EEC0055 - Digital Systems Design

2019/2020

Laboratory 3
17 November – 24 December 2019

Ultrasonic wind speed and direction sensor

Revision history

date	notes	author
Nov 18, 2019	Preliminary version V0.1	jca@fe.up.pt
Nov 25, 2019	Version V0.2 – Adds a preliminary specification of data formats	jca@fe.up.pt
Nov 29, 2019	Version V0.3 – Adds the functional description of the modules to design; revises the specification of the number of bits and numeric format in each path; reorganizes the sub-sections inside section 5 (Design implementation).	jca@fe.up.pt
Dec 12, 2019	Version 0.4 – Adds section 6 with the description of the verification kit, including the Matlab scripts and testbench modules.	jca@fe.up.pt

1 – Introduction

In this project we will implement an ultrasonic wind speed and direction sensor, based on the variation of the sound speed due to the movement of the air (the sound wave propagation medium) induced by the wind. The system will be designed using the XILINX ISE design environment and the Verilog Hardware Description Language, and implemented in the ATLYS board. Some modules will be provided as the source RTL code, as well as reference implementations as Matlab scripts. The CORDIC module developed in the first project will also be a fundamental building block. The overall design goal is minimize the logic complexity (area) while using a low clock frequency (2MHz) to save power.

2 – Principle of operation

The ultrasonic wind sensor calculates the wind speed and the wind direction by measuring the phase differences between the acoustic signals received by four orthogonally placed receivers. Figure 1 shows the physical arrangement of the ultrasonic transducers. A central transducer continuously transmits a single tone at frequency **Ftx** that is received by the four receivers. In the absence of wind, and assuming the 4 receivers are at the same distance **D** from the transmitter, the four signals received will be in phase (although out of phase from the transmitted signal).

If the air is moving along the direction formed by two opposite receivers (**Rx1** and **Rx3** or **Rx2** and **Rx4**) the sound propagation speed from the transmitter to the receiver downwind will increase and the speed to the upwind receiver will decrease. This will create a phase shift in the signals received, positive in the downwind receiver (signal arrives first) and negative in the upwind receiver. Measuring the phase difference between the signals arriving to two opposite receivers (**Rx1** and **Rx3** or **Rx2** and **Rx4**) will allow to determine the Y and X components of the wind speed along the directions defined by the sensors. Then, using a CORDIC module these values are converted from the rectangular X and Y components of the wind speed to polar coordinates, representing the wind speed modulus and the wind direction.

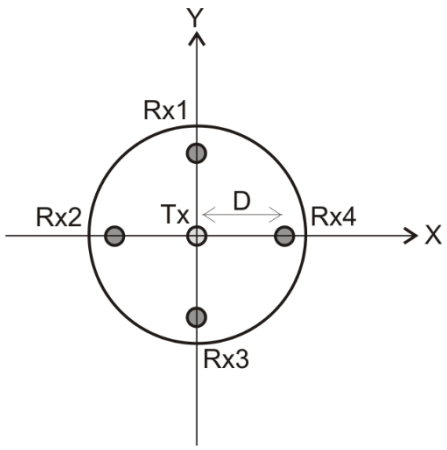

Figure 1 – Physical arrangement of the transmitter and receiver ultrasonic transducers.

The relationship between the phase difference and the wind speed depends on the propagation speed of the sound, which varies approximately linearly with the air temperature. However, because the air temperature will affect equally the X and Y components of the wind speed, the temperature must be considered only for the calculation of the modulus of the wind speed and will not impact the result of the wind direction. The modulus of the wind speed is compensated for the current air temperature by a custom correction module.

Two final calibration modules, based on the linear interpolation over a piece-wise linear curve, adjust the final wind speed and wind direction against deviations due to misalignments of the ultrasonic transducers. The output interface will be based on a low speed bidirectional serial interface (I2C, SPI or UART, to be defined later). The serial interface will also be used to configure various system parameters and to define the calibration functions.

