Ordered-Statistic CFAR high level synthesis implementation

Aleksei Rostov

Munich, 2022

Contents

1 Overview and background

The targets detection in RADAR signal processing is an essential part of a "secondary" processing algorithms prior making decision and/or a tracking algorithm. Detection itself it is a process of separation signal reflected from target and background noise caused by system's noise, clutters and interference. There are a lot of algorithms based on various approaches for signal from noise separation. Some of them based on linear algorithms, another one - on non-linear algorithms. In this lesson non-linear algorithms for target detection was considered, since non-linear algorithm is very interesting from point of FPGA implementation. The main problem of the signal detection in the RADARs it is a changing background of noise, clutter and interference (Fig. 1.1).

Signal detection occurs after comparison with a threshold. In case when *constant* threshold is used, many false targets due to changing background can be detected. Avoiding false targets detection can be done by applying adaptive threshold based on predetermined constant probability of false alarm (PFA). For adaptive threshold calculation Constant False Alarm Rate (CFAR) detectors are used.

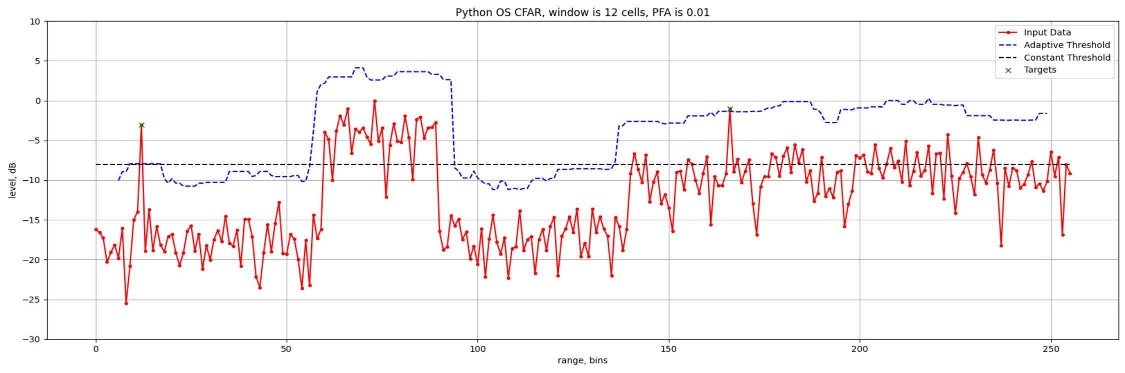


Figure 1.1: Simulation of constant and adaptive threshold

The generic structure of CFAR detector is depicted on Fig.1.2. There are two sliding windows with reference and guard (may be omitted) cells, the Cell-Under-Test between, processor with Linear or Nonlinear operation and multiplier with comparator. Adaptive Threshold for comparator depends on value Z (calculated by processor unit) and coefficient α (determined by desired probability of false alarm).

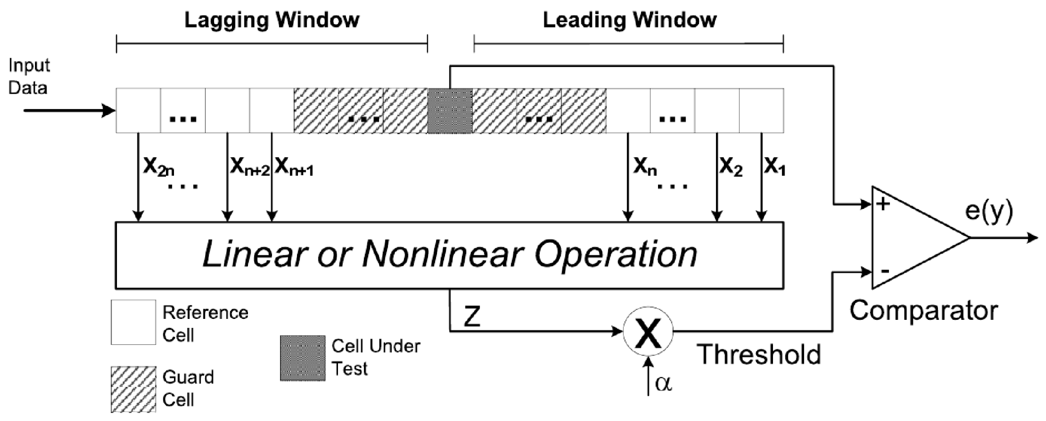


Figure 1.2: Generic structure of CFAR detector

The comparator compares Cell-Under-Test and Threshold after the processor unit. The processor unit can implement different algorithms for calculation Threshold (Cell-Averaging, Smallest-Of-Cell-Averaging and others). Here is considered only nonlinear Ordered-Statistic algorithms (OS CFAR).The principle of OS CFAR is very simple: all samples from the sliding windows are sorted in ascending order and then *k-th* sample (*order statistic*) is chosen for calculating Threshold (Fig. 1.3).

