FP7- Grant Agreement no. 283393 – *RadioNet3*

Project name: Advanced Radio Astronomy in Europe

Funding scheme: Combination of CP & CSA

Start date: 01 January 2012 Duration: 48 month

Deliverable 8.6

Firmware design document: **Pulsar Binning**

Due date of deliverable:

Actual submission date: 2-08-2015

Deliverable Leading Partner: University of Manchester

Document information

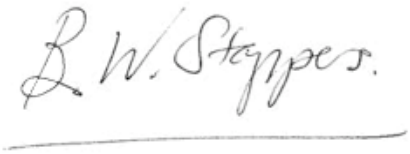
Document name:	Firmware design document: Pulsar Binning
Type	Design Document
WP	Microsoft Office Professional Plus 2010 V14.0.7145.5000 (32-bit)
Authors	Benjamin Stappers Prabu Thiagaraj Jayanta Roy Aziz Ahmedsaid **JBCA, University Of Manchester, Manchester, UK**

Dissemination Level

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

TABLE OF CONTENTS

DOCUMENT SIGNOFF



PREFACE

Purpose of this document

This document describes a generic Pulsar-binning design meant for the Uniboard^2. The details available in the document could be used to provide a guideline for the actual implementation. The document may be also useful for anyone involved in the design of the associated hardware/software systems or handling the data produced from the pulsar-binning module.

1 INTRODUCTION

1.1 Purpose

This document is a generic technical design as applicable to the Pulsar-binning design implementation in the Uniboard^2. The design described in the document is intended to assist the technical staff while implementing the design. The document may also be useful for anyone involved in the design of the associated hardware/software systems or handling the data produced from the pulsar-binning module.

1.2 Scope

The pulsar-binning module inputs are time-series data that is either from a filter-bank or correlator. The outputs from the module can be sent to a recorder or be sent to other real-time processing modules for subsequent analysis of the binned data streams.

1.3 Key words

Baseline, Correlator, dispersion measure (DM), Doppler correction, filter-bank, gating, binning, visibility

2. PULSAR-BINNING HARDWARE DESIGN

2.1 INTRODUCTION

The motivation is to enable a variety of studies on millisecond, and other interesting pulsars that are discovered recently. While the periodicity of known pulsars varies between milliseconds to a few seconds, the pulsed-emission is typically observed only over a small fraction of the pulsar's period. A pulsar-binning hardware produces binned and averaged profiles at periods of interest by processing the observed data in real-time. Pulsar-binning makes use of the precise knowledge of the pulsar phase and any modulation to its phase during the entire length of the observation. This technique can be used to produce so-called on-off images which separate the pulsed emission from the steady background, making localisation of the pulsar in a wide beam more simple and accurate (useful for efficient timing observations). It can also be useful when trying to study any extended emission associated with the pulsar, for example a pulsar wind nebula that may be "obscured" by the pulsed emission.

A critical task in this design is in providing a precise timing signal to the FPGA hardware. In order to appreciate the associated hardware complexity, a pilot study (see Appendix A) was carried out to implement the timing hardware in an FPGA. Based on the results of this implementation we were able to converge on the final design presented in this report.

2.2 FUNCTIONAL DESCRIPTION

In Fig.1, we show the pulsar-binning hardware as consisting of three functional blocks, viz., pre-processing, bin-processing and output-processing modules. The input to the hardware is a time domain data stream consisting of one or more frequency channels or visibilities (for brevity subsequent discussions we may use only channels) from a filter bank or correlator. The output from the pulsar-binning hardware is an averaged, over the desired integration time, profile across bins and different frequency channels.

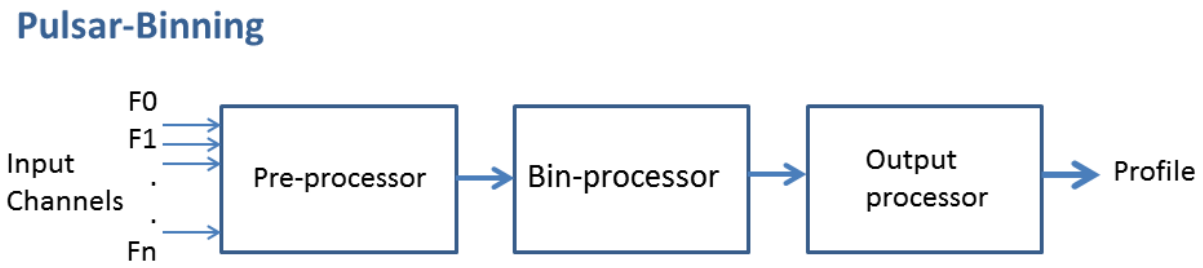


Figure 1. Pulsar-binning hardware.
Selected channels are pre-processed, binned and the averaged profile is sent out.

The pre-processor prepares the data stream for binning. The pre-processor, optionally,

- detects the signal
- pre-integrates the time samples
- combines all or some of the frequency channels and
- pass the pre-processed data to the processing section.

The bin-processor is the main functional block of the binning hardware. It makes a profile across the given period by integrating the time samples in bins. The bin for averaging is determined using the phase/bin relation supplied by the user. The averaging can be carried out over multiple periods. By updating phase/bin, periodically, the modulation from Doppler-shift (either due to the Earths motion or if the pulsar is in a binary) is corrected. Thus, the bin-processor,

- bins the data
- folds the binned data or visibility over the period of interest
- uses updated binning parameter for Doppler-shift correction
- passes the gated profile to the output-processor

The output processor can be programmed to output the integrated profile with the selected number of bins and specific frequency channels as input from the user parameter. Thus the output-processor,

- selects frequency channels
- selects bins and
- exports the profile

The main input parameter required to operate the pulsar-binning hardware is shown in Fig. 2

