n = 1:size(ref\_vec,2)

    %output printed on Command window

    fprintf('%s %d (absolute), %d (percentage)\n',tag{n},max\_err(n),max\_err(n)/max\_ref(n)\*100);

    if iscell(figure\_number)

        total\_plot = figure\_number{2};% equal to 3

        plot\_num = figure\_number{3};%figure number can be 1 to 3

    else

        total\_plot = size(ref\_vec,2);

        plot\_num = n;

    end

    subplot(total\_plot,1,plot\_num)%subplot generation

    plot(ref\_vec(:,n),'Color',c(3,:));%uses 1st color for reference

    hold on %plots 3 graphs over the same plot

    plot(act\_vec(:,n),'Color',c(1,:));%uses 2nd color for actual

    plot(err\_vec(:,n),'Color',c(2,:));%uses 3rd color for error

    legend('Reference','Actual','Error')%defining legend

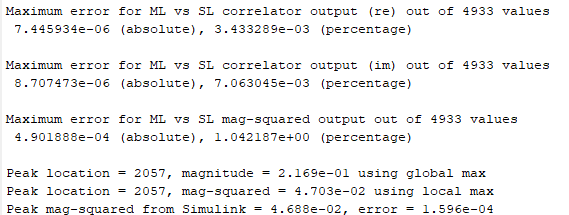
    hold off

    title(sprintf('%s %s, max error = %.3d',textstring,tag{n},max\_err(n)));%title

end

end

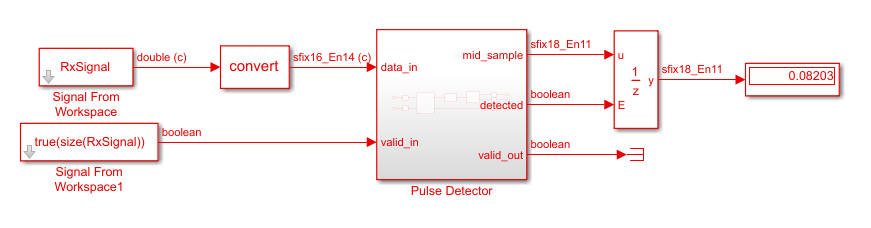
Command prompt output:



*Fig 2.6. Command Window Output*

**CHAPTER 2.4: SIMULINK MODEL**

**Top model:**

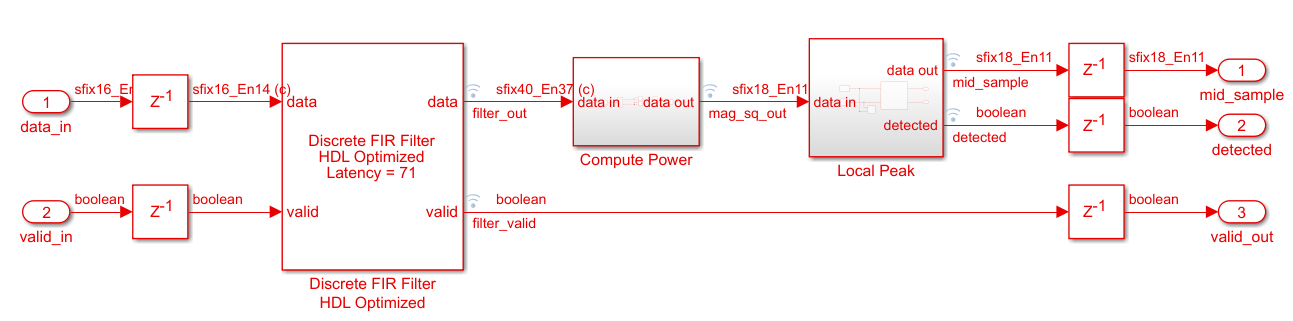


*Fig 2.7. Block diagram of the top module*

The *RxSignal* is given in parallel to valid in signal to trigger the incoming of received signal. The convert block converts the double precision floating point value to fixed point value. The output is displayed in the output block which is controlled by a unit delay block to avoid glitches.

**Pulse Detector Top module:**

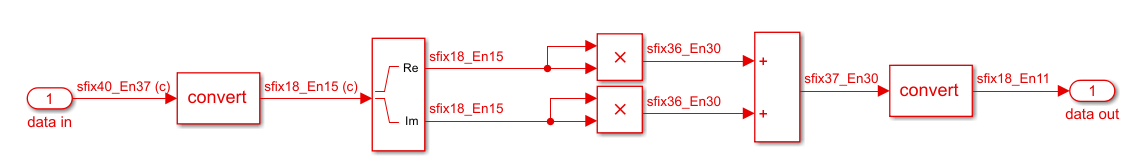
This block will be the main module in the synthesis of RTL and other sub blocks will be instantiated from this top module.

