Andes System Architecture Design

Yan Zhu

Feb 18, 2021

CONTENTS

CHAPTER ONE

DOCUMENT SCOPE

This document discusses system design of Andes SoC. It mainly includes design details of radar signal processing algorithms. The main content is confined to logic level function description. Some special requirements for HW behavior is mentioned but it puts little constraints on RTL implementation. For example, when describing FFT, we only specify input/output size, format, radix size and so on. The HW details, such as number of butterfly engines and the exact RTL-related micro-architecture, are not included here. In other words, the description here only cover up-to bit-level behavior.

CHAPTER

TWO

OVERVIEW

Andes's radar signal processing architecture is fundamentally different from Alps in following ways:

1. The whole data processing chain is now divided into two parts: radar processing frontend and radar processing
2. Frontend functional blocks operate mainly in time scale of ADC sampling period.
3. Backend is more like traditional radar baseband processor.
4. Backend consists of several independent IPs or engines. These engines are controlled by a dedicated CPU/Sequencer to manipulate input/output data in a flexible way.
5. All backend engines share the same memory pool with other parts of the chip, such as DSP and other CPUs.

2.1 Birdeye View

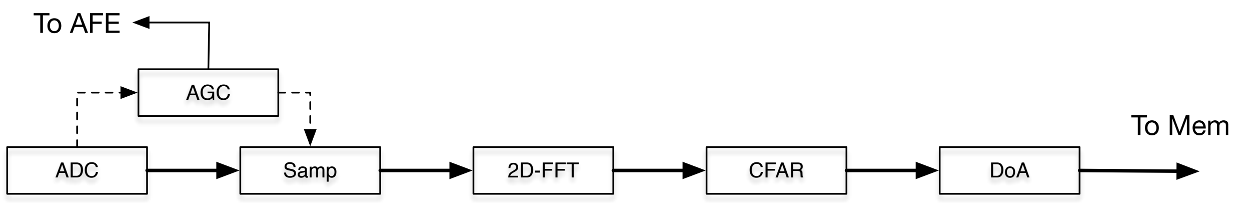

Fig. 2.1: Signal/Data processing flow of Alps radar baseband processing Unit

Fig. 2.1 shows the data processing flow of Alps. It is a fixed data processing flow with some variation on top of it. For example, it essentially allows stopping at any stage and starting from any stage. Andes is required to provide more flexibility where data processing flow could be arbitrary at least in principle.

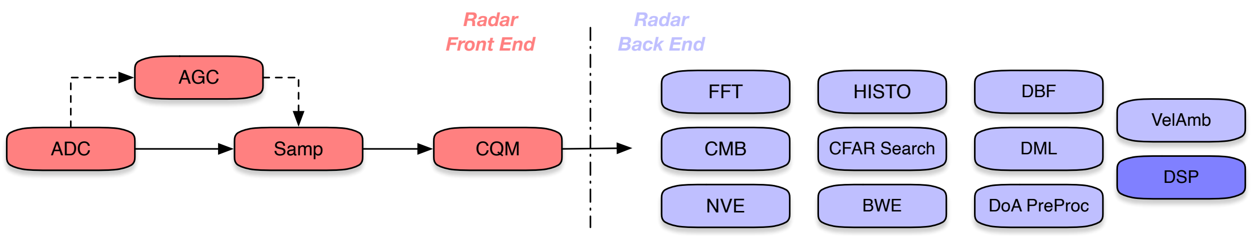

Fig. 2.2: Functional partition of Andes' radar data processing chain

Comparing with Fig. 2.1, Fig. 2.2 shows a quite different architecture, where all backend modules are not tied to a particular data processing flow. Note that backend modules are controlled and scheduled by a dedicated CPU and a Sequencer, which are not shown in Fig. 2.2 and this document will not discuss their design detail.

2.2 Radar Frontend

Front-End includes most timing critical functions. Most of their functions need to be finished within either one-chirp period or ADC sampling time scale. Therefore, its processing flow is generally fixed with certain tuning knobs built-in to achieve certain level of flexibility.

- **ADC**: It samples IF signal continuously. It is up-to remaining HW to track the start and end points of frames and chirps.
- **AGC**: It follows the same design of Alps. It uses first three chirps to find the best Rx gain for the remaining frame. It has certain interactions with analog/RF as Alps.
- **Samp**: It has two basic functions:
  - i) Downsampling ADC output with programmable decimate filter;
  - ii) Cutting frame samples into chirps. As Alps, it should have flexibility to control the backoffs at chirp begin and end to avoid RF/analog unsettling periods.
- **CQM**: It monitors data quality of each chirp. It includes interference detection and mitigation functions.

2.3 Radar Backend

At backend, the data processing timing is less critical than front-end. In other words, it is assumed that we can meet the update rate requirement with reasonable HW resource[1]. All modules in backend can be viewed as a HW engine. Given input data and input parameter, they will produce the corresponding data output after some time. From system design view, we do not have much visibility of timing constraints. But the each engine's timing constraints should be defined based on the overall update rate constraints.

- **FFT**: HW accelerator for Fast-Fourier Transform. It also includes certain pre/post process, Inverse FFT and STFT modes.
- **CMB**: It combines several equal-size array into one single array with certain weights or preprocess. One of its primary usages is to generate CFAR coherent/non-coherent combination before CFAR searching.
- **NVE**: It estimates noise-level along range gate as Alps. But it does not includes moving average part, which is proposed to have a separate engine or compute via DSP.
- **NVE2**: It is a histogram-based noise-level estimate, which does not require moving averaging across range gates. It is more computationally intensive than **NVE** but provides more robust results in principle. Both are kept in short term. In long term, it aims to keeping **NVE2** only.
- **CFAR Search**: Given a rectangular region input, the engine will search specific rectangular sub-region inside the given input for qualified data points and output corresponding results.
- **BWE**: Given input arrays, the engine extends these arrays backwardly and forwardly.
- **DBF**: Same function as Alps's DBF algorithm with size extension. With given input Rx vector and steering vectors, it performs traditional digital-beamforming algorithm with certain variations.
- **DML**: Same function as Alps's DML algorithm with size extension. With given input Rx vector and steering vectors, it performs up-to 2-object deterministic maximum likelihood algorithm.
- **DoA PreProc**: It formats input vector in certain ways. Its main purpose is to process Rx vectors before feeding into DBF or DML engines, such as weighing, phase correction based on velocity ambiguity number, and data rearrangement.

[1] This document will have some update rate requirements. However, how to meet it is not discussed here.

- **VelAmb**: Anti-velocity-ambiguity module. It uses same algorithm as Alps with extension to two additional chirps for each subframe.
- **DSP**: Digital signal processor. It serves to increase further flexibility that current engines can not provide.
- **Mov Avg**: It does moving average based along input data. It works with **NVE** engine to achieve same algorithm of NVE in Alps. This is optional one, since this can be done through DSP. If so, special care is required for RTL matching.