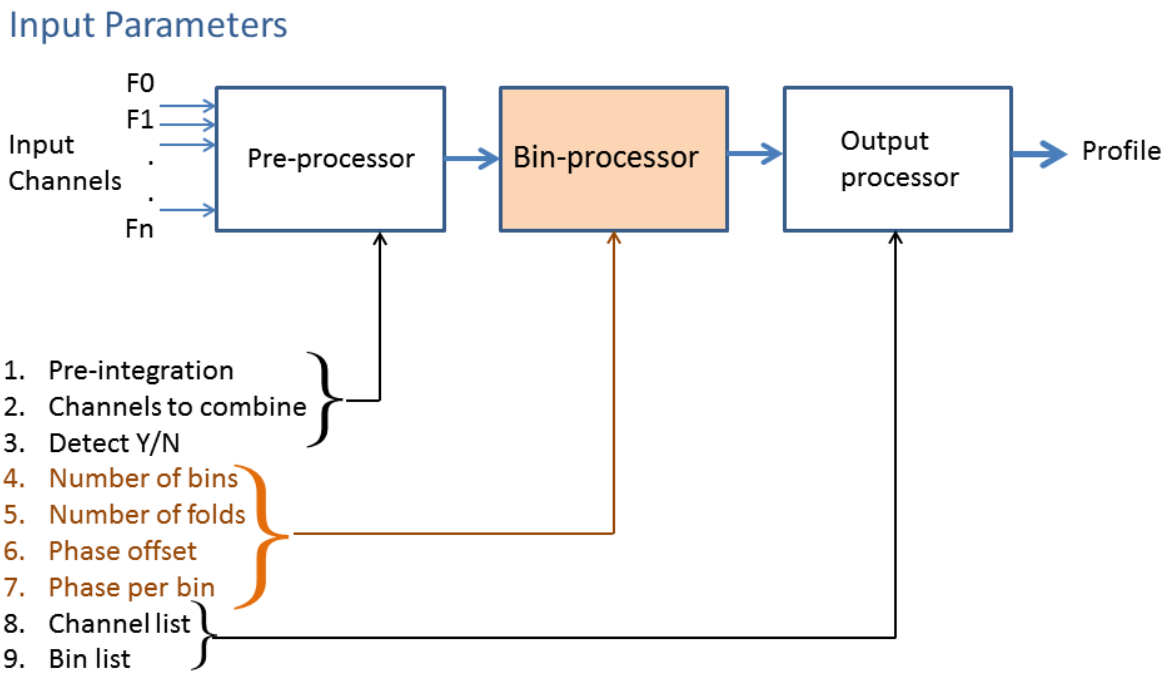


Figure 2. Input parameters

2.2.1 PRE-PROCESSOR

The pre-processor, as shown in Fig. 3, consists of a channel selector, detector, channel-combiner and pre-integrator modules. The role of the channel selector module is to select fixed set of channels from the inputs and pass it to the channel combiner module. This module is required to limit the number of maximum channels that would be fed into the binning hardware. The detector is used to produce a power stream if the input were in a voltage form (for e.g., from a filterbank). In the case of visibility data stream, the detector functionality is bypassed. The channel-combiner module (based on input parameter 2: channels to combine) groups and combines the adjacent frequency channels. Typically, sufficiently small channel widths for dedispersion, and sufficient channels for bandpass calibration are maintained. Then the data stream reach the pre-integrator module The pre integrator module averages the successive time samples (based on input parameter 1: Pre-integration) and passes the data to the bin-processor.

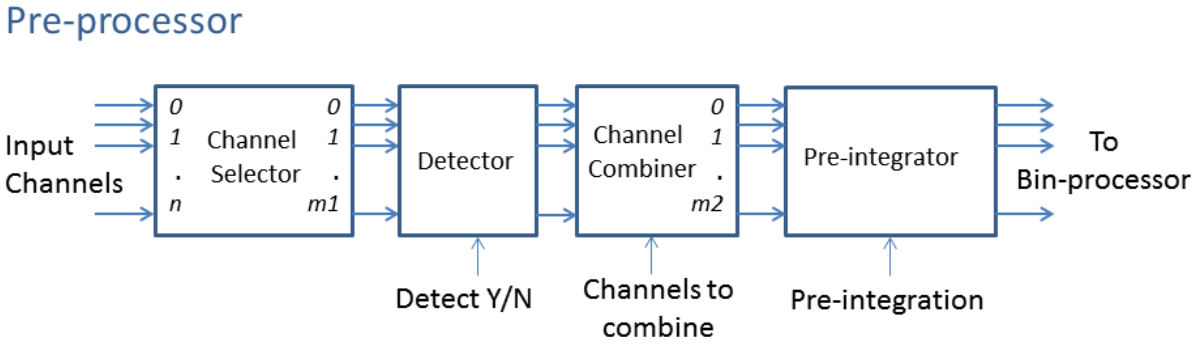


Figure 3. Pre-processor module

2.2.2 BIN-PROCESSOR

The pre-processed data stream arrives at the bin-processor. The bin processor, as shown in Fig. 4 A, consists of a large enough memory arrays (Bin-memory) to hold the binned profile across all

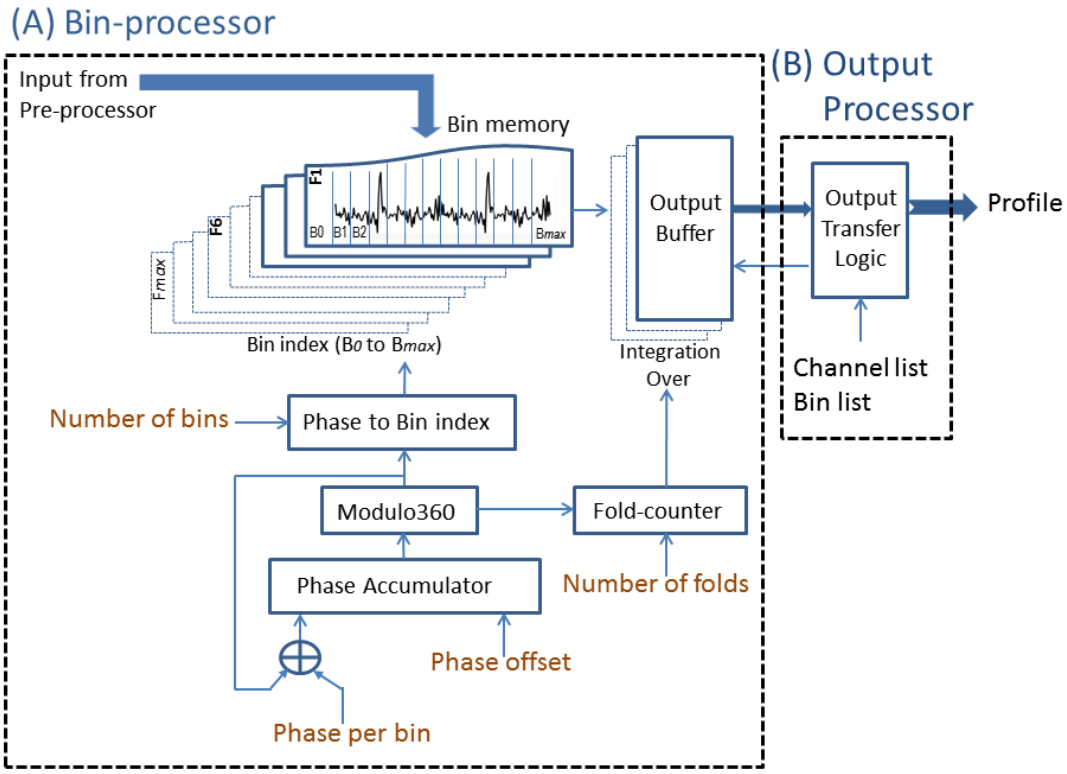


Figure 4. (A) Bin-processor and (B) Output processor

channels, and a phase computation section to decide the bin position for averaging. The phase computation section consists of a *phase-adder, phase accumulator, Modulo360, fold-counter, and phase-to-bin-index* modules. The *phase accumulator* is initialized with the phase offset supplied by the user. At the arrival of a new sample from the pre-integrator, the phase-per-bin value (supplied by the user) is added to the *phase-accumulator*. The *phase-accumulator* internally retains the phase value to a very large precision. The output from the *phase-accumulator* is passed through the *modulo360* module to represent phase within 0-360^{o} range. The phase value coming out of the *modulo360* module is fed to the phase-to-bin-index module, where the phase value is converted into a bin index. The bin-index represents the bin position where the incoming time sample is to be added for binning. The *modulo360* module also triggers the *fold-counter* to keep track of the number of completed period folds in the Bin-memory. The *fold-counter* triggers the *output-buffer* when the count value (completed folds) matches the *number-of-folds* parameter. When the *output-buffer* is triggered, the binned and averaged profile from the bin-memory is be written to the *output-buffer*.

The parameters required for phase computation (i.e. phase per bin, phase offset, number of bins) are fed from a external computer routine using the pulsar ephemerides and the timing package like TEMPO, which is described in section-2.3.

2.2.3 OUTPUT-PROCESSOR

The output-processor shown in Fig. 4 B, consists of an *output-transfer-logic*. This logic is triggered when the *output-buffer* gets a new set of binned profile. It operates by transferring only those channels and bins specified in the user input parameters: bin-list and channel list, for storage or processing by subsequent stages.