The most important parameter for OS-CFAR is the scaling factor α, which depends on

$$X_1 \le X_2 \le \cdots \le \underbrace{X_k}_{\boxed{\text{k}^{\text{th}}\text{ order statistic}}} \le \cdots \le X_n$$

Figure 1.3: Order statistic definition

required PFA and size of window. The required scaling factor may be defined by using the following formula:

$$PFA = \frac{N!(\alpha + N - kth)!}{(N - kth)!(\alpha + N)!}$$

where kth is a sample defined as $kth = 0.75N$, N - a size of the window. Example of the scaling factor selection is depicted on (Fig. 1.4)

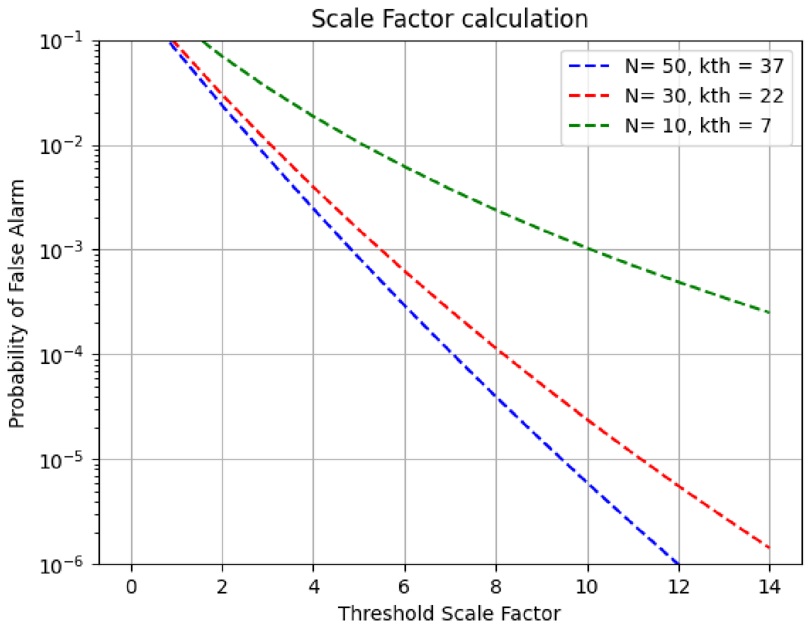


Figure 1.4: Scaling factor selection

The code for the scaling factor selection

```python
import matplotlib.pyplot as plt
import numpy as np
```

```
Niter        = 15

N            = 50
# k-th cell is 75 % from size of sliding window
KTH_CELL     = (N*75) // 100
# scaling factor calculating for Pfa
dPfa_0 = np.zeros(Niter)
for k in range(Niter):
    alpha = k + 1
    dPfa_0[k] = (np.math.factorial(N) * np.math.factorial(alpha + N - KTH_CELL)) /
         (np.math.factorial(N - KTH_CELL) * np.math.factorial(alpha + N))

N             = 30
# k-th cell is 75 % from size of sliding window
KTH_CELL      = (N*75) // 100
# scaling factor calculating for Pfa
dPfa_1 = np.zeros(Niter)
for k in range(Niter):
    alpha = k + 1
    dPfa_1[k] = (np.math.factorial(N) * np.math.factorial(alpha + N - KTH_CELL)) /
         (np.math.factorial(N - KTH_CELL) * np.math.factorial(alpha + N))


N             = 10
# k-th cell is 75 % from size of sliding window
KTH_CELL      = (N*75) // 100
# scaling factor calculating for Pfa
dPfa_2 = np.zeros(Niter)
for k in range(Niter):
    alpha = k + 1
    dPfa_2[k] = (np.math.factorial(N) * np.math.factorial(alpha + N - KTH_CELL)) /
         (np.math.factorial(N - KTH_CELL) * np.math.factorial(alpha + N))



plt.figure()
plt.semilogy(np.arange(Niter), dPfa_0, '--b', label='N=␣{},␣kth␣=␣{}'.format(50,
    (50*75) // 100))
plt.semilogy(np.arange(Niter), dPfa_1, '--r', label='N=␣{},␣kth␣=␣{}'.format(30,
    (30*75) // 100))
plt.semilogy(np.arange(Niter), dPfa_2, '--g', label='N=␣{},␣kth␣=␣{}'.format(10,
    (10*75) // 100))
plt.xlabel('Threshold␣Scale␣Factor')
plt.ylabel('Probability␣of␣False␣Alarm')
plt.title('Scale␣Factor␣calculation')
plt.legend()
plt.ylim((10**(-6), 10**(-1)))
plt.grid()

plt.show()
```