 *Fig 2.8. Block diagram of the pulse detector module*

The signal is fed into the discrete FIR filter which then sends out the filter out signal. The filter\_out signal is logged into Compute power sub block.

**Compute Power sub module:**

The incoming fixed point value is converted into less precision value of the same and is squared in a hardware friendly manner. The result is then logged to the next sub block.

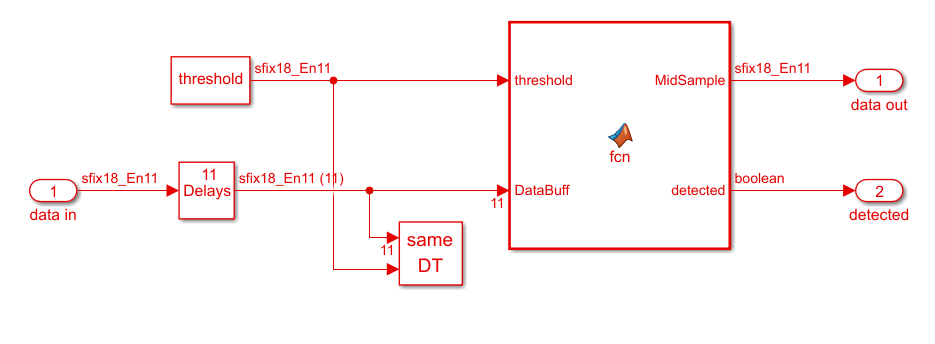


*Fig 2.9. Block diagram of compute power sub module*

**Local Peak sub module:**

11 values are collected using the delay block which is connected to same DT block which checks if the connected inputs are of same data type. If not it forces all the input to be of same type.

The fcn block holds a matlab code, which checks for the peak, if detected will send out the data in data out port and sets the detected signal to be high. If not the window is updated to receive the next bit contnuing till detection of the peak.



*Fig 2.10. Block diagram of local peak module*

**fcn code:**

% Hardware friendly implementation of peak finder

%

% Function inputs:

% \* WindowLen - non-tunable parameter defined under Simulink->Edit Data

% \* threhold  - input port (connected to constant)

% \* DataBuff  - input port (buffering done using Simulink block)

%

% Function outputs:

% \* "detected" is set when MidSample is local max

function [MidSample,detected] = fcn(threshold, DataBuff, WindowLen)

%#codegen

MidIdx = ceil(WindowLen/2);

% Compare each value in the window to the middle sample via subtraction

MidSample = DataBuff(MidIdx);

CompareOut = DataBuff - MidSample; % this is a vector

% if all values in the result are negative and the middle sample is

% greater than a threshold, it is a local max

if all(CompareOut <= 0) && (MidSample > threshold)

    detected = true;

else

    detected = false;

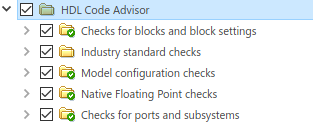
end

This mimics the part of window code in Matlab but has few keywords converted to hardware friendly Boolean terms such as true and false.

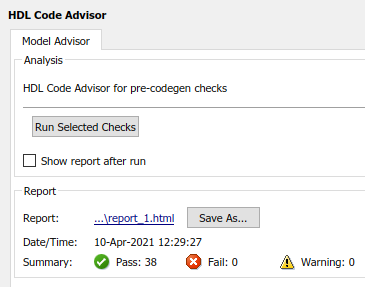
**CHAPTER 2.5: HDL CODER**

**HDL Coder:**

The DUT compatibility is checked using HDL coder, checker not only ensures the blocks used in the subsystem is real compatible but also ensures the settings, ports and configurations of these blocks do not generate inefficient hardware. The HDL model checker includes options for native floating-point and industry standard checks.