2.3 PHASE CALCULATION

This design assumes, a precession program based on pulsar timing package TEMPO running in a host computer calculating and supplying the phase-offset and phase-increment-per-bin values to the pulsar-binning hardware at the needed intervals. For a given pulsar ephemerides (obtained from ATNF pulsar catalog) and observing frequency, TEMPO can produce a set of polynomial coefficients to predict the pulse phase at any given time. The inputs to this phase predictor routine are pulsar ephemerides, observing frequency (v_{obs}), number of coefficient (N_{coeff}) to approximate the Taylor expansion of the phase variation of the pulsar, time span (t_{span}) for each polynomial series (solution) and total time observing span (t_{obs}). The polynomial solutions are written in a "polyco.dat" file, which lists reference phase (ϕ_0), reference rotational frequency (F_0) and values for coefficients (C_1, C_2, C_3 etc) as function of observing time.

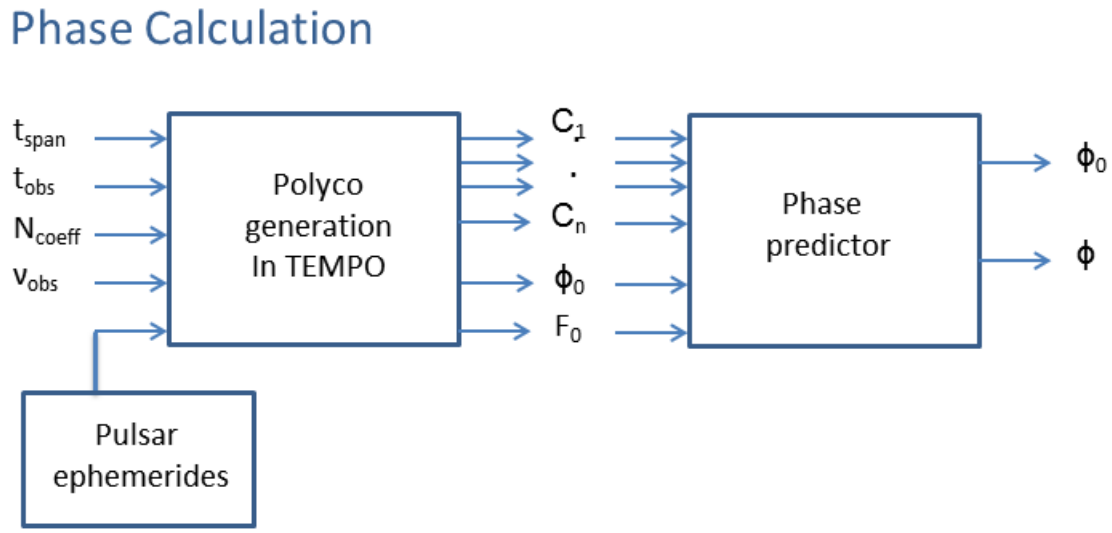


Figure 5. Phase parameter calculation using an external program

The pulse phase at a given time T, can be derived as,

$$\phi(t) = \phi_0(t_0) + (t-t_0) \times F_0 + C_1 + (t-t_0) \times C_2 + (t-t_0)^2 \times C_3 + \ldots\ldots$$

The reference phase (ϕ_0), phase per bin (ϕ/N_{bin}) and number of bins (N_{bin}) are provided as user inputs to the bin-processor.

In order to keep the period error < ± 0.5 phase bin, the user routine needs to supply N_{bin} based on minimisation of Modulo[P, N_{bin}].

3. IMPLEMENTATION

3.1 INTRODUCTION

This section discusses implementation of pulsar-binning in the Uniboard^2 and presents the FPGA resource usage for an example design approach. This study is made for an Arria-10 FPGA (10AX115R3F4012SGE2), and it identifies critical resources for pre-processor, bin-processor, and output processor modules.

3.2 RESOURCES FOR PRE-PROCESSOR

Resources required for implementing the pre-processor in FPGA is outlined in Fig. 6. For this study, an input bandwidth of 300 MHz, sample width of 16-bit (8,8 for real, imaginary) at each channel, and n-channels are assumed.

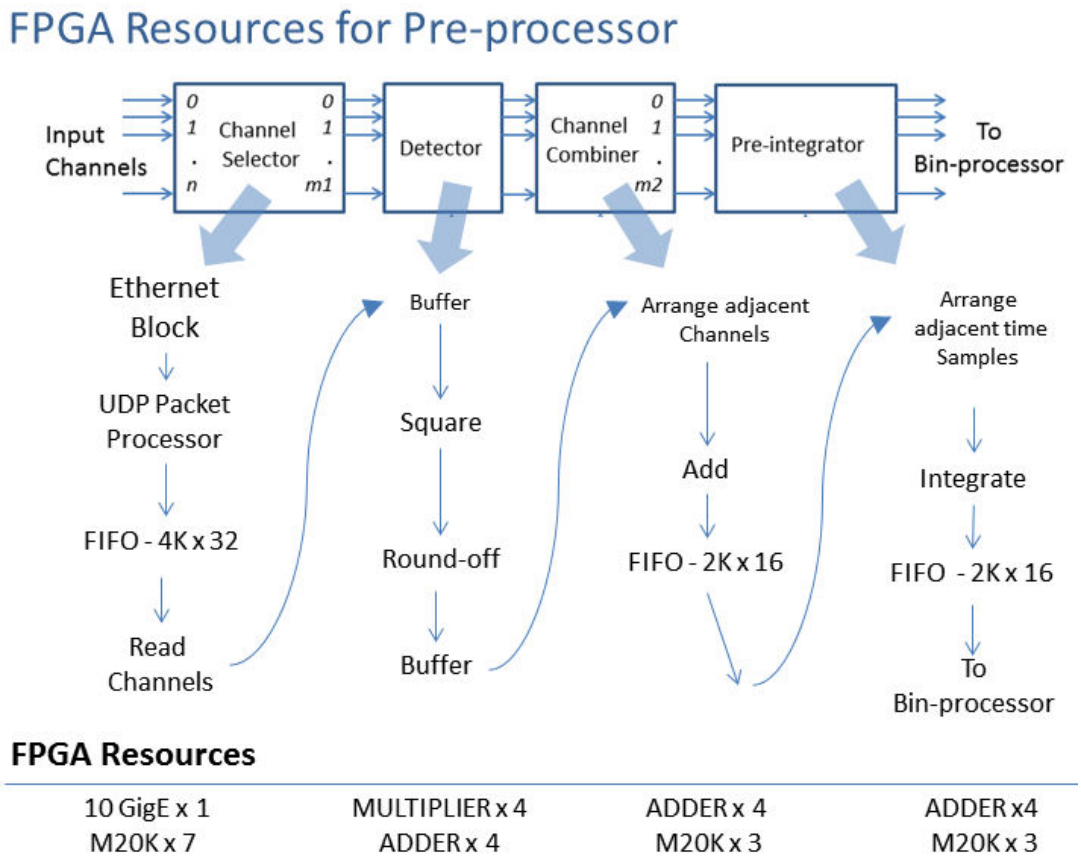


Figure 6. FPGA Resources for Pre-processor

The pre-processor section interfaces with a front-end external to the pulsar-binning hardware. The complexity associated with the data rate at the different submodules the pre-processor is shown in Fig. 7.