Summarizing the information above:

1. OS-CFAR implementation is based on *sorting* algorithm and sliding window
2. Scaling factor calculation depends on required PFA and size of the sliding window

2 HLS Implementation

High-Level-Synthesis implementation of OS-CFAR IP core is based on the following hardware design (Fig. 2.1)

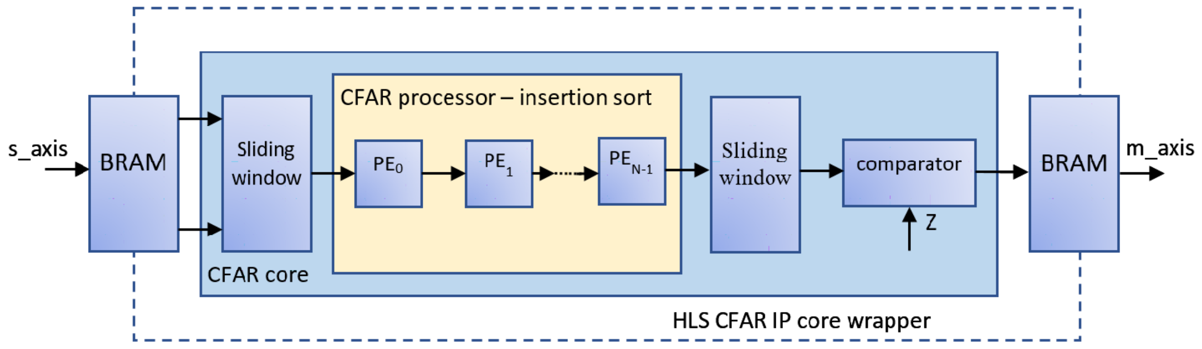


Figure 2.1: Hardware design of OS-CFAR IP core

The hardware design consists of the following main units:

1. Two 2-ports BRAMs for AXI STREAM compatibility
2. Two 2-ports BRAMs for two sliding windows implementation (to keep data before and after sorting)
3. CFAR processor with insertion sort algorithm
4. Simple comparator with multiplication by scale factor

Input BRAM is required for flushing in AXI STREAM data (s_axis). The sliding window slides across the input BRAM reading out data into the CFAR processor for sorting operation. The CFAR processor sorts data and writes one to output sliding window BRAM, where *k-th* sample are used for threshold calculation with scaling factor. The threshold is compared with the current Cell-Under-Test and decision is stored into output AXI STREAM BRAM. The steps with sliding window and threshold calculation is repeated until the end of the input sequence will not be reached. Afterwards the output AXI STREAM BRAM (m_axis) is flushed out.

For throughput improvement and resources optimization the following techniques were used:

- AXI STREAM interface with **HLS INTERFACE axis port** pragma

- Assigning specific memory type with **HLS BIND_STORAGE** pragma
- Assigning specific implementation of arithmetic operations with **HLS BIND_OP** pragma
- Loop Pipelining with **HLS PIPELINE** pragma
- Loop Unrolling with **HLS UNROLL** pragma
- Array Reshaping with **HLS ARRAY_RESHAPE** pragma

The AXI STREAM interface with 2-ports BRAMs is implemented as follows:

cfar_hls.cpp

```
...
void CFAR_TOP(stream<stream_1ch> &in_stream, stream<stream_1ch> &out_stream)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in_stream
#pragma HLS INTERFACE axis port=out_stream
        wrapper_cfar<uint32_t>(in_stream, out_stream);
}
...
```

cfar_hls.h

```
...
template <typename T>
void wrapper_cfar(stream<stream_1ch> &in_stream, stream<stream_1ch> &out_stream)
{
        T x[NPOINTS], y[NPOINTS];
#pragma HLS BIND_STORAGE variable=x type=ram_t2p impl=bram
#pragma HLS BIND_STORAGE variable=y type=ram_t2p impl=bram
        pop_input      <T>( in_stream, x);
        cfar           <T>(x, y);
        push_output <T>(out_stream, y);
}
...
```