CHAPTER

THREE

REQUIREMENT

Some special requirements are discussed here. Some of them may require special care from system designers, and some of them need to be fulfilled by RTL designer. In all case, system designers need to communicate actively with RTL designers to ensure no requirement is missing.

3.1 Data Processing Flow

In principle, backend supports arbitrary data processing flow, but special attention is required for some common used flows. Based on CFAR searching matrix, we defines following two flows: Range-Doppler (RD) flow and Doppler-Angle (DA) flow.

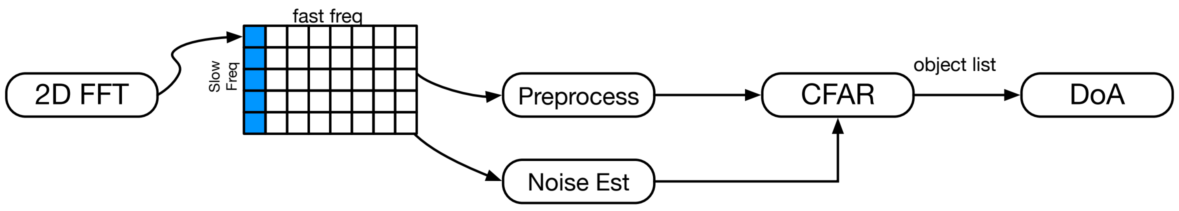

Fig. 3.1: Data Processing of Range-Doppler Flow

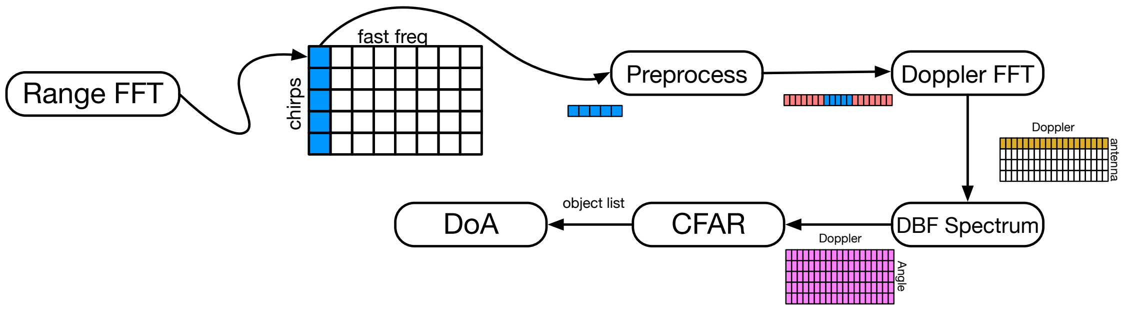

Fig. 3.2: Data Processing of Doppler-Angle Flow

RD flow is shown in Fig. 3.1 and is essentially the same flow adopted in Alps. It first does both range and doppler FFT and the following 2D-CFAR searching is done on range-doppler matrix. The CFAR outputs are fed into DBF or DML engineer for DoA. Different from Alps, Andes achieve this flow by calling the HW engines through CPU/sequencer.

Fig. 3.2 shows DA flow accordingly. First, it performs FFT for each chirp. For each range gate, before doing the doppler domain FFT, BWE algorithm extends the input data to a longer version. After converting data to doppler domain for each channel, DBF spectrum is computed along each doppler gate so that the output is a doppler-angle matrix. CFAR search is done on the doppler-angle matrix to resolve the doppler index of the object based on various

CFAR criteria. Next, one need to feed the Rx vectors into DoA engine to refine the object DoA results. After that, repeat the whole process for next range gate.

Both Fig. 3.1 and Fig. 3.2 are shown in functional level. From radar backend engines level, they are shown in Fig. 3.3 and Fig. 3.4, respectively. In both figures, colored round boxes shows the particular backend engines and the dotted lines indicate how these engines are used in each processing step.

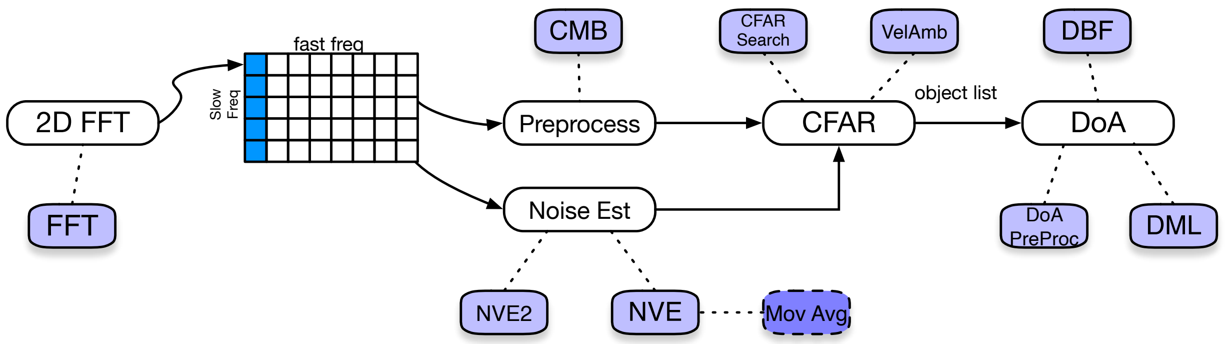

Fig. 3.3: Range-Doppler Flow from Engine Level

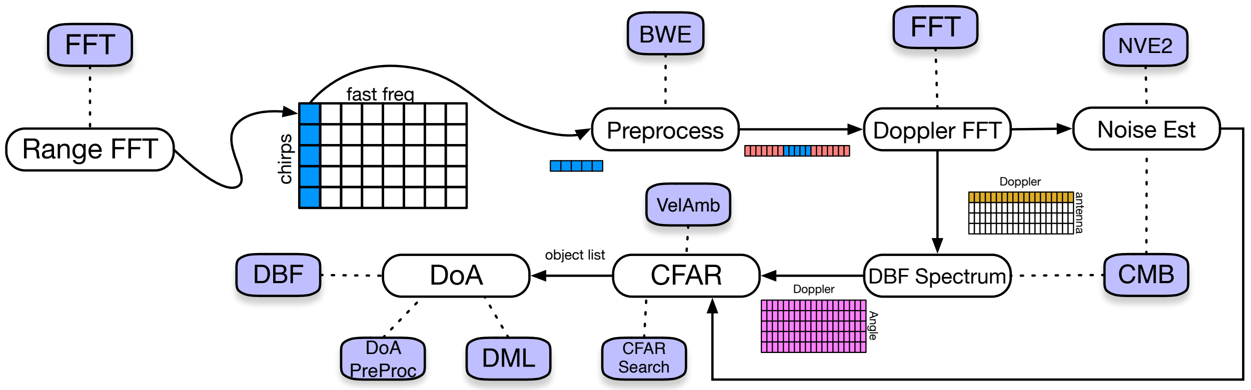

Fig. 3.4: Doppler-Angle Flow from Engine Level

System designers mainly use these two flows to tuning the performance, fine trimming the algorithm, and ensure C-Model implemented correctly.