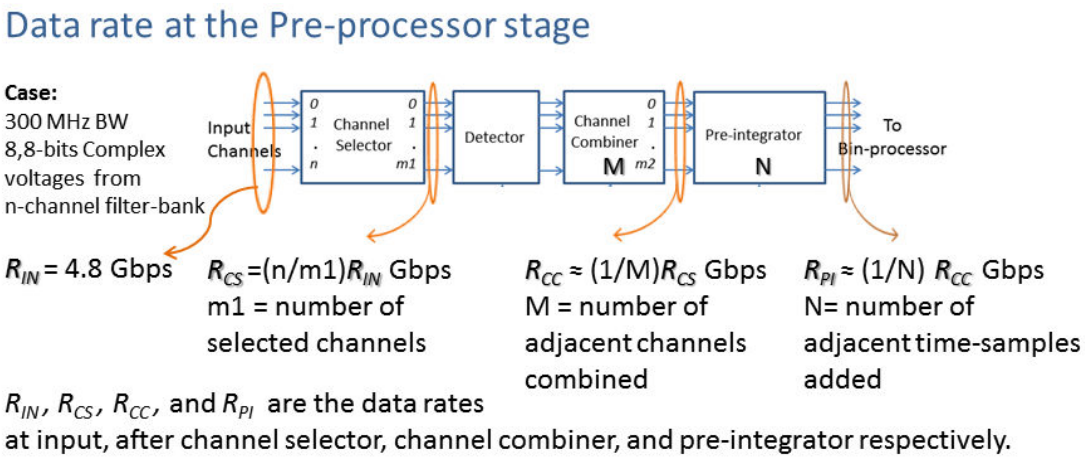


Figure 7. Data rate at the pre-processor stage

3.3 RESOURCES FOR BIN-PROCESSOR AND OUTPUT PROCESSOR

The Bin-processor moves the data across large memories. The complexity associated with size of the Memory buffers is shown in Fig. 8. The bin-memory is a dual port memory, and its size is decided by the number bins and channels. A data-word size of 32 bits, allows an 8-bit inputs to be integrated (due to samples/bin x number of folds) 2^{24} times. While, an implementation as shown in Fig. 8 where 512 bins across 512 channels uses about 20% of available memory, it is possible to other implementations in our FPGA with twice as many bins and channels (1024, 1024). Retaining the fold counts to at least 20 bits allows to track up to million folds. Similarly the phase accumulator being 128-bits wide allows to retain 2.4 times better resolution compared to the only the mantissa field of the IEEE 754 double float. The phase-per-pin parameter is typically supplied as list containing updates spanning the observation period. The phase update logic reads this list and effects updates at the arrival of specified time sample in the bin-processor. This function requires a logic similar to the phase accumulator, but attached to the phase-per-bin input path.

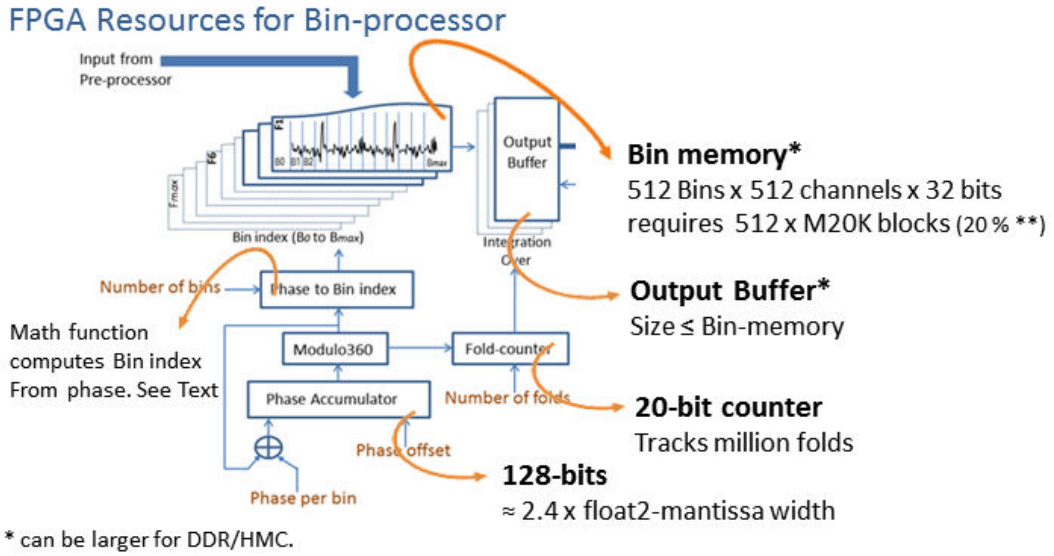


Figure 8. FPGA Resources for Bin-processor

3.4 **RESOURCES FOR OUTPUT-PROCESSOR**

The output-processor could be implemented in a state-machine logic to transfer the integrated profile, consisting of selected bins and channels. Assuming a data-rate reduction of about 10-times, due to various processing, in the pulsar-binning, a 1-Gpps capable Ethernet port would be required to transfer the integrated profile.

--

4. DISCUSSION

This Pulsar-binning hardware design can be implemented in Uniboard FPGAs using a HDL or High-level FPGA design tools. The basic design described here can be scaled to handle signals from larger bandwidths and channels. In case of operating with correlator with size of $N_{baseliens}$ (including two cross-polar vectors in case of full-polar implementation of correlator), there will be $N_{baselines}$ times independent implementation of the pulsar-binning hardware described here. The Uniboard architecture specifically suits for such an approach, where multiple instantiation within Arria-10 FPGA (or footprint compatible Stratix-10 in future) or across multiple FPGAs in the board(s) is viable. Also, as it has been highlighted in Fig. 8, the external memory available in the Uniboard^2 allows to make an implementation for pulsar-binning with large enough bins and frequency channels suitable for millisecond pulsar studies and for many baseline inputs. Having a fine resolution for bin period and frequency allows studying millisecond pulsars occurring with high dispersion measures (DM).