Input/Output memory BRAM for AXI STREAM buffers are assigned to 2-ports BRAM types by *type=ram_t2p impl=bram*. Interface AXI STREAM is defined by stream_1ch data type in *cfar_hls.h* file

```
typedef ap_axiu<32, 0, 0, 0> stream_1ch;
```

The sliding window with sorting function and scaling is implemented in *cfar_hls.h* as well:

```
...
template <typename T>
void cfar(T x[NPOINTS], T y[NPOINTS])
{
        static T x_sort[REFWIND];
#pragma HLS BIND_STORAGE variable=x_sort type=ram_t2p impl=bram
```

```
        static T y_sort[REFWIND];
#pragma HLS BIND_STORAGE variable=y_sort type=ram_t2p impl=bram
        T tmp_0, tmp_1;
#pragma HLS BIND_OP variable=tmp_0 op=mul impl=dsp
                    // DSP may be omitted

        L0_CFAR:for (uint16_t idx =  0; idx < NPOINTS; idx ++)
                    // cannot be pipelined since has L1_CFAR and sort_insert<T> in
            a sequence
        {

                if(idx >=  REFWIND / 2 && idx < (NPOINTS - REFWIND / 2))
                {
                        L1_CFAR:for(char t1 = 0; t1 < REFWIND / 2; t1 ++)
                                    // pipelined loop (REFWIND / 2 clocks)
                        {
#pragma HLS PIPELINE
                                x_sort[t1] = x[idx - t1 - 1];
                                x_sort[t1 + REFWIND / 2] = x[idx + t1];
                        }

                        sort_insert<T>(x_sort, y_sort);

                        tmp_0           = ((T)Z_COEF  * y_sort[KTH_CELL]) >> 10;
                        y[idx]          = tmp_0;
                                                                        // NO
                            SCALING,  here the threshold itself instead of decision
                             is provided
                }
                else
                        y[idx] = 0;
        }
}
...
```

Input/Output memory BRAM for the sliding window are assigned to 2-ports BRAM types by *type=ram_t2p impl=bram*. The sliding window is being shifted in loop L0_CFAR, that can not be pipelined since there are operations read/write for sliding window itself, sorting algorithm and scaling with comparator. Loop L1_CFAR is implemented for flushing data from input BRAM to the sliding window. It is fully pipelined loop for 2-ports BRAM. Scaling or multiplication for getting threshold can be implemented either by using DSP blocks (*op=mul impl=dsp*) or FPGA's fabric.

Non-linear sorting algorithm in the CFAR processor is implemented by using systolic array with processing element (PE) units. Hardware design of a processing element is illustrated on Fig. 2.2

Every clock the processing element takes input *din* and compares one with the previous input stored in the register *reg*. If the input is greater than the stored value, the stored value is provided on the output and the input is stored to the register *reg*. The systolic array (chain of processing elements) will sort input sequence after passing one via the chain.

Regarding RTL implementation of the insertion sort algorithm required resources (number of PE) and latency should be considered. The number of processing elements in the