3 – System architecture

Figure 2 presents a simplified functional block diagram of the system. System blocks are:

- The ultrasonic transmitter placed at the center of the sensor is driven by a fixed frequency square wave generated by a clock divider (block **txdriver**). The frequency of this signal will be around 20 kHz but the precise value will be defined later.
- The four acoustic signals acquired by the ultrasonic transducers are amplified and filtered by an external analog circuit, and digitized by a 4-channel multiplexed ADC (as the AD7091-R4 from Analog Devices).
- The block **rxreceiver** implements the serial interface with the ADC and provides the 4 streams of data with the 12-bit digital samples acquired from the 4 ultrasonic receivers.
- To determine the difference of phase between the signals received at opposite receivers, the digital signals coming from the acoustic receivers are converted to their complex representation. To achieve this, a set four modules based on Hilbert FIR filters (blocks **real2cpx**) calculate the real and imaginary components of each signal.
- The instantaneous phase of each signal is calculated from the real and imaginary components by a CORDIC module (the four modules **phasecalc**) and the phase differences between each two opposite receivers are calculated, wrapped to the interval [-180º, +180º] and converted to time by the two modules **phasediff**.
- Then, the phase differences are averaged and converted to the wind speed components along the X and Y axis (modules **phase2speed**).
- An additional CORDIC module converts the X and Y wind speed components to the wind direction and the modulus of the wind speed (module **windrec2pol**).
- As the X and Y wind speed calculation considers a fixed 20 ºC air temperature, the modulus of the wind speed has to be compensated to the current air temperature by module **tempcomp**.
- To compensate misalignments of the acoustic receivers, two calibration modules (**calibwspd**, **calibwdir**) adjust the values of the wind angle and the wind speed implementing linear interpolations over a piecewise linear functional stored in a memory.
- Finally, the serial interface (**serialinterface**) implements the digital interface with the external system.