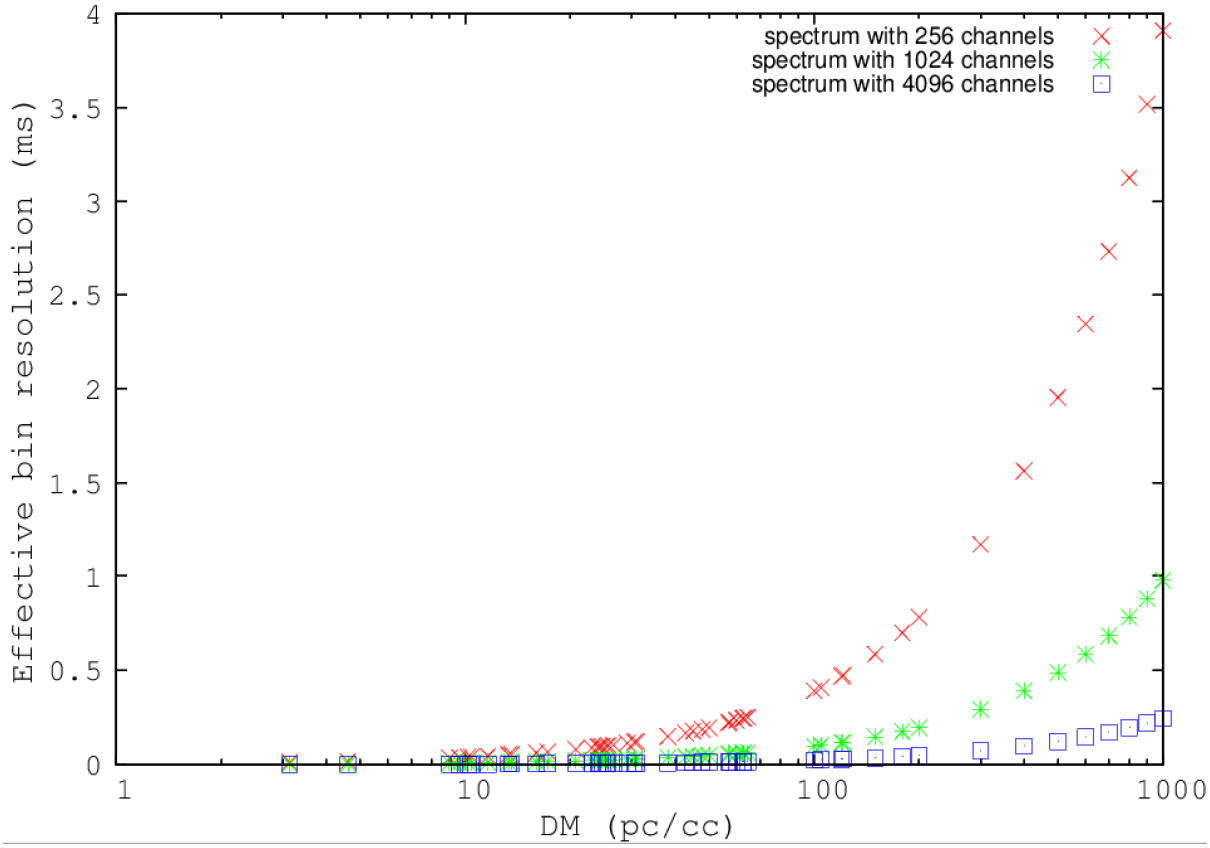


Figure 9 Optimal Choice for bin resolution

The plot in Fig 9 shows the optimal choice of bin resolution as function of DM for three selections of number of spectral channels across the full band of 300 MHz centred at an observing frequency of 1.3 GHz. The minimum bin resolution equals to the temporal broadening due to dispersion smearing of the pulsed signal within a given channel. For 4096 spectral channel mode, the bin resolution is around 250 µs at DM of 1000 pc/cc, which gives minimum required bin for effective binning of a

millisecond pulsar. Whereas for spectrum with coarser channels (e.g. 256), the allowed DM range for binning of millisecond pulsar reduces to about 50 pc/cc

As a possible future scope, the design can be extended to include an onboard processor in the FPGA to compute the phase computation. Also a Matlab functional model of the pulsar-binning design could be developed to aid the development and testing of the hardware.

While there is further scopes to extended the basic design described here, the document covers all core functionalities required for pulsar binning and also shows how it can be implemented by using Uniboard^2 architecture.

References:

1. http://www.atnf.csiro.au/people/pulsar/psrcat/
2. http://www.atnf.csiro.au/research/pulsar/tempo/
3. B. W. Stappers, B. M. Gaensler, S. Johnston, A deep search for pulsar wind nebulae using pulsar gating,1999
4. Jayanta Roy, Bhaswati Bhattacharyya, Coherently dedispersed gated imaging of millisecond pulsars, 2013