*Fig 2.11. HDL Coder Compatibility check*



*Fig 2.12. HDL Code Advisor Output*

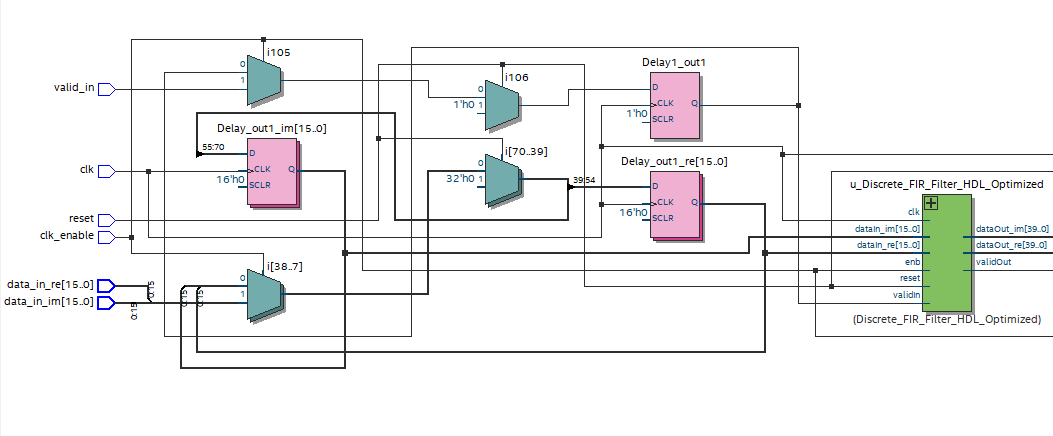
Once the checks are successful, the HDL workflow advisor has to be run to initiate generation of HDL code. After generation of HDL code, Quartus prime is used to visualize the resources and RTL design.

**3. RESULTS AND OUTPUTS**

**CHAPTER 3.1: RTL DESIGN OF MODULES**

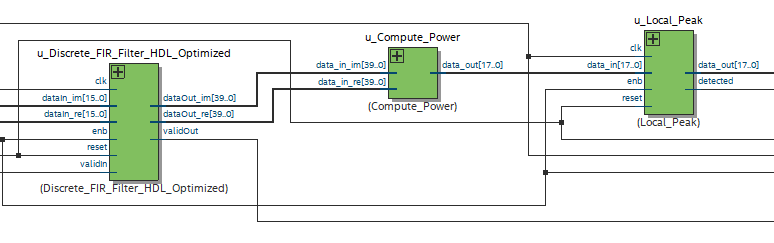
**Pulse Detector module:**

**a)**

****

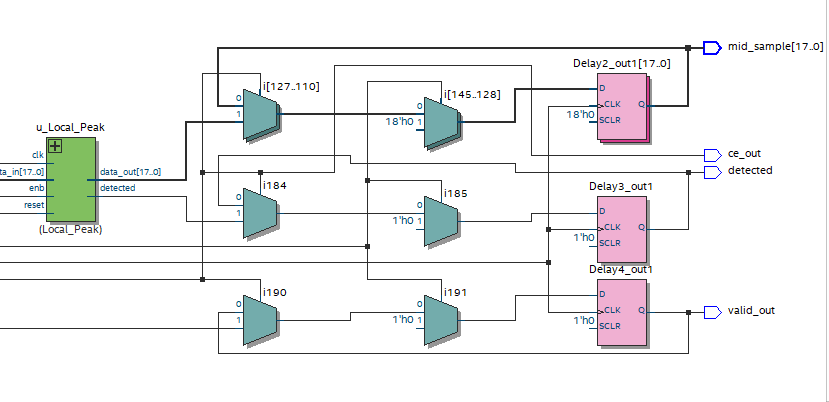
*Fig 3.1. Pulse Detector left part of RTL*

**b)**

****

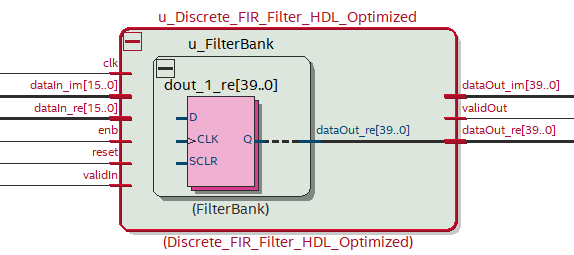
*Fig 3.2. Pulse Detector middle part of RTL*

**c)**

****

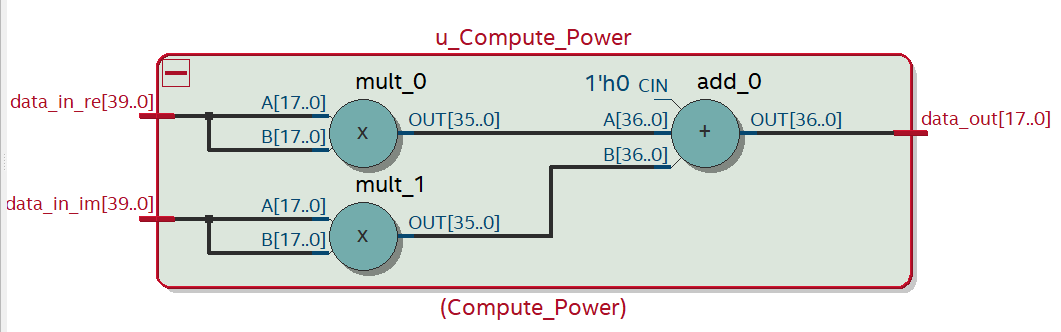
*Fig 3.3 . Pulse Detector right part of RTL*