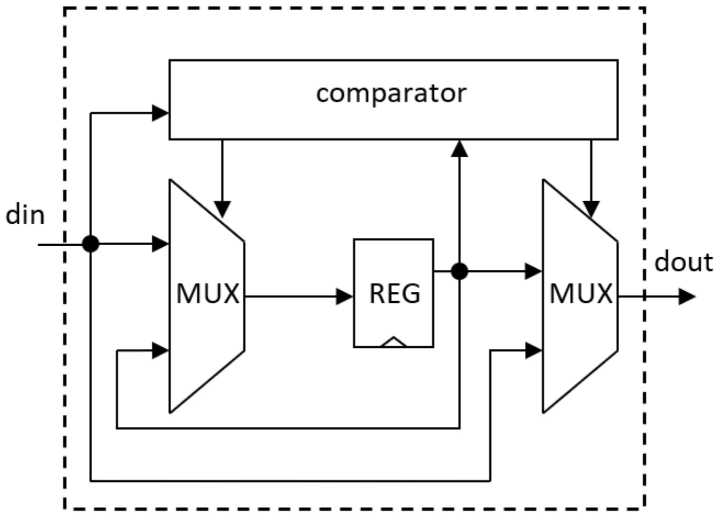


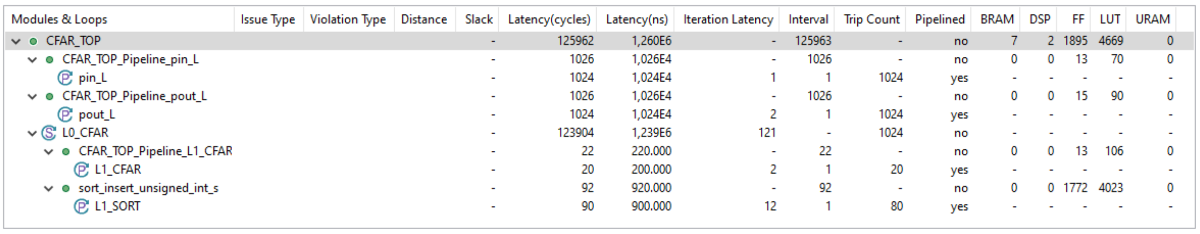
Figure 2.2: Hardware design of a systolic array processing element

systolic array is equal to the length of the input sequence. The last output will be available after number of clocks equal to two length of the input sequence. The insertion sort implementation in HLS is shown in the following listing:

```
template <typename T>
void sort_insert(T x[REFWIND], T y[REFWIND])
{
        static T d[REFWIND];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=d
        static T s[REFWIND];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=s
        L0_SORT:for(char idx = 0; idx < REFWIND; idx ++)
        {
#pragma HLS UNROLL
                d[idx] = 0;
                s[idx] = 0;
        }

        L1_SORT:for(T idx = 0; idx < 2*REFWIND; idx ++)
                            // pipelined loop (2*REFWIND clocks)
        {
                if(idx < REFWIND)
                        pe<T>(x[idx], &s[0], &d[0]);
                else
                        pe<T>(MAX_VALUE, &s[0], &d[0]);

                L2_SORT:for (T idx_2 = 0; idx_2 < REFWIND-1; idx_2 ++)
                            // fully unrolled (has REFWIND number of PE instances)
                        pe<T>(d[idx_2], &s[idx_2 + 1], &d[idx_2 + 1]);

                if(idx >= REFWIND)
                        y[idx - REFWIND] = d[REFWIND-1];
        } // for
}
```

The insertion sort implements the systolic array by using three *for* loops and four arrays (input, output and two internal). Internal arrays (*d[REFWIND], s[REFWIND]*) are implemented by using distributed memory resources, it provides multiple access to the array's elements, since a BRAM memory has only two ports. Input and Output arrays (or sliding window) are implemented by using 2-ports BRAM memory. The loops *L0_SORT* and *L2_SORT* are fully unrolled, since both of loops are used for multiply instances (elements of the systolic array) generation. The latency of the implementation will be in two times greater than number of clocks (REFWIND) for passing all inputs.

Performance and resources for OS-CFAR with parameters generated by *cfar_generator.py 1024 40 1e-2* can be seen from Fig. 2.3.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
CFAR_TOP				-	125962	1,260E6	-	125963	-	no	7	2	1895	4669	0
CFAR_TOP_Pipeline_pin_L				-	1026	1,026E4	-	1026	-	no	0	0	13	70	0
pin_L				-	1024	1,024E4	1	1	1024	yes	-	-	-	-	-
CFAR_TOP_Pipeline_pout_L				-	1026	1,026E4	-	1026	-	no	0	0	15	90	0
pout_L				-	1024	1,024E4	2	1	1024	yes	-	-	-	-	-
L0_CFAR				-	123904	1,239E6	121	-	1024	no	-	-	-	-	-
CFAR_TOP_Pipeline_L1_CFAR				-	22	220.000	-	22	-	no	0	0	13	106	0
L1_CFAR				-	20	200.000	2	1	20	yes	-	-	-	-	-
sort_insert_unsigned_int_s				-	92	920.000	-	92	-	no	0	0	1772	4023	0
L1_SORT				-	90	900.000	12	1	80	yes	-	-	-	-	-

Figure 2.3: OS-CFAR HLS performance and resource estimation

Performance or latency of the OS-CFAR implementation is defined by three main function and their *for loop*: *pop_input, cfar, push_output* or three main operations: reading in axis, cfar processing and reading out axis. Functions for axis interfacing take at least time requiring for passing entire sequence (around *NPOINTS* clocks in the example). CFAR itself (at loop *L0_CFAR*) requires *NPOINTS* intervals with iteration latency approximately four *REFWIND* clocks (121 in the example). The iteration latency during *L0_CFAR* is caused by two folded *for loop*: *L1_CFAR* (used by sliding window) and *L1_SORT* (passing data across systolic array, which is implemented with fully unrolled *L2_SORT*).