Appendix A

Source codes

1. Polynomial (MATLAB codes).

```
%-------------------------------------------------------------------------------------------
%% ComputePolynomial:

% Computes the polynomial coefficients used for the computation of the dedispersion
coefficients.

%-------------------------------------------------------------------------------------------

% Designed and implemented by Dr. Aziz AhmedSaid, University of Manchester, UK.

% Email: aziz.ahmedsaid@manchester.ac.uk or a.ahmedsaid@qub.ac.uk

% Created in: August 2012

% Last modification: August 2012

%-------------------------------------------------------------------------------------------

% THIS MATERIAL (DOCUMENTS AND/OR SOURCE CODE) IS CONFIDENTIAL.

% WE REMIND OUR COLLEAGUES THAT MEMBERS ON THE PROJECT ARE BOUND BY THE
CONFIDENTIALITY CLAUSE IN THE

% 'FP7 GRANT AGREEMENT - ANNEX II - GENERAL CONDITIONS' (SECTION 3 II.9) AND CAN'T
DISCLOSE THE

% CONTENT TO ANY THIRD PARTY WITHOUT OUR PERMISSION.

% THE UNIVERSITY OF MANCHESTER IS CURRENTLY IN THE PROCESS OF ASSESSING COMMERCIAL
INTEREST IN THE

% DEVELOPMENT AND MAY WISH TO PROTECT IT IF THAT'S THE CASE.

% THE DECISION ABOUT HOW THE SOURCE CODE WILL BE MADE AVAILABLE AND THE LICENSING
TERMS WILL BE MADE

% ONCE THE ASSESSMENT IS COMPLETE.

% THIS MATERIAL IS PROVIDED WITH NO WARRANTY WHATSOEVER.

%-------------------------------------------------------------------------------------------
function
[Coefficients,ProductsMSP]=ComputePolynomial(POLYNOMIAL_ORDER,COEFFICIENTS_WIDTH,PR
ODUCTS_WIDTH,INPUT_WIDTH,OUTPUT_WIDTH,Input)


% Parameters

K=POLYNOMIAL_ORDER;% polynomial order

NC=COEFFICIENTS_WIDTH-1;% number of bits to represent the coefficients (no more
than 32 bits).

Nf=OUTPUT_WIDTH-1;% number of bits to represent the fractional part of the result.
```

```
Np=PRODUCTS_WIDTH-1;% Number of bits to represent the products.

Nx=INPUT_WIDTH;% Number of bits to represent the frequencies


T=Input;


% polynomial computation

N=length(T);

d=max(round(N/1000),1);% Data sampling step.

p=polyfit(0:d:N-1,T(1:d:N),K);


pf=p-fix(p);% Only the fractional parts are relevant

c=[pf(1:K-1)./pf(2:K),pf(K),pf(K+1)];


pscale=2^NC./2.^ceil(log2(abs(c(1:K))));


c(1:K)=c(1:K).*pscale;% Coefficients 1:K represented using NC bits


c(K+1)=c(K+1)*2^Nf;% Coefficient c(K+1) represented using Nf bits


Coefficients=round(c);


sp=zeros(1,K);% product of coefficients scales needed to sum the products.


sp(1)=pscale(K)*2^(Np-NC-Nx);

for n=2:K,

    sp(n)=sp(n-1)*pscale(K-n+1)*2^(-NC-Nx);

end


ProductsMSP=Np+log2(2^Nf./sp);



%-------------------------------------------------------------------------------------------------
%% DedisPolynomialGen:
% Generates the polynomial coefficients used for the computation of the
dedispersion coefficients.

%-------------------------------------------------------------------------------------------------
% Designed and implemented by Dr. Aziz AhmedSaid, University of Manchester, UK.

% Email: aziz.ahmedsaid@manchester.ac.uk or a.ahmedsaid@qub.ac.uk

% Created in: September 2011

% Last modification: April 2013
```

```
%-------------------------------------------------------------------------------------------------
% THIS MATERIAL (DOCUMENTS AND/OR SOURCE CODE) IS CONFIDENTIAL.

% WE REMIND OUR COLLEAGUES THAT MEMBERS ON THE PROJECT ARE BOUND BY THE
CONFIDENTIALITY CLAUSE IN THE

% 'FP7 GRANT AGREEMENT - ANNEX II - GENERAL CONDITIONS' (SECTION 3 II.9) AND CAN'T
DISCLOSE THE

% CONTENT TO ANY THIRD PARTY WITHOUT OUR PERMISSION.

% THE UNIVERSITY OF MANCHESTER IS CURRENTLY IN THE PROCESS OF ASSESSING COMMERCIAL
INTEREST IN THE

% DEVELOPMENT AND MAY WISH TO PROTECT IT IF THAT'S THE CASE.

% THE DECISION ABOUT HOW THE SOURCE CODE WILL BE MADE AVAILABLE AND THE LICENSING
TERMS WILL BE MADE

% ONCE THE ASSESSMENT IS COMPLETE.

% THIS MATERIAL IS PROVIDED WITH NO WARRANTY WHATSOEVER.

%-------------------------------------------------------------------------------------------------
%% Program start

% Clean the work space:

clc;

clear all;

close all;


%% Parameters


F0=1340e6;% The carrier/center frequency

B=32e6;% Bandwidth.

Fs=2*B;% The sampling frequency (of the baseband signal).

DM=56.7949;

D=DM*4.16e+015;% The dispersion constant


K=3;% polynomial order

Numpols=8;% NUMBER of POLYNOMIALS

MAXX=18;% Max hardware multiplier width.

NC=32;% number of bits to represent the coefficients (no more than 32 bits).

Nf=14;% number of bits to represent the fractional part of the result.

Ntv=2^4;% This is the number of test samples (used for simulation verification). 0
= no test vectors.

UseOptimaleNp=0;

if UseOptimaleNp==0,
```

```
    Np=2*MAXX-2;% Number of bits to represent the products.
end

filename='polynomial_pkg.vhd';% Polynomial module package file name.
%% Coherent dedispersion

% Dedispersion window size determination
Tmax=D*(1./(F0-B/2)^2-1/(F0+B/2)^2);

if Tmax==0
    Tmax=1;
end

n=ceil(log2(Tmax*Fs))+1;

Nd=2^n;% Number of non-overlapping samples per dedispersion window.

f=(0:Nd-1)*Fs/(2*Nd)+F0-B/2;% F0-B/2<f<F0+B/2
T=D*(1./f-f./(F0+B/2)^2);%
clear f;

% polynomials computation
PLNM=zeros(Numpols,K+1);
PMSP=zeros(Numpols,K);
N=length(T);
Nx=round(log2(N));% Number of bits to represent the frequencies
NC=NC-1;
Nf=Nf-1;
Np=Np-1;

B=round(N/Numpols);

for p=1:Numpols,

    xt=(p-1)*B+1:min(p*B,N);
    xi=0:length(xt)-1;
```

```
    [PLNM(p,:),PMSP(p,:)]=ComputePolynomial(K,NC+1,Np+1,Nx,Nf+1,T(xt));


    log2(abs(PLNM(p,:)))

    % Time delays computation

    % I=polyval(p,0:N-1);

    %

    % fprintf(1,'\nmax(abs(T-I)) = %e\n',max(abs(T-I)));

    %

    % clear I;

    I2=PolynomialModuleModel(NC,Np,Nx,Nf,MAXX,PMSP(p,:),PLNM(p,:),xi);

    Tf=(T(xt)-fix(T(xt)));

    %T=2^Nf*T;%

    I2f=(I2/2^Nf-fix(I2/2^Nf));

    figure;

    plot(sin(2*pi*Tf)-sin(2*pi*I2f),'.');

    %fprintf(1,'Scale powers =\n');fprintf(1,'%d\n',log2(2^Nf./sp));

    fprintf(1,'Number of MSBits =\n');fprintf(1,'%d\n',PMSP(p,:));

    fprintf(1,'\nThe coefficients =\n(');fprintf(1,'%5.0f,',PLNM(p,K+1:-
1:1));fprintf(1,');\n');

end

if (Ntv ~=0)


    TVi=zeros(Numpols,Ntv);

    TVo=zeros(Numpols,Ntv);


    for p=1:Numpols,


        TVi(p,:)=[0,1+round((N-3)*rand(1,Ntv-2)),N-1];
```

```
TVo(p,:)=PolynomialModuleModel(NC,Np,Nx,Nf,MAXX,PMSP(p,:),PLNM(p,:),TVi(p,:));


    end


else

    TVi=[];

    TVo=[];

end


%% Generate Coefficients vhdl package:

msc=max(max(PMSP));

if (msc > Np)

    fprintf(1,'\nERROR: The products require at least %d bits ...\n',msc+1);

    fprintf(1,'Options:\n\tReduce the results fractional part width (Nf) to
%d\n\tIncrease the products width (Np) to %d.',Nf-(msc-Np)+1,msc+1);

%     fprintf(1,'\n\tIncrease the polynomial order (K)\n\tIncrease the number of
polynomials (Numpols)');

%     fprintf(1,'\n\tOr any combination of the above.\n');

else

    if (Nf < (Nf-(msc-Np)))

        fprintf(1,'\nOptimal results fractional part width (Nf) is %d (can be
achieved at almost no cost).\n',Nf-(msc-Np)+1);

    end