**FIR filter sub module:**



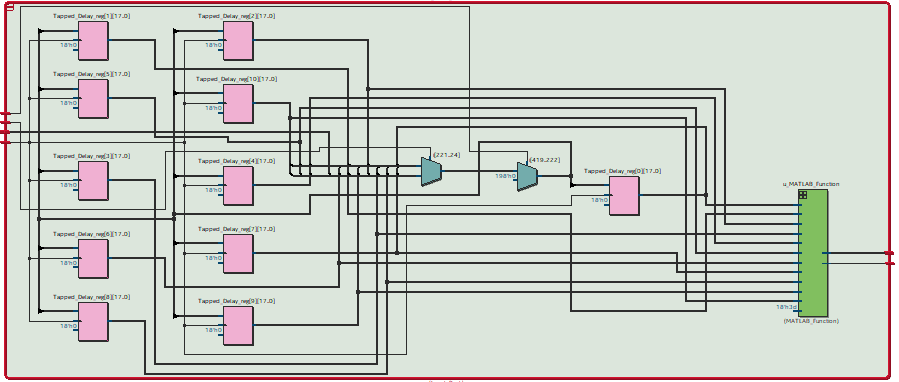
*Fig 3.4 .RTL Design of Filter sub module*

**Compute Power sub module:**



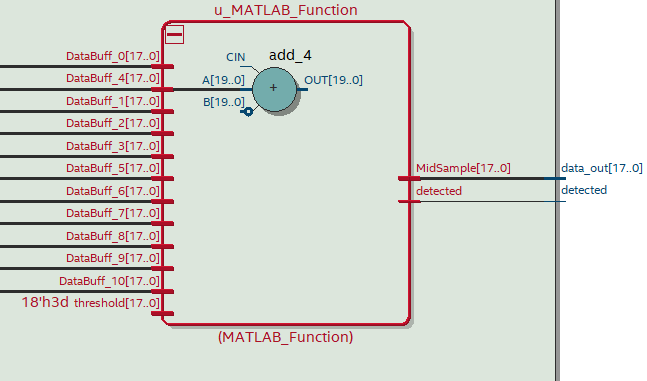
*Fig 3.5 .RTL Design of Compute Power sub module*

**Local peak sub module:**



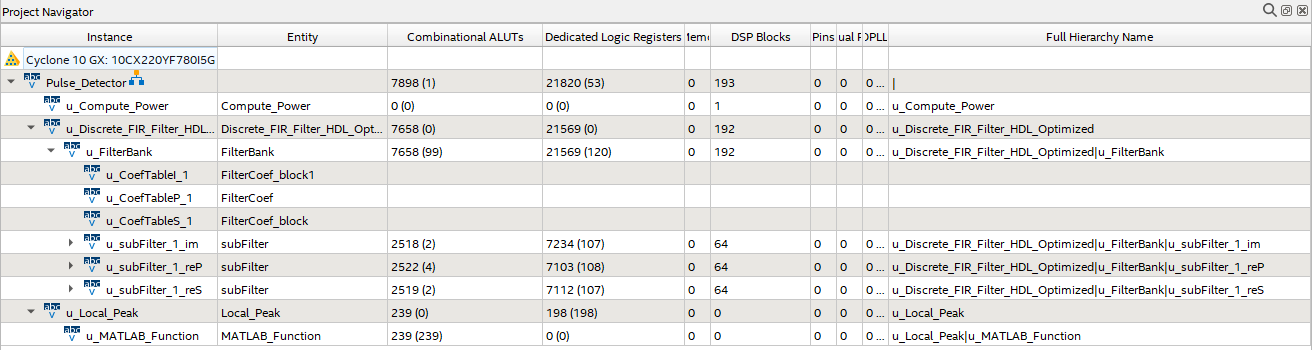
*Fig 3.6 .RTL Design of Local Peak sub module*