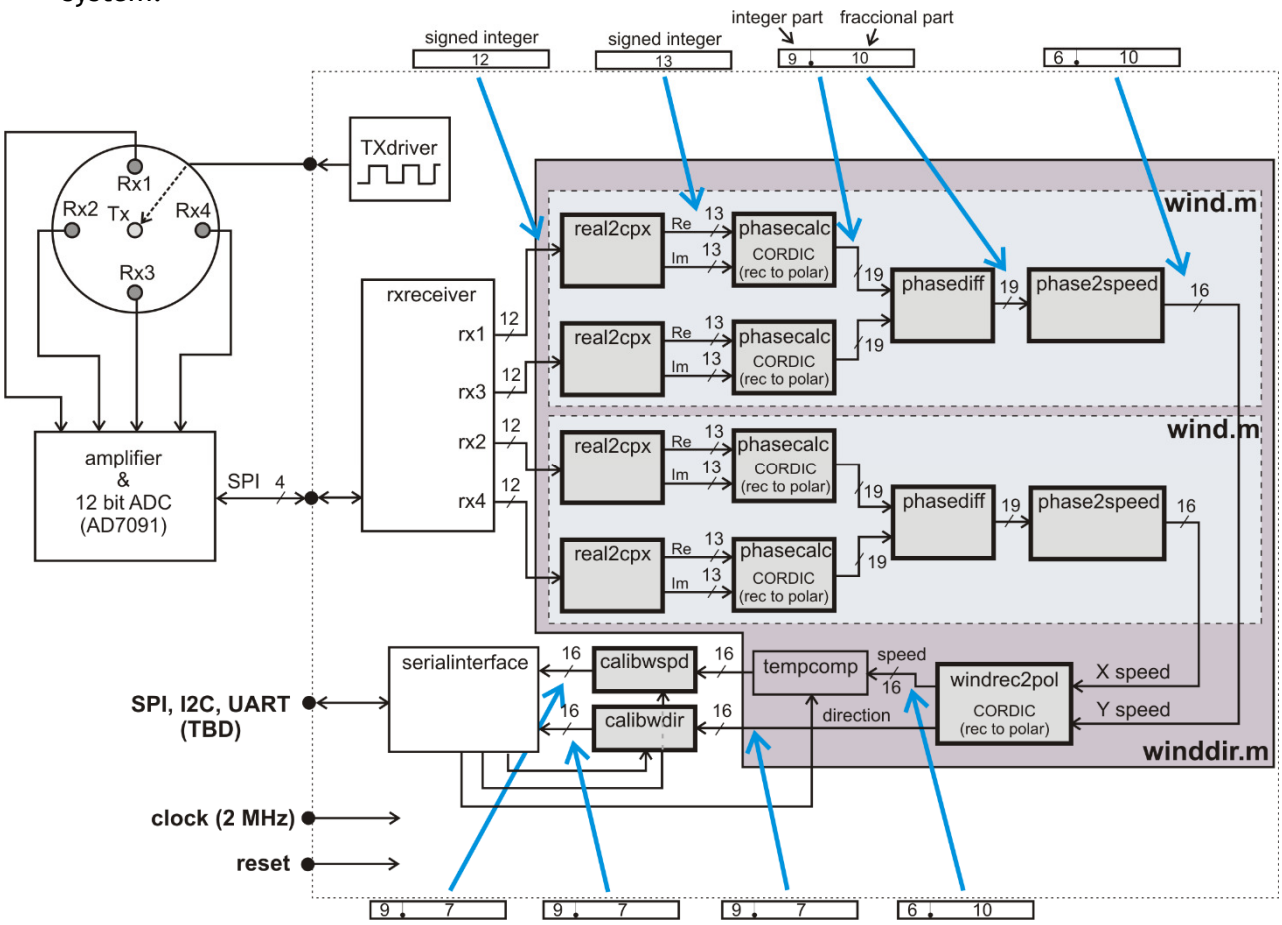

Figure 2 – Wind sensor bock diagram. Note this is only a functional view as some blocks may be shared between the two identical signal processing paths.

4 – Matlab functional model
A Matlab functional model is provided in files '**wind.m**', '**winddir.m**', '**cordic_angle.m**' and '**cordic.m**'. Download these scripts to your **./matlab** directory (**wind.m** and **windir.m** include a brief help text). Script **winddir.m** receives as arguments the true wind direction, wind speed and air temperature, calls **wind.m** to simulate the signal processing paths for the pairs of opposite receivers to calculate the **X** and **Y** speeds from the phase differences, and then call script **cordic.m** to convert the wind speed components to the wind speed modulus and direction.

5 – Design implementation
The system must be implemented as a clock synchronous digital system using a single clock of 2 MHz. The sampling rate of the acquired signals is 100 KHz and the same sampling rate must be used until the input of the module **phase2speed** (this means there are only 20 clock cycles to process one signal sample in each channel). After this module the sampling rate is reduced by a factor between 64 and 2048 (see section 5.4). The final output values for the wind speed and direction should be made available through the serial interface at the same sampling rate. The latency of the signal processing system can be as high as 10 ms or 20 k clock cycles.

The number of bits defined for each point in the datapath is represented in figure 2 and explained in detail in the following sections describing the functionality of each module.

Your role in this project is to design and validate the following key modules (the highlighted blocks in figure 2):

1. the datapath to calculate the phase difference between the signals acquired by opposite receivers and the conversion to wind speed along each axis;
2. the **windrec2pol** module to convert the X and Y wind speed components to the wind speed modulus and wind direction (this implements the CORDIC algorithm and is similar to the module to use in the phase calculator datapath);
3. the calibration modules (note that the same module will be used for wind speed and wind direction, although using different calibration tables).

The other modules will be provided either as their Verilog source code or as "black boxes" (i.e. only a compiled version will be available).

5.1 – Real to complex conversion (module real2cpx)
The two modules **real2cpx** create the complex representation of the audio signals acquired by the analog to digital converters. These circuits receive a stream of data samples (real signed 12-bit numbers) and calculate the imaginary part of the signal using a FIR Hilbert filter (the real part is the input signal delayed by a certain number of sampling periods – see below).

In the Matlab model **wind.m**, the filter coefficients are calculated by the Matlab function **designfilt( )**. The convolution between the input data vectors **rxuw** and **rxdw** and the filter impulse response is done with a "for" loop to help clarify the implementation of the convolution operation. The current implementation uses an 8th-order FIR filter whose coefficients are:

Coefficients	Matlab full precision	Quantized to 8 bits fractional
h_0	-0.000050047994083	0.00000000
h_1	-0.239316705733721	-0.23828125
h_2	0.000080979355495	0.00000000
h_3	-0.626102953444243	-0.62500000
h_4	0.000000000000000	0.00000000
h_5	0.626102953444243	0.62500000
h_6	-0.000080979355495	0.00000000
h_7	0.239316705733721	0.23828125
h_8	0.000050047994083	0.00000000

The imaginary part is the result of convolution and the real part is the original signal but delayed a number of samples equal to the group delay of the FIR filter (4 sampling periods in this case).

The function to be implemented by this module is described by the following pseudo-code (note that the multiplications by the coefficients equal to zero are not represented).