3.1.1 Range-Doppler Flow

In terms of data processing step, it is exactly same as one used in Alps. Please refer to *Alps Baseband User Guide* or corresponding C-model for details. The major differences in Andes are:

- **NVE2** can be optionally used for noise estimate;
- Some processing data specs will have some changes for example, FFT size and so on. These changes will be detailed in next chapter.
- Before doing DoA, CPU intervines to format and feed the Rx signals to DoA process.

3.1.2 Doppler-Angle Flow

This flow is new, so more details are provided here. Notations used in discussion are listed below:

- N: Number of valid samples in a chirp;
- n: Sample index within chirp, i.e., $0 \leq n < N$;
- L: Number of valid chirps;
- l: Chirp index, i.e. $0 \leq l < L$;
- A: Number of receive channels;
- a: Receive channels index, i.e., $0 \leq a < A$;
- x: Valid ADC sampled signal, e.g., $x_{n,l,a}$ denotes valid ADC sample at a-th channel, l-th chirp, n-th sample.
- K: Size of range domain FFT;
- k: Range gate index, i.e., $0 \leq k < K$;
- X: Output of range domain FFT with all compensations;
- wr: Window for range domain FFT;
- wd: Window for doppler domain FFT;

3.1.2.1 Range FFT

For each chirp l, suppose the input data is denoted by $x_{n,l,a}$, then its output $X_{k,l,a}$ is given by:

$$X_{k,l,a} = \sum_{n=0}^{N-1} wr_n x_{n,l,a} \exp\left(\frac{j2\pi kn}{K}\right)$$

For simplicity, it is assumed that all compensations, such as phase scramble or frequency hopping, have been done in $X_{k,l,a}$. In other words, some compensations are included in above equations implicitly.

3.1.2.2 Pre-Process

There are two sub-steps for pre-processing. First, for fixed k, prediction coefficients $a_1, a_2, \ldots, a_O$ are computed from $X_{k,l,a}$, $0 \leq l < L$ and $0 \leq a < A$ via burg algorithm. Next, use these coefficients to extend $X_{k,l,a}$ backwardly and forwardly along l to get $\tilde{X}_{k,l,a}$:

$$\tilde{X}_{k,l,a} = \begin{cases} \sum_{i=1}^{O} a_i^* \tilde{X}_{k,i+l,a} & \text{if } l < 0 \\ X_{k,l,a} & \text{if } 0 \leq l < L \\ \sum_{i=1}^{O} a_i \tilde{X}_{k,l-i,a} & \text{if } l \geq L \end{cases} \tag{3.1}$$

In (3.1), value of $\tilde{X}_{k,l,a}$ for $l < 0$ or $l \geq L$ is defined recursively. To avoid negative indices, we use $\tilde{X}_{k,\tilde{l},a}$ instead, where $0 \leq \tilde{l} < \tilde{L}$. It is also assumed that data of $X_{k,\cdot,a}$ always sits in the middle of $\tilde{X}_{k,\cdot,a}$ and $\tilde{L}/L \geq 1$.

3.1.2.3 Doppler FFT

Traditional FFT is computed with $\tilde{X}_{k,\cdot,a}$ for each a:

$$Y_{k,p,a} = \sum_{\tilde{l}=0}^{\tilde{L}-1} wd_{\tilde{l}} \tilde{X}_{k,\tilde{l},a} \exp\left(\frac{j2\pi\tilde{l}p}{P}\right)$$

where $wd_{\tilde{l}}$ are doppler window coefficients.

3.1.2.4 Noise Estimate

Noise estimator is using non-coherent combination of $Y_{k,p,a}$ as its input:

$$\hat{Y}_{k,p} = \sum_{a=0}^{A-1} \|Y_{k,p,a}\|^2$$

NVE2 creates a histogram based on the input data to give an estimate of noise variance at range gate k. Further details of **NVE2** are described in next chapter.

3.1.2.5 DBF Spectrum

It essentially does digital beamforming for each doppler gate:

$$Z_{k,p,b} = \sum_{a=0}^{A-1} \|v_{b,a} Y_{k,p,a}\|^2$$

where v_b denotes the steering vector along directions of b and $v_{b,a}$ denotes the a-th coefficient of vector v_b.

3.1.2.6 CFAR

CFAR is done on 2D-array $Z_{k,p,b}$, where index k is fixed. Note that the CFAR detectors are essentially same as in RD flow. But it has following differences via taking some advantages of properties of the doppler-angle matrix:

1. Both doppler and beam domains requires peak detectors in normal usage. The peak detector can be limited to $5 \times 3 (p \times b)$. Therefore up-to 3 set of beamed data need to be cached for computation.
2. CFAR sliding window size can also be limited to $x \times 3$.
3. At boundary of b, one should have option to do it cyclically.
4. Both indices p and b should be reported for each bin passing the CFAR criteria.
5. CFAR should pull the corresponding FFT values for CPU to form the Rx vector.

The last requirement requires further clarifications. Suppose that $\hat{p}$. It is required to output following two types of FFT values.

$$Y_{k,\hat{p},a} \quad \text{for} \quad 0 \leq a < A$$

and

$$\sum_{l=0}^{L-1} X_{k,l,a} \exp\left(\frac{j2\pi\hat{p}l}{P}\right) \quad \text{for} \quad 0 \leq a < A$$

The difference between these two is that the first one uses both extended and original data while the second one only uses original data.

3.1.2.7 DoA

As in RD flow, CPU will format and feed the Rx signals to `DoA PreProc` and then `DBF` or `DML`.

3.2 Other Requirement

3.2.1 Update Rate

For RD flow, the update rate targets for rate no slower than Alps. For DA flow, it targets 30 frames per second (FPS). System designers should communicate this requirement with RTL designers.

3.2.2 Flexible Chirp Profiles

The basic requirement from marketing is to have 512 different chirps in one frame. With this requirement, following system settings should be able to variate for each individual chirp:

1. Valid chirp sampling start and stop points
2. Pre-Range FFT compensations
3. Pre-Range FFT DC cancellation
4. Pre-Range FFT template cancellation.
5. Range FFT window coefficients
6. Range FFT sizes
7. All settings for CQM
8. Three chirps for AGC may have different shape so its observation window should be able to change accordingly.
9. Observation window for saturation detector also should be able to change accordingly for normal frame.