**MATLAB function sub module:**



*Fig 3.7 .RTL Design of MATLAB function sub module*

**Modules hierarchy and resource utilization:**



*Fig 3.8 .Resource utilization details*

**CHAPTER 3.2: VERILOG CODES**

**Pulse detector:**

`timescale 1 ns / 1 ns

module Pulse\_Detector

          (clk,

           reset,

           clk\_enable,

           data\_in\_re,

           data\_in\_im,

           valid\_in,

           ce\_out,

           mid\_sample,

           detected,

           valid\_out);

  input   clk;

  input   reset;

  input   clk\_enable;

  input   signed [15:0] data\_in\_re;  // sfix16\_En14

  input   signed [15:0] data\_in\_im;  // sfix16\_En14

  input   valid\_in;

  output  ce\_out;

  output  signed [17:0] mid\_sample;  // sfix18\_En11

  output  detected;

  output  valid\_out;

  wire enb;

  reg signed [15:0] Delay\_out1\_re;  // sfix16\_En14

  reg signed [15:0] Delay\_out1\_im;  // sfix16\_En14

  reg  Delay1\_out1;

  wire signed [39:0] filter\_out\_re;  // sfix40\_En37

  wire signed [39:0] filter\_out\_im;  // sfix40\_En37

  wire filter\_valid;

  wire signed [17:0] mag\_sq\_out;  // sfix18\_En11

  wire signed [17:0] mid\_sample\_1;  // sfix18\_En11

  wire detected\_1;

  reg signed [17:0] Delay2\_out1;  // sfix18\_En11

  reg  Delay3\_out1;

  reg  Delay4\_out1;

  assign enb = clk\_enable;

  always @(posedge clk)

    begin : Delay\_process

      if (reset == 1'b1) begin

        Delay\_out1\_re <= 16'sb0000000000000000;

        Delay\_out1\_im <= 16'sb0000000000000000;

      end

      else begin

        if (enb) begin

          Delay\_out1\_re <= data\_in\_re;

          Delay\_out1\_im <= data\_in\_im;

        end

      end

    end

  always @(posedge clk)

    begin : Delay1\_process

      if (reset == 1'b1) begin

        Delay1\_out1 <= 1'b0;

      end

      else begin

        if (enb) begin

          Delay1\_out1 <= valid\_in;

        end

      end

    end

  Discrete\_FIR\_Filter\_HDL\_Optimized u\_Discrete\_FIR\_Filter\_HDL\_Optimized (.clk(clk),

                                                                         .reset(reset),

                                                                         .enb(clk\_enable),

                                                                         .dataIn\_re(Delay\_out1\_re),  // sfix16\_En14

                                                                         .dataIn\_im(Delay\_out1\_im),  // sfix16\_En14

                                                                         .validIn(Delay1\_out1),

                                                                         .dataOut\_re(filter\_out\_re),  // sfix40\_En37

                                                                         .dataOut\_im(filter\_out\_im),  // sfix40\_En37

                                                                         .validOut(filter\_valid)

                                                                         );

  Compute\_Power u\_Compute\_Power (.data\_in\_re(filter\_out\_re),  // sfix40\_En37

                                 .data\_in\_im(filter\_out\_im),  // sfix40\_En37

                                 .data\_out(mag\_sq\_out)  // sfix18\_En11

                                 );

  Local\_Peak u\_Local\_Peak (.clk(clk),

                           .reset(reset),

                           .enb(clk\_enable),

                           .data\_in(mag\_sq\_out),  // sfix18\_En11

                           .data\_out(mid\_sample\_1),  // sfix18\_En11

                           .detected(detected\_1)

                           );

  always @(posedge clk)

    begin : Delay2\_process

      if (reset == 1'b1) begin

        Delay2\_out1 <= 18'sb000000000000000000;

      end

      else begin

        if (enb) begin

          Delay2\_out1 <= mid\_sample\_1;

        end

      end

    end

  assign mid\_sample = Delay2\_out1;

  always @(posedge clk)

    begin : Delay3\_process

      if (reset == 1'b1) begin

        Delay3\_out1 <= 1'b0;

      end

      else begin

        if (enb) begin

          Delay3\_out1 <= detected\_1;

        end

      end

    end

  assign detected = Delay3\_out1;

  always @(posedge clk)

    begin : Delay4\_process

      if (reset == 1'b1) begin

        Delay4\_out1 <= 1'b0;

      end

      else begin

        if (enb) begin

          Delay4\_out1 <= filter\_valid;

        end

      end

    end

  assign valid\_out = Delay4\_out1;

  assign ce\_out = clk\_enable;

endmodule  // Pulse\_Detector

**FIR filter:**

`timescale 1 ns / 1 ns

module Discrete\_FIR\_Filter\_HDL\_Optimized

          (clk,

           reset,

           enb,

           dataIn\_re,

           dataIn\_im,

           validIn,

           dataOut\_re,

           dataOut\_im,

           validOut);

  input   clk;

  input   reset;

  input   enb;

  input   signed [15:0] dataIn\_re;  // sfix16\_En14

  input   signed [15:0] dataIn\_im;  // sfix16\_En14

  input   validIn;

  output  signed [39:0] dataOut\_re;  // sfix40\_En37

  output  signed [39:0] dataOut\_im;  // sfix40\_En37

  output  validOut;