Being $\mathbf{x}_0$ the input sample received now and $\mathbf{x}_{-i}$ the input samples received **i** sampling periods before, the imaginary component is calculated as:

$$\mathbf{Imag}_0 = \mathbf{x}_{-1} * C_1 + \mathbf{x}_{-3} * C_3 + \mathbf{x}_{-5} * C_5 + \mathbf{x}_{-7} * C_7$$

The real component is the input sample received 4 sampling periods before:

$$\mathbf{Real}_0 = \mathbf{x}_{-4}$$

The following Matlab code (adapted from script **wind.m**) implements this procedure. Note this cannot be translated statement by statement to synthesisable Verilog code! Besides, this function should be completed in a few clock cycles (although it may implemented as a pipelined datapath).

```
% x( ) is the vector with the input samples
% h( ) is the vector with the Fir filter coefficients
% constant 8 is the filter order (impulse response has 9 coefficients)
% N is the length of input data vector
for i = 8+1 : N   % For each input sample, starting at sample 9
  imagp(i) = 0;   % initialize the imaginary part to zero
  for k = 0 : 8   % for all the filter coefficients
      imagp(i) = imagp(i) + x(i-k) * h(k+1); % multiply and accumulate
  end
  realp(i) = x(i-4); % The real part is the input delayed by 4 samples
end
```

The filter coefficients are quantized to 8 fractional bits and the input samples are represented as 12 bit two's complement integers. This module should produce the two outputs in the same scale as the inputs (12 bit integer signed).
As Verilog for RTL synthesis only allows signed integers, the procedure to implement the multiplications by fractional numbers is:

1. Represent the coefficients as integer signed numbers by multiplying the fractional coefficients by 28. This will yield h7' = - h1' = 61, h5' = - h3' = 160
2. Calculate the convolution operation using the integer coefficients. Note that the multiplication of a 12-bit integer by a 8-bit integer will generate a 20 bit result. The convolution operation sums four 20-bit numbers, but as the coefficients are known constants and their absolute sum does not exceed 2, the final result only needs to be represented with 21 bits maximum.
3. The 21-bits of the result represent 8 fractional bits and 13 integer bits. To reduce this result to the 13 integer bits, the 8-bit fractional part can be truncated (just discarding the fractional bits) or rounded to the nearest integer.

5.2 – Phase calculation (module phasecalc)
These modules should be an adaptation of the **rec2pol** module developed in the laboratory project 1 and re-used in project 2. If you were unable to simulate, synthesize and implement your **rec2pol** module during the laboratory project 2, use the reference module provided in that project. Some adaptations have to be done:

1. Only the angle output is necessary. This eliminates the final scaling operation of the X value;
2. The module should be able to calculate the angle in the whole domain [-180º, +180º]. The provided Matlab function **cordic_angle.m** includes this correction: first the initial value of **X** iteration variable must be the absolute value of the **X** input (the real part); then, after the calculation of the angle, the following correction must be applied (again, note this is not Verilog RTL code!)

```
% X0 is the input X or the real part, coming from module real2cpx
% angxy is the angle (in degrees) calculated by the CORDIC algorithm
% angxy360 is the angle output in the range [-180, +180]
if ( X0 < 0 )
    if ( Y0 < 0 )
        angxy360 = -180 - angxy;
    else
        angxy360 =  180 - angxy;
    end
end
```

3. The lookup-table used by the algorithm has now only 16 entries represented with 16 bits, where 6 bits represent the integer part and 10 bits the fractional part. Note that the values in the lookup table are the angles in degrees **arctan(2^{-i})**, for **i=0..15** and the maximum value in the table is **+45.0** . The lookup table contents are:

Address	Data in decimal (Matlab precision)	Data in hexadecimal (16 bits: 6 integer, 10 decimal)
0	45.0000000000	B400
1	26.5654296875	6A43
2	14.0361328125	3825
3	7.12500000000	1C80
4	3.57617187500	0E4E
5	1.79003906250	0729
6	0.89550781250	0395
7	0.44726562500	01CA
8	0.22363281250	00E5
9	0.11230468750	0073
10	0.05566406250	0039
11	0.02832031250	001D
12	0.01367187500	000E
13	0.00683593750	0007
14	0.00390625000	0004
15	0.00195312500	0002

4. Note that although the angle values stored in the table are positive numbers, the iterative process implemented by the CORDIC algorithm for variable **z** implements signed arithmetic using these values. Thus, the angle values read from the table must be extended with a leading bit equal to zero and then casted to signed data type:

```
$signed( { 1'b0, rom_data} )
```