There are several system settings are assumed to be *not* changed within a frame:

1. ADC sampling frequency;
2. Decimate filter coefficients and factor;
3. AGC mode;
4. Interference avoidance modes;
5. Anti-VelAmb mode.

At curent stage, it is not clear whether remaining system settings requires to change in chirp level or frame level only. System design should be cautious on future changes.

CHAPTER

FOUR

MODULES/FEATURES

This chapter gives some design details on modules and features.

4.1 AGC

Refer *Alps Baseband User Guide* for details.

4.2 Decimiate Filter

Down-sampling up-to factor up-to 64. In Alps, the internal bitwidth of decimiate filter is tuned to support factor of 8 downsampling. With factor increasing to 128, Andes is prefer to keep 16-bit input and 16-bit output.

Given the design constraints, the main challenge is that the filter has very narrow bandwidth. One may consider:

- Lower down the requirement to 32/16;
- Minor increases bitwidth of input/output, say 18 bits;
- Special design for 128 by using downsampling in-between of two filters.

4.3 Interference Avoidance

Alps supports phase scrambling (PS), frequency hopping (FH), and chirp shifting (CS) modes. In Andes, we do following adjustment:

- Phase Scramble:
  - It requires compensation after 1DFFT with together 4x4 complex compensation coefficients, which is part of **FFT** engine described later.
  - Ideally, this should be done at RF/Analog so that no compensation is needed at baseband side. It will be confirmed by Andes-A0 and Alps-Mini.
- Chirp Shifting:
  - No compensation is done at radar frontend and backend.
  - In radar frontend, sampling start point should be counted from the point that chirp is ramping up instead of at fixed periods as in Alps.
- Frequency Hopping:
  - No compensation is done at radar frontend and backend.

  - If there is no advantage, this mode should be removed completely.
- All Modes:
  - XOR chains should supported to update every several chirps instead of every chirp. For example, it should be able to do CS in group of four chirps.
  - XOR chains should be able to bypass certain number of subframes. For example, user can choose to do CS for first three AGC chirps or not when AGC mode is on.

4.4 Interference Detection

Let $x_{n,l,a}$ be valid n-th sample of Rx channel a within chirp l after decimate filter. There are two independent detectors: one for related low frequency interference and one for related high frequency interference.

For low frequency interference, it claims low frequency interference at n-th sample if and only if $|x_{n,l,a}| > \alpha T_{l,a}$, where α is a predetermined correction factor no less than 1 and

$$T_{l,a} = \frac{1}{N} \sum_{n=0}^{N-1} |x_{n,l,a}|$$

For high frequency interference, for fixed $\hat{n}$, collect $D^{\hat{n}}_{n,l,a} = x_{n-1,l,a} - x_{n,l,a}$ for $n = \hat{n} - W, \ldots, \hat{n} + W$. Let $\text{ZN}_{\hat{n},l,a}$ denote the number of zero-crossing in data $D^{\hat{n}}_{n,l,a}$ for $n = \hat{n} - W, \ldots, \hat{n} + W$. It claims that high frequency interference at $\hat{n}$ if and only if $\text{ZN}_{\hat{n},l,a} > \text{ZN}_{th}$, where ZN_{th} is a predefined threshold.

Tuning knobs are summarized as follows:

- α: correction factor for low frequency interference;
- W: Half window size of sliding window for high frequency interference;
- ZN_{th}: Predefined threshold of number of zero-crossing.

No need to make parameters depend on Rx channels. But these parameters are required to be updated for each chirp.

Output info are summarized as follows:

- Each detector should has her own outputs;
- Each detector output up-to 3 interference positions within a chirp;
- Output up-to 3 positions where the input data is saturated numerically;
- Output info should be separated for each Rx channels.

4.5 Bit Compression/Decompression

It is mainly applied to FFT results and it supports both 1D/2D FFT output:

- It takes 4 `cal_complex<14, 1, true, 4, false>` data (32×4 bits) to 64 bits.
- All blocks to be compressed are formed along dimension that FFT is computed. For example, to compress 1D-FFT, input block is created along range gate; to compress 2D-FFT, input block is created along doppler gate.

4.5.1 Compressor

Input data is an array with data type of `cal_complex<14, 1, true, 4, false>`. Output is compressed data array with unit of 64-bit.

4.5.1.1 List of parameters

- Input length;
- Compression algorithm indicator: Even if there is only one algorithm, this option should be reserved in C-Model;
- Other Compression options: Fine tuning of the compression algorithm, such as block length and so on.

4.5.2 De-Compressor

It is a reversing process of de-compressing. The input is an array of compressed data and the output is an array with data type of `cal_complex<14, 1, true, 4, false>`

4.5.2.1 List of parameters

- Input length with unit of 64-bit or compressed block length;
- Compression algorithm indicator: Even if there is only one algorithm, this option should be reserved in C-Model;
- Other Compression options: Fine tuning of the compression algorithm, such as block length and so on.

4.6 FFT Engine

Data processing flow of FFT engine is shown in Fig. 4.1. Each computation step can be skipped optionally. Some details of processing steps are described as below.

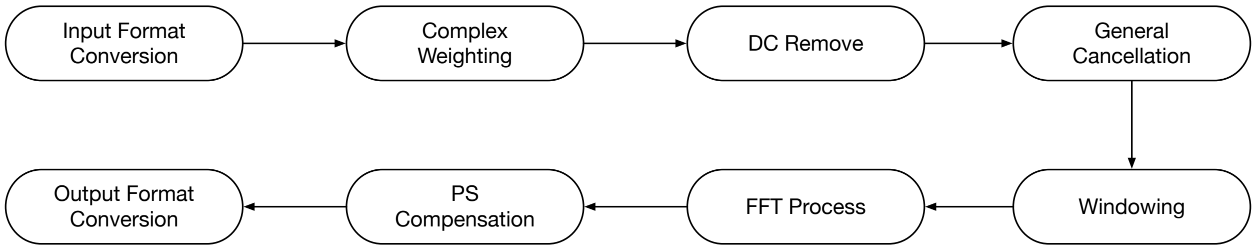

Fig. 4.1: Data processing flow of FFT engine

4.6.1 Input Format Conversion

All inputs are organized in array and the supported data types are listed below:

1. `ac_complex<ac_fixed<29, 1, true>>`
2. `cal_complex_ce<14, 1, true, 4, false>`

All inputs are converted to data type `ac_complex<ac_fixed<29, 1, true>>` before doing any computation.