  FilterBank u\_FilterBank (.clk(clk),

                           .reset(reset),

                           .enb(enb),

                           .dataIn\_re(dataIn\_re),  // sfix16\_En14

                           .dataIn\_im(dataIn\_im),  // sfix16\_En14

                           .validIn(validIn),

                           .dataOut\_re(dataOut\_re),  // sfix40\_En37

                           .dataOut\_im(dataOut\_im),  // sfix40\_En37

                           .validOut(validOut)

                           );

endmodule  // Discrete\_FIR\_Filter\_HDL\_Optimized

The submodule Filterbank used is attached in the <https://github.com/raja-aadhithan/RTL-design-for-MATLAB-model/tree/main/verilog_syntesis> page, which consists of further submodules namely : subFilter, FilterCoef\_block, FilterCoef\_block1.

**Compute power:**

`timescale 1 ns / 1 ns

module Compute\_Power

          (data\_in\_re,

           data\_in\_im,

           data\_out);

  input   signed [39:0] data\_in\_re;  // sfix40\_En37

  input   signed [39:0] data\_in\_im;  // sfix40\_En37

  output  signed [17:0] data\_out;  // sfix18\_En11

  wire signed [17:0] Data\_Type\_Conversion1\_out1\_re;  // sfix18\_En15

  wire signed [17:0] Data\_Type\_Conversion1\_out1\_im;  // sfix18\_En15

  wire signed [35:0] Product\_out1;  // sfix36\_En30

  wire signed [35:0] Product1\_out1;  // sfix36\_En30

  wire signed [36:0] Add\_add\_cast;  // sfix37\_En30

  wire signed [36:0] Add\_add\_cast\_1;  // sfix37\_En30

  wire signed [36:0] Add\_out1;  // sfix37\_En30

  wire signed [17:0] Data\_Type\_Conversion\_out1;  // sfix18\_En11

  assign Data\_Type\_Conversion1\_out1\_re = data\_in\_re[39:22];

  assign Data\_Type\_Conversion1\_out1\_im = data\_in\_im[39:22];

  assign Product\_out1 = Data\_Type\_Conversion1\_out1\_re \* Data\_Type\_Conversion1\_out1\_re;

  assign Product1\_out1 = Data\_Type\_Conversion1\_out1\_im \* Data\_Type\_Conversion1\_out1\_im;

  assign Add\_add\_cast = {Product\_out1[35], Product\_out1};

  assign Add\_add\_cast\_1 = {Product1\_out1[35], Product1\_out1};

  assign Add\_out1 = Add\_add\_cast + Add\_add\_cast\_1;

  assign Data\_Type\_Conversion\_out1 = Add\_out1[36:19];

  assign data\_out = Data\_Type\_Conversion\_out1;

endmodule  // Compute\_Power

**Local Peak:**

`timescale 1 ns / 1 ns

module Local\_Peak

          (clk,

           reset,

           enb,

           data\_in,

           data\_out,

           detected);

  input   clk;

  input   reset;

  input   enb;

  input   signed [17:0] data\_in;  // sfix18\_En11

  output  signed [17:0] data\_out;  // sfix18\_En11

  output  detected;

  wire signed [17:0] Constant\_out1;  // sfix18\_En11

  reg signed [17:0] Tapped\_Delay\_reg [0:10];  // sfix18 [11]

  wire signed [17:0] Tapped\_Delay\_reg\_next [0:10];  // sfix18\_En11 [11]

  wire signed [17:0] Tapped\_Delay\_out1 [0:10];  // sfix18\_En11 [11]

  wire signed [17:0] MidSample;  // sfix18\_En11

  assign Constant\_out1 = 18'sb000000000000111101;

  always @(posedge clk)

    begin : Tapped\_Delay\_process

      if (reset == 1'b1) begin

        Tapped\_Delay\_reg[0] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[1] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[2] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[3] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[4] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[5] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[6] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[7] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[8] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[9] <= 18'sb000000000000000000;