5. The output angle is represented by 19 bits, with 9 bits for the integer part and 10 bits for the fractional part.

5.3 – Phase difference (module phasediff)

Module **phasediff** calculates the difference between the two angles generated by the **phasecalc** modules and convert that difference to the interval [-180º, +180º]. The Matlab code below describes the function to be implemented by this module:

```
% Calculate the phase difference
phasediff = phasedw – phaseuw;
% Convert the phase difference to the range [–180, +180]:
if ( phasediff > 180 )
  phasediff = phasediff – 360;
else
  if ( phasediff < –180 )
      phasediff = phasediff + 360;
  end
end
```

Again, this is not Verilog code and the direct translation to Verilog may not be the most efficient way to implement it. Note that all signals and arithmetic must be signed and the output of this module should be represented in 19 bits, 9 integer and 10 decimal.

5.4 – Phase to speed conversion (module phase2speed)

Module **phase2speed** converts the phase difference to the wind speed that originated that phase difference. Two of these modules calculate the **X** and **Y** component of the wind speed that are further converted to polar coordinates (wind speed modulus and the wind direction). The exact translation of the phase difference to wind speed requires solving a 2^{nd} order equation that also includes the sound propagation speed (which depends on the air temperature). To simplify the implementation, this module considers a fixed air temperature equal to 20ºC and uses a linear approximation that provides enough accuracy for the bit widths used along the datapath (less than 1% of error in speed for 20ºC and speed above 1m/s). The effect of temperature in the final wind speed is corrected later with module **tempcomp**.

This module should average the phase difference and then multiply it by the phase to speed conversion factor. The step to implement are:

1. Accumulate **N** successive phase differences. **N** is a configuration parameter input to this module and will have values equal to integer power of two, between 64 and 2048. The accumulator must be able to sum up to 2048 values of 19 bits, representing the phase differences. Thus, a minimum of 11+19 = 30 bits are necessary to implement this accumulator ($2048 = 2^{11}$). This module will generate the averaged phase difference at a rate equal to 100000/N (from 1562.5 to 48.8 samples per second).
2. After **N** sampling periods, divide the accumulator contents by **N**. As **N** is equal to $\mathbf{2^k}$, this division is just a right shift of **k** bits.
3. Multiply the averaged phase difference (19 bits) by the constant 20450 (this value is for the 15kHz input signal; for 17 kHz the constant is 18026 and for 19 kHz it is 16139). The result of this multiplication represents the wind speed with 17 bits for the fractional part and it must be rounded or truncated to 10 bits fractional. Six bits will be enough to represent the integer part because the wind speeds the system is able to measure are limited to the interval [-25m/s , +25 m/s]. Note that the phase difference is signed and the wind speed result will also be signed.

5.5 – Wind speed and wind direction from X and Y (module windrec2pol)

The final stage calculate the polar representation of the wind speed: wind speed modulus and wind direction. This is done with another variant of the **rec2pol** module, which is implemented in the Matlab model by the function **cordic.m**. Important details for this module are:

1. The inputs are the **X** and **Y** components of the wind speed coming from the **phase2wind** modules (16 bit signed words).

2. The speed output is represented in the same scale as the inputs (16 bits signed, 10 bit fractional) and the angle output is represented with 9 integer bits and 7 fractional bits.
3. The **arctan( )** lookup table is the same used for module **phasecalc** (see section 5.2).
4. The module should also work for the whole angle domain, generating results in the range [-180º, +180º].

5.6 – Temperature compensation (module tempcomp)

This module performs the compensation in the wind speed for the current air temperature. This operation is included in the Matlab function **windir.m** and will be provided as a IP core.

5.7 – Final calibration (modules calibwspd and calibwdir)

The calibration modules translate an M-bit input value to an N-bit output value, using a look-up table that represents a piece-wise linear function defined by 16 points and performing linear interpolation to calculate the values between the points stored in the table. The implementation must be parameterized with M and N and the values of these parameters should be between 10 and 20.