4.6.2 Complex Weighting

Suppose the input of this stage is denoted by x_i, $i = 0, \dots, I-1$. Then the output of this stage is given by $x_i w_i$, where w_i is of data type `ac_complex<ac_fixed<15, 2, true>>`. Data type of its output is `ac_complex<ac_fixed<29, 1, true>>`. Purpose of this stage is to apply certain compensations.

4.6.3 DC Removal

Suppose the input of this stage is denoted by x_i, $i = 0, \dots, I-1$. The output is given by $y_i = x_i - dc$, where $dc = \frac{1}{I}\sum_{i=0}^{I-1} x_i$. Value of $\frac{1}{I}$ should be computed before-hand and input as a parameter, whose data type is `cal_float<6, 2, false, 4, false>`. Its output data type is `ac_complex<ac_fixed<29, 1, true>>`.

4.6.4 General Cancellation

This is used for near-field cancellation. With input of x_i, the output is given by $y_i = x_i - \alpha t_i$, where t_i is the template sample for cancellation and α is the scaling factor. The data type of output is `ac_complex<ac_fixed<29, 1, true>>`.

4.6.5 Windowing

With input of x_i, the output is $x_i w_i$, where w_i is of data type `ac_complex<ac_fixed<15, 2, true>>`. The output data type is `ac_complex<ac_fixed<29, 1, true>>`.

4.6.6 FFT Processing

It supports following modes exclusively:

1. FFT
2. Inverse FFT
3. Short-Time FT (STFT)

All modes only support FFT size of 2^n, where $3 \leq n \leq 12$. Suppose input data is x_i. FFT is computed as

$$X_k = \frac{1}{2^M}\sum_{i=0}^{I-1} x_i W_N^{ki}$$

where M is controlled by shifter mask as in Alps and W_N^{ki} are twiddles. For inverse FFT, it is recommended to use following implementation:

$$\begin{aligned} X_k &= \frac{1}{2^M}\sum_{i=0}^{I-1} x_i W_N^{-ki} \\ &= \frac{1}{2^M}\left(\sum_{i=0}^{I-1} x_i^* W_N^{ki}\right)^* \end{aligned}$$

In other words, it takes conjugate for input and run through original FFT computation. The inverse FFT result is nothing but the conjugate version of FFT computation results.

Given input x_i, STFT sample size T, window coefficients, STFT size F, and skipping step S, it computes all STFT outputs not just one of them:

1. Window coefficients should be used a different window coefficients for FFT
2. Data type of output is `ac_complex<ac_fixed<29, 1, true>>`.
3. All outputs should be in right order.

4.6.7 PS Compensation

With input x_i, the output is given by $x_i w$, where w is independent of sample index and is of type `ac_complex<ac_fixed<15, 2, true>>`.

4.6.8 Output Format Conversion

It supports following output data types:

1. `ac_complex<ac_fixed<29, 1, true>>`
2. `cal_complex_ce<14, 1, true, 4, false>`

4.6.9 List of Parameters

- Input format flags;
- Bypasses for complex weight, DC removal, general cancellation, PS compensation;
- input len: To indicate length of input data
- window coefficients:
  - window1
  - window2
  - STFT window
- window options: Controls the behavior at windowing stage
  - 0: bypass windowing stage
  - 1: apply coefficients from window1
  - 2: apply coefficients from window2
- fft_exp: 1 $\ll$ fft_exp indicates the FFT, IFFT or STFT size

- shifter_mask: fine control of right shift in each FFT stage in format of `ac_int<12, false>`
- fft_mode: Controls on FFT processing stage
  - 0: bypass
  - 1: FFT mode
  - 2: IFFT mode
  - 3: STFT mode
- STFT Size: STFT input length
- STFT Step: STFT skip length

4.7 NVE Engine

The noise estimation is done in two steps: first, NVE engine will give an estimate for each range gate; second, a moving average performs along range gates. Moving average can be achieved via DSP and discussion here focuses on the first step.

For fixed range gate k and channel a, input data is range-FFT $X_{k,l,a}$ along all chirps $0 \leq l < L$ and 2D-FFT of corresponding range-gate $Y_{k,p,a}$ with p taking T different values. The output of NVE depends on following two values

$$Var = \frac{1}{2^\alpha} \frac{1}{LA} \sum_{a=0}^{A-1} \sum_{l=0}^{L-1} \|X_{k,l,a} - dc\|^2$$

where $dc = \frac{1}{L} \sum_{l=0}^{L-1} X_{k,l,a}$.

$$TP = \gamma \sum_{p:P_p/2^\beta < P_{\min}} P_p$$

where $P_p = \frac{1}{A} \sum_{a=0}^{A-1} \|Y_{k,p,a}\|^2$, $P_{\min} = \min_p P_p$, $\beta = 1, 2, 4, 8, \ldots$, and γ is a predetermined scalar.

The final NVE output is the minimum between Var and TP.

4.7.1 Summary of Input, Output and Parameters

- Input:
  - range-FFT and selected 2D-FFT results.
  - Two input data types: `ac_complex<ac_fixed<29, 1, true>>` and `cal_complex_ce<14, 1, true, 4, false>`.
- Output:
  - Noise estimate value in type of `cal_float<15, 1, false, 5, false>`.
- Parameters:
  - input format control flag;
  - Number of channels, i.e., value of A;
  - Number of 2DFFT tones, i.e., value of T;
  - Number of chirps, i.e., value of L;

- Tone shifter, i.e., value of β, in `ac_int<3, false>`;
- Covar shifter, i.e., value of α, in `ac_int<8, false>`;
- win scalar, i.e., value of $\frac{1}{LA}$, in type of `cal_float<6, 2, false, 4, false>`;
- dc scalar, i.e., value of $\frac{1}{L}$, in type of `cal_float<6, 2, false, 4, false>`;
- tone scalar, i.e., value of γ, in type of `cal_float<6, 2, false, 4, false>`.

4.8 HISTO Engine

It generates histogram from input array. As a by-product, it also provides noise estimates from input array based on the generated histogram. Its input is an array and its element's possible data types as follows:

1. `cal_float<15, 1, false, 5, false>`;
2. `cal_float<16, 1, false, 6, true>`.

If the input data type is `cal_float<15, 1, false, 5, false>`, it will be first converted to `cal_float<16, 1, false, 6, true>` for further processing.

The first step is to find the maximum and minimum values because the bin boundaries depend on these two values.

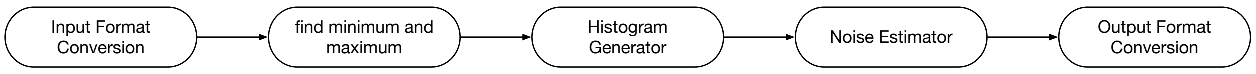

Fig. 4.2: Data processing flow of HISTO engine

4.8.1 Histogram Generator

There will be several different modes to process the input data with given total number of bins B

1. Linear mode:

   The bins are created as B intervals with equal length, such that $b_0 = x_{\min}$ and $b_B = x_{\max}$. The input data will be gone through to associate to one of these bins if and only the value satisfies $b_i \leq x < b_{i+1}$, where b_i and b_{i+1} are boundaries of i-th bin.