        Tapped\_Delay\_reg[10] <= 18'sb000000000000000000;

      end

      else begin

        if (enb) begin

          Tapped\_Delay\_reg[0] <= Tapped\_Delay\_reg\_next[0];

          Tapped\_Delay\_reg[1] <= Tapped\_Delay\_reg\_next[1];

          Tapped\_Delay\_reg[2] <= Tapped\_Delay\_reg\_next[2];

          Tapped\_Delay\_reg[3] <= Tapped\_Delay\_reg\_next[3];

          Tapped\_Delay\_reg[4] <= Tapped\_Delay\_reg\_next[4];

          Tapped\_Delay\_reg[5] <= Tapped\_Delay\_reg\_next[5];

          Tapped\_Delay\_reg[6] <= Tapped\_Delay\_reg\_next[6];

          Tapped\_Delay\_reg[7] <= Tapped\_Delay\_reg\_next[7];

          Tapped\_Delay\_reg[8] <= Tapped\_Delay\_reg\_next[8];

          Tapped\_Delay\_reg[9] <= Tapped\_Delay\_reg\_next[9];

          Tapped\_Delay\_reg[10] <= Tapped\_Delay\_reg\_next[10];

        end

      end

    end

  assign Tapped\_Delay\_out1[0] = Tapped\_Delay\_reg[0];

  assign Tapped\_Delay\_out1[1] = Tapped\_Delay\_reg[1];

  assign Tapped\_Delay\_out1[2] = Tapped\_Delay\_reg[2];

  assign Tapped\_Delay\_out1[3] = Tapped\_Delay\_reg[3];

  assign Tapped\_Delay\_out1[4] = Tapped\_Delay\_reg[4];

  assign Tapped\_Delay\_out1[5] = Tapped\_Delay\_reg[5];

  assign Tapped\_Delay\_out1[6] = Tapped\_Delay\_reg[6];

  assign Tapped\_Delay\_out1[7] = Tapped\_Delay\_reg[7];

  assign Tapped\_Delay\_out1[8] = Tapped\_Delay\_reg[8];

  assign Tapped\_Delay\_out1[9] = Tapped\_Delay\_reg[9];

  assign Tapped\_Delay\_out1[10] = Tapped\_Delay\_reg[10];

  assign Tapped\_Delay\_reg\_next[0] = Tapped\_Delay\_reg[1];

  assign Tapped\_Delay\_reg\_next[1] = Tapped\_Delay\_reg[2];

  assign Tapped\_Delay\_reg\_next[2] = Tapped\_Delay\_reg[3];

  assign Tapped\_Delay\_reg\_next[3] = Tapped\_Delay\_reg[4];

  assign Tapped\_Delay\_reg\_next[4] = Tapped\_Delay\_reg[5];

  assign Tapped\_Delay\_reg\_next[5] = Tapped\_Delay\_reg[6];

  assign Tapped\_Delay\_reg\_next[6] = Tapped\_Delay\_reg[7];

  assign Tapped\_Delay\_reg\_next[7] = Tapped\_Delay\_reg[8];

  assign Tapped\_Delay\_reg\_next[8] = Tapped\_Delay\_reg[9];

  assign Tapped\_Delay\_reg\_next[9] = Tapped\_Delay\_reg[10];

  assign Tapped\_Delay\_reg\_next[10] = data\_in;

  MATLAB\_Function u\_MATLAB\_Function (.threshold(Constant\_out1),  // sfix18\_En11

                                     .DataBuff\_0(Tapped\_Delay\_out1[0]),  // sfix18\_En11

                                     .DataBuff\_1(Tapped\_Delay\_out1[1]),  // sfix18\_En11

                                     .DataBuff\_2(Tapped\_Delay\_out1[2]),  // sfix18\_En11

                                     .DataBuff\_3(Tapped\_Delay\_out1[3]),  // sfix18\_En11

                                     .DataBuff\_4(Tapped\_Delay\_out1[4]),  // sfix18\_En11

                                     .DataBuff\_5(Tapped\_Delay\_out1[5]),  // sfix18\_En11

                                     .DataBuff\_6(Tapped\_Delay\_out1[6]),  // sfix18\_En11

                                     .DataBuff\_7(Tapped\_Delay\_out1[7]),  // sfix18\_En11

                                     .DataBuff\_8(Tapped\_Delay\_out1[8]),  // sfix18\_En11

                                     .DataBuff\_9(Tapped\_Delay\_out1[9]),  // sfix18\_En11

                                     .DataBuff\_10(Tapped\_Delay\_out1[10]),  // sfix18\_En11

                                     .MidSample(MidSample),  // sfix18\_En11

                                     .detected(detected)

                                     );

  assign data\_out = MidSample;

endmodule  // Local\_Peak

1. **CONCLUSION**

In this project I have analyzed the strength of MATLAB and Simulink and created the Simulink model of the pulse detection algorithm. Introduced design architecture options that takes a control over speed and area trade-offs and converted the Simulink design to fixed point.