GeneratePolynomialVHDLPackage(filename,NC+1,Np+1,Nx+1,Nf+1,PMSP+1,PLNM,TVi,TVo);

end


beep


close all


%-------------------------------------------------------------------------------------------
% GeneratePolynomialVHDLPackage: This function automatically generates a package
containing

% definitions, test inputs and outputs for the Polynomial module v2.

%-------------------------------------------------------------------------------------------

% Designed and implemented by Dr. Aziz AhmedSaid, University of Manchester, UK.
```

```
% Email: aziz.ahmedsaid@manchester.ac.uk or a.ahmedsaid@qub.ac.uk

% Created in: November 2011

% Last modification: October 2012

%-------------------------------------------------------------------------------------------
% THIS MATERIAL (DOCUMENTS AND/OR SOURCE CODE) IS CONFIDENTIAL.

% WE REMIND OUR COLLEAGUES THAT MEMBERS ON THE PROJECT ARE BOUND BY THE
CONFIDENTIALITY CLAUSE IN THE

% 'FP7 GRANT AGREEMENT - ANNEX II - GENERAL CONDITIONS' (SECTION 3 II.9) AND CAN'T
DISCLOSE THE

% CONTENT TO ANY THIRD PARTY WITHOUT OUR PERMISSION.

% THE UNIVERSITY OF MANCHESTER IS CURRENTLY IN THE PROCESS OF ASSESSING COMMERCIAL
INTEREST IN THE

% DEVELOPMENT AND MAY WISH TO PROTECT IT IF THAT'S THE CASE.

% THE DECISION ABOUT HOW THE SOURCE CODE WILL BE MADE AVAILABLE AND THE LICENSING
TERMS WILL BE MADE

% ONCE THE ASSESSMENT IS COMPLETE.

% THIS MATERIAL IS PROVIDED WITH NO WARRANTY WHATSOEVER.

%-------------------------------------------------------------------------------------------
-------------------

function
GeneratePolynomialVHDLPackage(filename,COEFFICIENTS_WIDTH,PRODUCTS_WIDTH,INPUT_WIDT
H,OUTPUT_WIDTH,ProductsMSP,Coefficients,TestInput,TestOutput)


[NUMBER_POLYNOMIALS,POLYNOMIAL_ORDER]=size(Coefficients);

POLYNOMIAL_ORDER=POLYNOMIAL_ORDER-1;

TEST_VECTOR_SIZE=length(TestInput);

NumValsperLine=16;


fid = fopen(filename, 'w');


fprintf(fid,'-------------------------------------------------------------------------
---------------------------------\n');

fprintf(fid,'-- Automatically generated package containing definitions, test inputs
and outputs for the Polynomial module v2.\n');

fprintf(fid,'-- This file has been generated using the MATLAB function
GeneratePolynomialVHDLPackage (%s).\n',date);

fprintf(fid,'-------------------------------------------------------------------------
---------------------------------\n');

fprintf(fid,'-- Designed and implemented by Dr. Aziz AhmedSaid, University of
Manchester, UK.\n');
```

```
fprintf(fid,'-- Email: aziz.ahmedsaid@manchester.ac.uk or
a.ahmedsaid@qub.ac.uk\n');

fprintf(fid,'-- Created in: November 2011\n');

fprintf(fid,'-- Last modification: April 2013\n');

fprintf(fid,'-------------------------------------------------------------------------
---------------------------------\n');

fprintf(fid,'-- THIS MATERIAL (DOCUMENTS AND/OR SOURCE CODE) IS CONFIDENTIAL.\n');

fprintf(fid,'-- WE REMIND OUR COLLEAGUES THAT MEMBERS ON THE PROJECT ARE BOUND BY
THE CONFIDENTIALITY CLAUSE IN THE\n');

fprintf(fid,'-- "FP7 GRANT AGREEMENT - ANNEX II - GENERAL CONDITIONS" (SECTION 3
II.9) AND CANNOT DISCLOSE THE\n');

fprintf(fid,'-- CONTENT TO ANY THIRD PARTY WITHOUT OUR PERMISSION.\n');

fprintf(fid,'-- THE UNIVERSITY OF MANCHESTER IS CURRENTLY IN THE PROCESS OF
ASSESSING COMMERCIAL INTEREST IN THE\n');

fprintf(fid,'-- DEVELOPMENT AND MAY WISH TO PROTECT IT IF THAT IS THE CASE.\n');

fprintf(fid,'-- THE DECISION ABOUT HOW THE SOURCE CODE WILL BE MADE AVAILABLE AND
THE LICENSING TERMS WILL BE MADE\n');

fprintf(fid,'-- ONCE THE ASSESSMENT IS COMPLETE.\n');

fprintf(fid,'-- THIS MATERIAL IS PROVIDED WITH NO WARRANTY WHATSOEVER.\n');

fprintf(fid,'-------------------------------------------------------------------------
---------------------------------\n');


fprintf(fid,'library ieee;\nuse ieee.std_logic_1164.all;\nuse
ieee.numeric_std.all;\nuse ieee.math_real.all;\n');

fprintf(fid,'\npackage polynomial_pkg is\n\n');

if (COEFFICIENTS_WIDTH>32)

    fprintf(fid,'ERROR : COEFFICIENTS_WIDTH > 32, Integers wider then 32 bits are
not supported in VHDL\n');

else

    fprintf(fid,'    constant POLYNOMIAL_ORDER   : natural := %d;-- Polynomial
order or the number of coefficients-1.\n',POLYNOMIAL_ORDER);

    fprintf(fid,'    constant NUMBER_POLYNOMIALS : natural := %d;-- Number of
polynomes, i.e number of coefficient sets.\n',NUMBER_POLYNOMIALS);

    fprintf(fid,'    constant COEFFICIENTS_WIDTH : natural := %d;-- MAX_X_WIDTH <
COEFFICIENTS_WIDTH <= 2 x MAX_X_WIDTH\n',COEFFICIENTS_WIDTH);

    fprintf(fid,'    constant PRODUCTS_WIDTH     : natural := %d;-- MAX_X_WIDTH <
PRODUCTS_WIDTH <= 2 x MAX_X_WIDTH\n',PRODUCTS_WIDTH);

    fprintf(fid,'    constant INPUT_WIDTH        : natural := %d;-- MAX_X_WIDTH <
INPUT_WIDTH <= 2 x MAX_X_WIDTH\n',INPUT_WIDTH);

    fprintf(fid,'    constant OUTPUT_WIDTH       : natural := %d;-- This is just
the fractional part.\n',OUTPUT_WIDTH);
```

```
    fprintf(fid,'    constant TEST_VECTOR_SIZE   : natural := %d;-- This is the
number of test samples (used for simulation/verification).\n',TEST_VECTOR_SIZE);


    % fprintf(fid,'    constant MSP                : natural := %d;-- Width of the
most significant part for product 2 to POLYNOMIAL_ORDER (corresponding to x^2). The
LSP will be ignored.\n',MSP);

    % fprintf(fid,'    constant DMSP               : natural :=  %d;-- Difference
in number of bits between two consecutive products MSP.\n',DMSP);



    fprintf(fid,'\n    -- The polynomial module is intended for polynomials with
large dynamic range between the biggest and the smallest');

    fprintf(fid,'\n    -- coefficient in absolute value. An example of such
polynomials are the polynomials used for pulsar timing.');

    fprintf(fid,'\n    -- The computations are performed in fixed point arithmetic.
The input polynomial P must be in the following format:');

    fprintf(fid,'\n    -- P'' = {c(0),c(1),c(2)/c(1),c(3)/c(2),...,c(n)/c(n-1)}
were {c(0),c(1),c(2),c(3),...,c(n)} are the coefficient of P');

    fprintf(fid,'\n    -- The polynomial evaluation is performed as follows:');

    fprintf(fid,'\n    -- 1. The input x is multiplied by coefficient c(1)
(corresponding to x power 1);');

    fprintf(fid,'\n    -- 2. The product c(1)x is multiplied by c(2)/c(1)x yielding
c(2)x^2;');

    fprintf(fid,'\n    -- 3. And so on, product c(i)x^i is multiplied by
c(i+1)/c(i)x yielding c(i+1)x^2;');

    fprintf(fid,'\n    -- 4. The products will be scaled as follows: Frac(C0) ->
2^OUTPUT_WIDTH , P(i) -> 2^(PRODUCTS_WIDTH-product_msp(i));');

    fprintf(fid,'\n    -- 5. Finally, all the products are summed together in
addition to c(0) (using the appropriate scale);');

    fprintf(fid,'\n    -- 6. The output is the fractional part of the result,
because in the intended applications the integer part is irrelevant;');

    fprintf(fid,'\n    -- Users can check that this implementation is suitable for
their application if the coefficients {c(2)/c(1),c(3)/c(2),...,c(n)/c(n-1)} have a
small dynamic range relative to {c(2),c(3),...,c(n)}.');