From the point of view of this module, the look-up table is accessed as a ROM (read-only memory) with 16 memory locations pre-loaded with the X and Y values that define the piece-wise linear calibration function. This memory may also contain any other data that can help simplify the logic complexity of the module. The circuit to design must be synchronous with the 2 MHz master clock and one calculation may take up to 64 clock cycles. The interface of the module should be:

```
module calibration
   #( parameter M=16, // default value for M
      parameter N=16) // default value for N
    (    input  clock,
         input  reset,
         input  start, // set to 1 to start a new conversion
         output ready, // set to 1 when ready
         input  signed [N-1 : 0 ] X,
         output signed [M-1 : 0 ] Y
    );

// The lookup table, 16 locations, X and Y pairs:
reg [ N + M - 1 : 0 ] LUTcalib[0:15];

// Load initial contents to the LUT from file "datafile.hex":
initial
begin
  $readmemh( "./data/datafile.hex", LUTcalib );
end

// your code here ...

endmodule
```

6 – Design verification

This section presents the verification kit provided to help building the testbenches for the individual modules and the complete system. This kit contains:

./matlab	Adapted versions of the Matlab scripts **wind.m** and **winddir.m** that generate hex files readable by Verilog testbenches;
./src/verilog-tb	Verilog testbench for the verification of the two datapaths that generate the X and Y speed components, including a basic functional model (detailed below);
./sim	Simulation project for Questasim, including a script for formatting the waveform window to present some signals in analog format;

6.1 – Verification flow

To execute a simulation with the current testbench and simulation model do the following steps:

1. Execute in Matlab the script (this is an adaptation of the previous **winddir.m**)
   `[Dir,Speed] = winddir_simgen( wangle, wspeed, T, Nmean );`
   Input parameters are:
   `wangle`: the true wind angle, in degrees (real number)
   `wspeed`: the true wind speed, in m/s (real number)
   `T`: the air temperature in ºC (for now use only 20)
   `Nmean`: the length of the phase averaging filter (32=2^6 to 2048=2^11)
   Return values are:
   `Dir`: the vector with the wind directions calculated (plotted in fig 21)
   `Speed`: the vector with the wind speeds calculated (plotted in fig 20)

   This script calls twice the script **wind_simgen.m** (an adaptation of the previous script **wind.m**) that implement the two signal processing paths to calculate the X and Y components of the wind speed. In both scripts the variable disp can be set to 1 (0) to enable (disable) plotting various signals along the datapath.

2. The script **wind_simgen.m** includes the addition of white noise and modulation of the signal amplitude to the received signals. To enable this, set to 1 the variables **addnoise** and **addAM** and adjust the respective parameters. Adding white noise with SNR less than 40 dB will degrade significantly the results and require a longer averaging filter (reducing the sample rate of the final speed calculation).

3. Running these scripts will create in folder **`../simdata`** a set of 21 text files with data in hexadecimal format, captured at the outputs of the main blocks in figure 2 during the Matlab simulation. The filenames are self-explanatory:
   `data_rx`$_i$: the input data at the outputs of module **rxreceiver**
   `real_rx`$_i$**`, imag_rx`**$_i$: the real and imaginary parts of the 4 input signals
   `phase_rx`$_i$: the instantaneous phase of the input signals
   `phasediff_X, phasediff_Y`: the phase differences along the X and Y axes
   `speed_X, speed_Y`: the X and Y components of the wind speed

4. Open QuestaSim and load the simulation project ./sim/winddir.mpf. This project uses two Verilog files:
   `./src/verilog-tb/winddir_tb.v`: the toplevel testbench (do not change any parameter unless you know exactly what you are doing!)
   `./src/verilog-tb/winddirectionXY.v`: a functional model of the signal processing datapath implementing the two calculators of the X and Y wind speed components. Inputs are the 4 **rx**$_i$ inputs and the outputs are the two wind speed X and Y components. The final conversion to polar coordinates is not yet included in this model. Note that this model is not synthesizable, does not intend to suggest any RTL solution for the various blocks and its purpose is only for demonstrating how to set up a simulation process that uses the data files generated by the execution of the Matlab scripts. Also, this model does not perform any processing with the input data and only generates the internal data read from the hex files created by Matlab, simulating a certain number of clock cycles used by each block to compute the corresponding internal output. The

waveform configuration script **./sim/wave_winddir.do** present the simulation waveforms of the various internal signals (execute in the QuestaSim command window **do wave_winddir.do**).

Note this is not a complete testbench for the whole system. However, you can use these models to help building your own verification processes, making use of the data generated by the Matlab scripts to compare to the outputs of your modules and automate the verification process. As discussed before, a visual analysis of waveforms can be a nice to conclude rapidly that the simulation results are wrong (particularly when formatted as analog signals), but this is not a practical neither efficient way to verify the correctness of the simulation results!