Have generate and synthesize the optimized HDL code. Report generation are done in the HDL code generation. Have visualized and identified the pre and post routing timing information and highlight the critical parts in the model. This analysis used the annotate model which synthesis result option and enables the HDL coder to display the DUT with more accurate critical path timing with the steps in the workflow advisor. Have successfully generated and synthesized the optimized HDL code for the pulse detection algorithm implemented in MATLAB.

1. **EDA TOOLS USED**

**SCRIPTING LANGUAGE:** MATLAB

**TOOL USED:** MATLAB R2020b

**SIMULATOR:** MODEL SIM (Intel FPGA Started Edition)

**SYNTHESIZER:** QUARTUS **(**QUARTUS PRIME PRO 20.4)

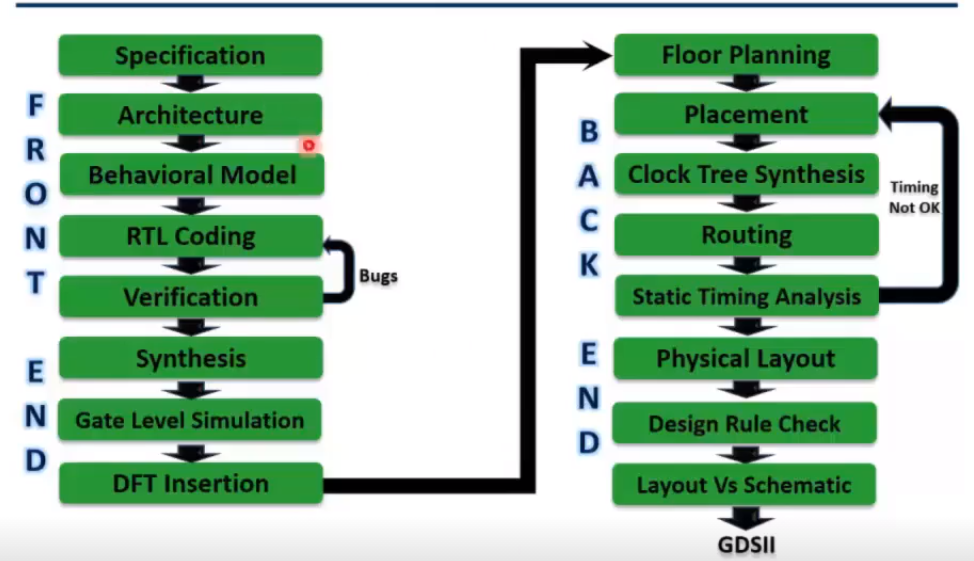
**IDE (BLOCK DIAGRAM):** SIMULINK

**TOOLBOX USED:**

* FIXED POINT DESIGNER
* HDL CODER
* SIGNAL PROCESSING
* DSP SYSTEM

1. **FUTURE SCOPE**

**Current state:** The project covers the First half of Front end design of the Chip. Attached below is the work flow of a ASIC/FPGA chip from specification to production.



*Fig 6.1 . VLSI Design Flow*

**Possible future scopes:**

* Using System Verilog and methodologies like UVM the verification can be done.
* After which synthesis can be done once STA is done.
* On completion of the above steps, the design can be pushed on to the back end.
* On completion of the 8 stages of back end, the GDSII file is generated which can be sent to the foundry.

**REFERENCES**

* <https://www.mathworks.com/matlabcentral/fileexchange>
* <https://www.mathworks.com/help/fixedpoint/ug/view-fixed-point-number-circles.html>
* <http://msdl.cs.mcgill.ca/people/mosterman/presentations/date07/tutorial.pdf>
* <https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user>
* <https://www.mathworks.com/help/hdlcoder/ug/system-design-with-hdl-code-generation-from-matlab-and-simulink.html>
* <https://www.techsource-asia.com/generating-hdl-code-from-simulink>
* <https://www.matlabexpo.com/content/dam/mathworks/mathworks-dot-com/images/events/matlabexpo/in/2017/accelerating-fpga-asic-design-verification.pdf>