2. Log2 mode:

   The only difference between linear mode and this one is how the bin boundary is defined. Let b_i and b_{i+1} denote the bin boundaries of i-th bin. Note that we have $b_i < b_{i+1}$ and $b_0 = x_{\min}$ and $b_B = x_{\max}$. For $0 < i < B$, $b_i = (b_{i-1} + b_B)/2$.

3. Fast histogram mode:

   TBD

4.8.2 Noise Estimator

The noise estimator is given by average of all points associating with certain bin. The bin is selected with following two schemes:

1. Bin contains largest number of points
2. Predefined bin index

4.8.3 Outputs

Its output consists of three parts:

1. An array consists of the histogram results. Data of the same bin is packed together as shown in shown in Fig. 4.3. The datatype can be either of following :
   - `cal_float<15, 1, false, 5, false>`
   - `cal_float<16, 1, false, 6, true>`
2. An array indicates the bin boundary of first array. For example, the corresponding array of Fig. 4.3 should be like: 0, 8, 13, 16, . . .
3. NVE outputs with data type of `cal_float<15, 1, false, 5, false>`.

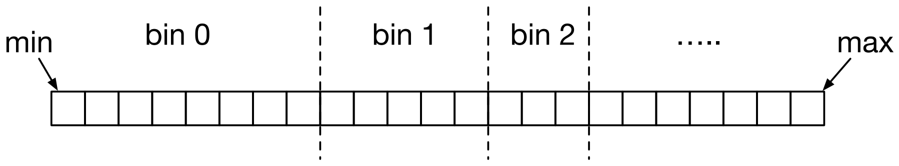

Fig. 4.3: An illustration on packing format of HISTO output array

4.9 CMB Engine

Input of **CMB** engine is a 2D array and its purpose is to combine the 2D array into a 1D array. It supports following data type

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

If the input data is `ac_complex<ac_fixed<29, 1, true>>`, it will be converted to `cal_complex_ce<14, 1, true, 4, false>` before further computing.

Fig. 4.4 shows the data processing flow of CMB engine. It has two sub-engines: coherent combiner and non-coherent combiner. Suppose that the input data is $x_{a,n}$. Non-coherent combiner behaves as

$$outA_n = \sum_a \|x_{a,n}\|^2$$

Coherent combiner has two outputs:

$$outB_n = \frac{1}{A}\sum_a x_{a,n}v_a$$
$$outA_n = \|outB_n\|^2$$

where v_a are complex weighting coefficients in format of `ac_complex<ac_fixed<14, 1, true>>`.

When selected coherent mode, engine should outputs both $outA$ and $outB$. $outA$ is in format of `cal_float<15, 1, false, 5, false>` and $outB$ is in format of `cal_complex_ce<14, 1, true, 4, false>`.

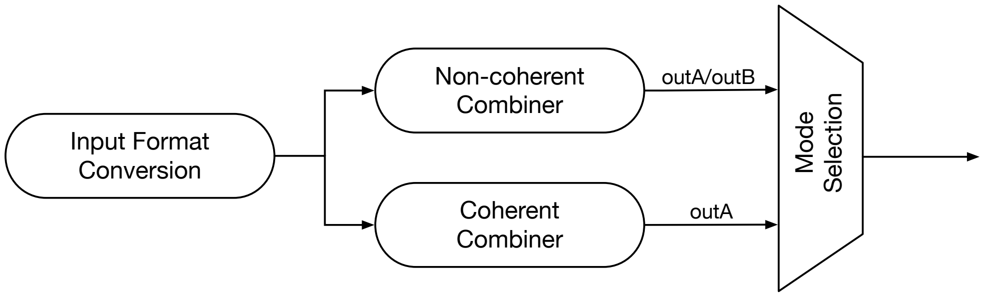

Fig. 4.4: Data processing flow of CMB engine

4.9.1 List of Parameters

- Input lengths for both dimensions
- mask: to select subset of a for combination
- norm shift: used in coherent combiner in order to prevent saturation and approximation of $1/A$.
- weighting vector: complex weighting vector, i.e., v_a.

4.10 CFAR Search Engine

Its inputs are

i. 2D array for CFAR searching

ii. noise estimates along row of the 2D array.

Data type of both are `cal_float<15, 1, false, 5, false>`.

Within the input region, CFAR/peak detectors operated same as Alps. Outputs are lists of detected objects contains following fields:

- Array indices
- Interpolation results along both dimensions
- noise level `cal_float<15, 1, false, 5, false>`
- bin power in `cal_float<15, 1, false, 5, false>`

It is worth to note that

1. The 2D sliding window just assume zero power for bin out of boundary of the input 2D array.
2. CFAR/Peak detectors may only apply to sub-rectangular region of the input region as shown in Fig. 4.5. Cyclic searching along certain domain can be achieved by feeding more data to the engine.

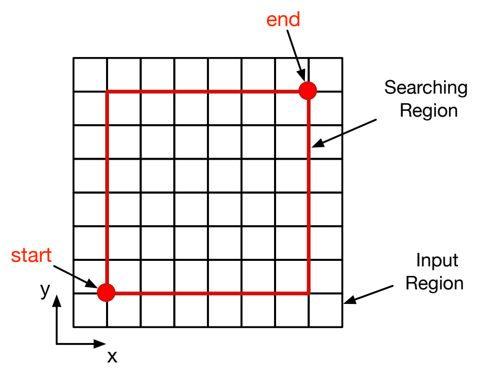

Fig. 4.5: Input Region/Searching Region of CFAR Searching Engine

4.10.1 List of Parameters

- Input format: it is a place-holder variable here. no functions
- len1, len2 : Lengths of the input 2D array and len1 is also input length of the second input.
- start: Coordination of starting point for searching as shown in Fig. 4.5
- end: Coordination of ending point for searching as shown in Fig. 4.5
- Parameters for peak detector: mask and threshold as in Alps
- Parameters for CFAR detector: mask and tuning knobs as in Alps
- range interpolation flag
- doppler interpolation flag

4.11 VelAmb Engine

It uses additional chirps to resolve velocity ambiguity number q. In Alps, only one additional chirp in each subframe is used in this purpose. Andes supports two additional chirps as shown in Fig. 4.6. Note that there are three different additional delay associate with these two chirps. The reason that using three instead of two additional delay is to ease the way to waveform design accordingly.