    %-------------------------------------------------------------------------

%     fprintf(fid,'\n\n    type natural_vector is array(1 to POLYNOMIAL_ORDER) of
natural;');

%

%     fprintf(fid,'\n\n    constant product_msp : natural_vector :=
(');fprintf(fid,'%d,',ProductsMSP(1:POLYNOMIAL_ORDER-
1));fprintf(fid,'%d);',ProductsMSP(POLYNOMIAL_ORDER));
```

```
%     fprintf(fid,'-- This is the width of the most significant part (msp) of a
product, i.e, msp = product''high downto product''high-product_msp+1');

    %---------------------------------

    fprintf(fid,'\n\n    type msp_indices_array is array(0 to NUMBER_POLYNOMIALS-
1,1 to POLYNOMIAL_ORDER) of natural;\n');

    fprintf(fid,'\n    -- A product_msp gives the width of the most significant
part (msp) of a product, i.e, msp = product''high downto product''high-
product_msp+1.');

    fprintf(fid,'\n    constant product_msp_array : msp_indices_array :=\n
(\n');

    for n=1:NUMBER_POLYNOMIALS-1,

        fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',ProductsMSP(n,1:POLYNOMIAL_ORDER-1));fprintf(fid,'%d), -- For
Polynomial %d\n',ProductsMSP(n,POLYNOMIAL_ORDER),n);

    end

    n=NUMBER_POLYNOMIALS;fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',ProductsMSP(n,1:POLYNOMIAL_ORDER-1));fprintf(fid,'%d)  -- For
Polynomial %d\n',ProductsMSP(n,POLYNOMIAL_ORDER),n);

    fprintf(fid,'    );\n');



    %-----------------------------------------------------------------------

    fprintf(fid,'\n\n    type integer_coefficients_array is array(0 to
NUMBER_POLYNOMIALS-1,0 to POLYNOMIAL_ORDER) of integer;\n');

    % fprintf(fid,'\n    type coefficients is array(POLYNOMIAL_ORDER downto 0) of
std_logic_vector(COEFFICIENTS_WIDTH-1 downto 0);\n');

    % fprintf(fid,'\n    type coefficients_array is array(NUMBER_POLYNOMIALS-1
downto 0) of coefficients;\n');



    c=Coefficients;

    Numpols=NUMBER_POLYNOMIALS;

    K=POLYNOMIAL_ORDER;

    fprintf(fid,'\n    constant integer_polynomial_coefficients_array :
integer_coefficients_array :=\n    (');

    fprintf(fid,'\n        -- Coefficients  = Frac( C0 , C1 , C2/C1 , C3/C2 , ... ,
Cn+1/Cn , ... )\n');

    for n=1:Numpols-1,

        fprintf(fid,'        %d => (',n-1);fprintf(fid,'%d,',c(n,K+1:-
1:2));fprintf(fid,'%d), -- Polynomial %d\n',c(n,1),n);

    end

    n=Numpols;fprintf(fid,'        %d => (',n-1);fprintf(fid,'%d,',c(n,K+1:-
1:2));fprintf(fid,'%d)  -- Polynomial %d\n',c(n,1),n);
```

```
    fprintf(fid,'    );\n');


    if(TEST_VECTOR_SIZE ~= 0)

        fprintf(fid,'\n    type test_vector is array(0 to NUMBER_POLYNOMIALS-1,0 to
TEST_VECTOR_SIZE-1) of integer;\n');


%         fprintf(fid,'\n    constant test_input : test_vector := -- This is the
test input (used for simulation/verification).\n    (');

%         for n=1:NumValsperLine:TEST_VECTOR_SIZE,

%             fprintf(fid,'\n        ');

%             fprintf(fid,'%d,',TestInput(n:min(n+NumValsperLine-
1,TEST_VECTOR_SIZE-1)));

%         end

%         fprintf(fid,'%d\n    );',TestInput(TEST_VECTOR_SIZE));



        fprintf(fid,'\n    constant test_input : test_vector := -- This is the test
input (used for simulation/verification).\n    (\n');

        for n=1:Numpols-1,

            fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',TestInput(n,1:TEST_VECTOR_SIZE-1));fprintf(fid,'%d), -- For
Polynomial %d\n',TestInput(n,TEST_VECTOR_SIZE),n);

        end

        n=Numpols;fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',TestInput(n,1:TEST_VECTOR_SIZE-1));fprintf(fid,'%d)  -- For
Polynomial %d\n',TestInput(n,TEST_VECTOR_SIZE),n);

        fprintf(fid,'    );\n');



%         fprintf(fid,'\n\n    constant test_output : test_vector := -- This is the
expected test output (used for simulation/verification).\n    (');

%         for n=1:NumValsperLine:TEST_VECTOR_SIZE,

%             fprintf(fid,'\n        ');

%             fprintf(fid,'%d,',TestOutput(n:min(n+NumValsperLine-
1,TEST_VECTOR_SIZE-1)));

%         end

%         fprintf(fid,'%d\n    );',TestOutput(TEST_VECTOR_SIZE));


       fprintf(fid,'\n\n    constant test_output : test_vector := -- This is the
expected test output (used for simulation/verification).\n    (\n');
```

```
        for n=1:Numpols-1,

            fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',TestOutput(n,1:TEST_VECTOR_SIZE-1));fprintf(fid,'%d), -- For
Polynomial %d\n',TestOutput(n,TEST_VECTOR_SIZE),n);

        end

        n=Numpols;fprintf(fid,'        %d => (',n-
1);fprintf(fid,'%d,',TestOutput(n,1:TEST_VECTOR_SIZE-1));fprintf(fid,'%d)  -- For
Polynomial %d\n',TestOutput(n,TEST_VECTOR_SIZE),n);

        fprintf(fid,'    );\n');



    end

end


fprintf(fid,'\n\nend polynomial_pkg;\n');


fclose(fid);
```

```
%-------------------------------------------------------------------------------------------------
%% PolynomialModuleModel: Bit accurate model of the Polynomial vhdl module.

% For the same input, this function will give exactly the same output as the
polynomial module.

%-------------------------------------------------------------------------------------------------

% Designed and implemented by Dr. Aziz AhmedSaid, University of Manchester, UK.

% Email: aziz.ahmedsaid@manchester.ac.uk or a.ahmedsaid@qub.ac.uk

% Created in: September 2011

% Last modification: May 2012

%-------------------------------------------------------------------------------------------------
% THIS MATERIAL (DOCUMENTS AND/OR SOURCE CODE) IS CONFIDENTIAL.

% WE REMIND OUR COLLEAGUES THAT MEMBERS ON THE PROJECT ARE BOUND BY THE
CONFIDENTIALITY CLAUSE IN THE

% 'FP7 GRANT AGREEMENT - ANNEX II - GENERAL CONDITIONS' (SECTION 3 II.9) AND CAN'T
DISCLOSE THE

% CONTENT TO ANY THIRD PARTY WITHOUT OUR PERMISSION.

% THE UNIVERSITY OF MANCHESTER IS CURRENTLY IN THE PROCESS OF ASSESSING COMMERCIAL
INTEREST IN THE

% DEVELOPMENT AND MAY WISH TO PROTECT IT IF THAT'S THE CASE.

% THE DECISION ABOUT HOW THE SOURCE CODE WILL BE MADE AVAILABLE AND THE LICENSING
TERMS WILL BE MADE

% ONCE THE ASSESSMENT IS COMPLETE.

% THIS MATERIAL IS PROVIDED WITH NO WARRANTY WHATSOEVER.

%-------------------------------------------------------------------------------------------------

function
Output=PolynomialModuleModel(COEFFICIENTS_WIDTH,PRODUCTS_WIDTH,INPUT_WIDTH,OUTPUT_W
IDTH,MAXX,ProductsMSP,Coefficients,Input)


N=length(Input);

K=length(Coefficients)-1;

NC=COEFFICIENTS_WIDTH;

Nx=INPUT_WIDTH;

Np=PRODUCTS_WIDTH;

c=Coefficients;


Output=zeros(1,N);

prod=zeros(1,K);

Ntp=min(Np+NC-2,2*MAXX-2);

for n=1:N,
```

```
    prod(1)=floor(Input(n)*c(K)/2^(NC+Nx-Np)+.5);

    for k=2:K,
        tmp = floor(prod(k-1)*c(K-k+1)/2^(NC+Np-Ntp)+.5);
        prod(k)=floor(tmp*Input(n)/2^(Ntp+Nx-Np)+.5);
    end

    proda=floor(prod.*2.^(ProductsMSP-Np));

    s=(c(K+1)+sum(proda))/2^OUTPUT_WIDTH;

    Output(n)=(s-fix(s))*2^OUTPUT_WIDTH;

end
```

Outstanding Issues: NIL

DOCUMENT CONTROL: NA

DOCUMENT NAME: Uniboard_Firmware_Design_Document8p6_Pulsar_Binning.doc

DOCUMENT CHANGE RECORD 2-08-2015 Version-01