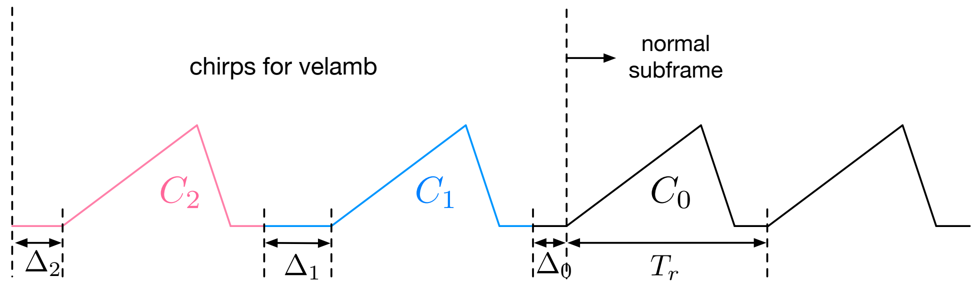

Fig. 4.6: An illustration on two additional chirps for anti-velocity-ambiguity

The input is Rx vectors of same reflector from chirps C_0, C_1, and C_2, respectively. They are denoted as $x_{c,a}$, where $c = 0, 1, 2$ and $0 \leq a < A$. Note that here designer can assume that maximum possible A is number of *physical* Rx channels. Its inputs are three 1D arrays and data type of its elements can be one of following:

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

If input data type is `ac_complex<ac_fixed<29, 1, true>>`, it will be converted to `cal_complex_ce<14, 1, true, 4, false>` first for further computation. The output is ambiguity number q.

The process can be summarized with the following equation:

$$\hat{q} = \arg\max_{q} \sum_{c=1}^{2} \sum_{a=0}^{A-1} \|x_{c,a} - x_{0,a} \exp(j2\pi f_c(q))\|^2$$

where the notations are clarified as below:

- The optimization is taken over a fixed set of value of q, say, $\{Q_{\min}, Q_{\min}+1, \ldots, Q_{\max}\}$.
- Function $f_c(q)$, $c = 1, 2$, is precomputed phase difference between chirp C_0 and C_c by assume the ambiguity number value is q.

4.11.1 List of Parameters

- Input format
- Input length: 3 arrays share same length
- List of values $f_c(q)$ for different q value. Maximum size is 100 and minium q value could be -100.
- nfft: FFT size
- p: doppler index

4.12 BWE Engine

This module is to do an extension on both ends of the input data. Its input should be 2D array with its elements are in following possible format:

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

Its output is also a 2D array and its elements are in following possible format

1. `ac_complex<ac_fixed<29, 1, true>>`
2. `cal_complex_ce<14, 1, true, 4, false>`.

The processing flow is shown in Fig. 4.7.

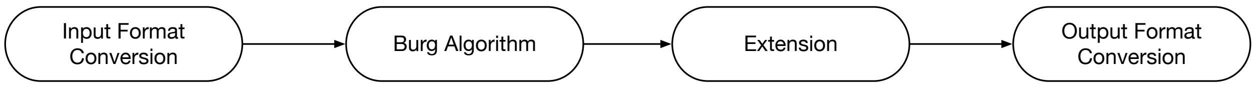

Fig. 4.7: Processing Flow of BWE Engine

4.12.1 Burg Algorithm

Suppose the input 2D array is given by $x_{l,a}$, where l is chirp index ($0 \le l < L$) and a ($0 \le a < A$) is channel index. Burg algorithm is to find a set of coefficients, denoted by $a_1, a_2, \ldots, a_O$. A sample python code is shown in Listing 4.1. Input `sigs` is a two dimensional array whose first dimension is indexed with chirp and second one is indexed with Rx channel.

Listing 4.1: Burg Algorithm with multiple input arrays

```python
def compute_burg_mchan(sigs, win_type, order, **kw) :
    e = 2 * np.sum(np.abs(sigs)**2)
    den = e
    f = sigs
    b = sigs
    for i in range(1, order+1) :
        num = 2 * np.sum(f[:, 1:] * np.conj(b[:, :-1]))
        den = np.sum(np.abs(f[:, 1:])**2) + np.sum(np.abs(b[:, :-1])**2)
        aii = num / den
        if i == 1 :
            a_pre = np.array([aii])
        else :
            a_new = a_pre - aii * np.conj(a_pre[::-1])
            a_pre = np.concatenate((a_new, np.array([aii])))
        if i != order :
            f_new = f[:, 1:] - aii * b[:, :-1]
            b_new = b[:, :-1] - np.conj(aii) * f[:, 1:]
            f = f_new
            b = b_new
    aa = a_pre
    a = np.zeros(order+1, dtype=np.complex)
    a[0] = 1
    a[1:] = a[1:] - aa
    return a, aa
```

4.12.2 Extension

Coefficients $a_1, a_2, \ldots, a_O$ found in previous step are used for forward extension and their conjugates are used for backward extension. Let $\tilde{x}_{l,a}$ denote the extended array. Then we have

$$\tilde{x}_{l,a} = \begin{cases} \sum_{i=1}^{O} a_i^* \tilde{x}_{i+l,a} & \text{if } l < 0 \\ x_{l,a} & \text{if } 0 \le l < L \\ \sum_{i=1}^{O} a_i \tilde{x}_{l-i,a} & \text{if } l \ge L \end{cases} \tag{4.1}$$

In other words, when $0 \le l < L$, $\tilde{x}_{l,a} = x_{l,a}$. When $l < 0$ or $l \ge L$, $\tilde{x}_{l,a}$ is obtained recursively. To avoid negative indices, denote the output as $\tilde{x}_{\tilde{l},a}$, where $0 \le \tilde{l} < \tilde{L}$. It is also assumed that data of $x_{\cdot,a}$ always sits in the middle of $\tilde{x}_{\cdot,a}$ and $\tilde{L}/L \ge 1$.

For design perspective, we put following constraints:

1. $\tilde{L} \le 2^{12}$ and $\tilde{L}/L \le 16$;
2. Predictor order O can take value from 1 to 40.

4.12.3 List of Parameters

- Input format
- Input lengths: data length of both dimension
- extension factor: i.e. $\tilde{L}/L$
- order: predictor order O

4.13 DoA PreProc Engine

The main purpose of this engine is to format Rx vectors before feeding it into DBF or DML engine. The input is a 2D-array, whose elements support following data type:

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

All input will be converted to data type of `cal_complex_ce<14, 1, true, 4, false>` first before further computation. The output is always a 1D-array in format of `cal_complex_ce<14, 1, true, 4, false>`.

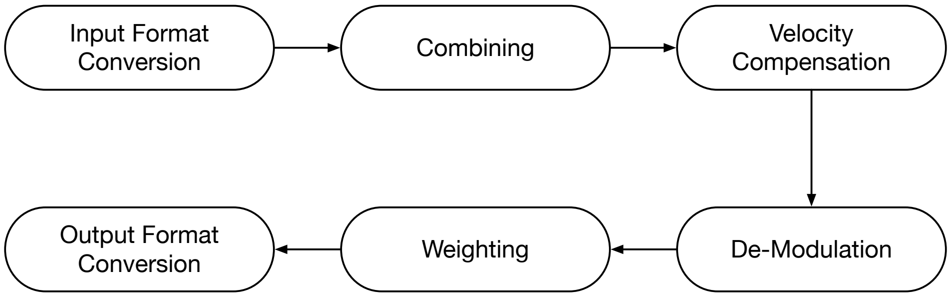

Fig. 4.8: Processing Flow of DoA PreProc Engine

The processing flow is shown in Fig. 4.8. Except for input/output, all stages can be bypassed. At input of each sub-block, all data is converted into format of `cal_complex_ce<14, 1, true, 4, false>`.

4.13.1 Combining

Suppose the input is denoted by $x_{p,a}$, where a is Rx channel index and index $p = 0, \dots, P-1$. When $P = 0$, this stage is bypassed. Otherwise, output following:

$$y_a = \sum_{p=0}^{P-1} x_{p,a}$$

The motivation of doing this is to take leakage into account, e.g., $P = 3$.

4.13.2 Velocity Compensation

It does velocity compensation based on doppler index p and velocity ambiguity number q. The virtual array group is assumed according to Fig. 4.9.

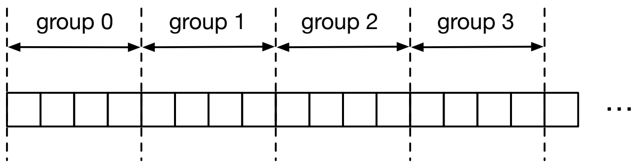

Fig. 4.9: An illustration on virtual array group partition by assuming number of active physical Rx channel is 4.

4.13.3 De-Modulation

Suppose number of active phaseical Rx channel is R. We re-arrange input vector $\mathbf{y}$ as following matrix $\mathbf{Y}$:

$$\begin{bmatrix} y_0 & y_1 & \cdots & y_{R-1} \\ y_R & y_{R+1} & \cdots & y_{2R-1} \\ \cdots & \cdots & \cdots & \cdots \\ y_{(V-1)R} & y_{(V-1)R+1} & \cdots & y_{VR-1} \end{bmatrix}$$

where the total number of virtual array group is V. Then left-multiplying a predefined $C \times V$ matrix $\mathbf{D}$, we obtain $\mathbf{Z} = \mathbf{DY}$. The rearrange $\mathbf{Z}$ into following vector as output:

$$[Z_{0,0}, Z_{0,1}, \ldots Z_{0,R-1}, Z_{1,0}, Z_{1,1}, \ldots Z_{1,R-1}, \ldots Z_{C-1,R-1}]$$

4.13.4 Weighting

It is simply a point-wise weighting with input z_a and output $z_a w_a$, where w_a is a complex weighting coefficients.

4.13.5 List of Parameters

- Input format
- Input lengths for both dimensions
- bypasses: stages of combining, velocity compensation, demodulation, and weighting
- doppler index p and velocity ambiguity number q
- nant: number of physical antenna
- nfft2: FFT size of doppler domain
- weights: values of w_a in format of `ac_complex<ac_fixed<14, 1, true>>`
- dem: matrix of $\mathbf{D}$ in format of `ac_complex<ac_fixed<14, 1, true>>`
- vel_comp: predefined constant in format of `ac_fixed<16, 1, false>`
- fdtr: predefined constant in format of `ac_fixed<16, 5, false>`

4.14 DBF Engine

The input is a array (Rx vector), whose elements support following input formats:

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

All input will be converted to data type of `cal_complex_ce<14, 1, true, 4, false>` first before further computation. The algorithm is exactly as Alps except the maximum input Rx vector is up-to 64.

4.14.1 List of Parameters and Outputs

Parameters:

- input_format: indicator of input format
- input_len: length of Rx vector
- peak_size: window size for peak search on DBF spectrum
- max_nobjs: maximum number of objects to output
- bfm_npoint: total size of steering vectors
- raw_search_step: step for raw searching DBF spectrum
- fine_search_range: searching range for fine searching DBF spectrum
- noise_level: noise estimate in format of `cal_float<15, 1, false, 5, false>`
- bfm_acc_hw: interpolation flag
- fft_mux: To mux out Rx vector further to form customized Rx vector
- noise_scalars: predefined SNR dependent scalars to adjust noise level, in format of `cal_float<6, 6, false, 4, false>`
- bfm_mode: 0: peak mode, 1: OMP mode
- peak_scalars: predefined SNR-dependent scalars to adjust peak power for comparison, in format of `cal_float<6, 6, false, 4, false>`.
- SNR thres: SNR thresholds to determine which peak_scalar and noise_scalar to use, in format of `ac_fixed<7, 7, false>`.
- steering vectors: 2D array and its elements are in format of `ac_complex<ac_fixed<14, 1, true>>`.

Outputs:

- number of objects and each object should have individual outputs of remaining output value.
- power: reflector's power in `cal_float<15, 1, false, 5, false>`
- beam index
- offset: interpolation results.

4.15 DML Engine

The input is a array (Rx vector), whose elements support following input formats:

1. `cal_complex_ce<14, 1, true, 4, false>`
2. `ac_complex<ac_fixed<29, 1, true>>`

All input will be converted to data type of `cal_complex_ce<14, 1, true, 4, false>` first before further computation. The algorithm is exactly as Alps except the maximum input Rx vector is up-to 64.

4.15.1 List of Parameters and Outputs

Parameters:

- input_format: indicator of input format
- input_len: length of Rx vector
- total_ang_num: Total number of steering vectors
- refine_step: Fine searching steps
- refine_n:
- max_obj_num: 1: output 1 object, 2: output 2 objects, 0: Auto
- p1p2_en:
- extra_1ddml_en:
- coarse_start: starting point of coarse searching
- coarse_end: end point of coarse searching
- res_coef: 1D array and its elements are in format of `ac_fixed<14, 1, true>`
- noise_level: noise estimate in format of `cal_float<15, 1, false, 5, false>`
- A: 2D array and its elements are in `ac_complex<ac_fixed<14, 1, true>>`
- d: a predefined scalar in format of `cal_complex_ce<18, 1, true, 5>`
- k: a predefined scalar in format of `cal_complex_ce<18, 1, false, 5>`

Outputs:

- number of objects and each object should have individual outputs of remaining output value.
- power: reflector's power in `cal_float<15, 1, false, 5, false>`
- beam index