Resources required for CFAR IP core implementation are: 2 DSP blocks for multiplication with scaling factor (can be implemented on fabric logic) and seven 2-ports BRAM for the input / output sequence itself and the sliding window.

3 HLS verification

For HLS verification two Python scripts were created:: *cfar_generator.py* and *cfar_model.py*. The first script generates a synthetic signal for HLS Co-Simulation and header *parameters.h* file for initialization (Fig. 3.1).

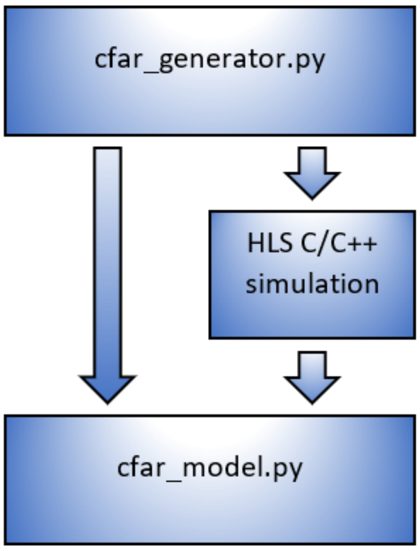


Figure 3.1: Scheme for CFAR simulation and HLS output verification

The script may be launched with, for example,the following arguments

```
python3 cfar_generator.py --h

Input signal size and CFAR parameters

positional arguments:
  NPOINTS      Number of points
  REFWIND      Number of cells in CFAR window
  PFA          Probability of false alarm

python3 cfar_generator.py 1024 40 1e-2
```

The arguments are the synthetic signal consists of 1024 samples, CFAR window size is 40 and Pfa is 1e-2.

The script will generate several files:

```
cfarIn_float.txt
cfarIn_u16.txt
cfarPy_param.txt
parameters.h
```

Files *cfarIn_float.txt* and *cfarIn_u16.txt* are float and unsigned 16-bits representation of the input data for HLS Co-Simulation and cfar_model. In the header *parameters.h* are defined the following parameters: length of input array (NPOINTS), length of reference window (REFWIND), *kth* cell order (KTH_CELL) and scaled threshold coefficient (Z_COEF).

The *cfar_model.py* consists of OS CFAR implementation which is used for HLS Co-Simulation output validation. The OS CFAR model is implemented in *cfar_1d(s, GrdHalf, RefHalf, T, k_th, Type)* function.

```
def cfar_1d(s, GrdHalf, RefHalf, T, k_th, Type):
    """CFAR for one-dimentional array.

    Args:
    s (numpy.ndarray)    : Input array.
            GrdHalf(int): Half of guard window.
            RefHalf(int): Half of reference window.
                T(float): Scaling factor for thresholding.
               k_th(int): cell for ordered-statistic
            Type(String): OS - ordered statistic
                          CA - cell-averaging
                          GO - greatest-of
                          SO - smallest-of

    Returns:
        s_cfar(numpy.ndarray): output array of thresholding.
        t_cfar(numpy.ndarray): output array of detection.
    """
    if s.ndim > 1:
        print('numpy.ndarray␣error')
        return -1
    N                   = np.size(s)
    s_cfar              = np.empty(N)
    t_cfar              = np.zeros(N)
    s_cfar[:]    = np.nan
    ref_win      = np.zeros(2*RefHalf)

    for idx in range(GrdHalf+RefHalf, N-(GrdHalf+RefHalf)):
        ref_win[0:RefHalf] = s[idx-(GrdHalf + RefHalf):idx-GrdHalf]
        ref_win[RefHalf :] = s[idx+ GrdHalf + 1: idx + GrdHalf + RefHalf + 1]
        ref_win            = sort_insertion(ref_win)
        Z                          = T*ref_win[k_th]

        s_cfar[idx + 0]  = Z
        t_cfar[idx + 1]  = comp_geb(Z, s[idx + 1])
    return s_cfar, t_cfar
```

The *sort_insertion(x)* function implements sorting algorithm by using Insertion sort which is good for FPGA implementation by using network based on Systolic Array.

The *cfar_model.py* will produce two plots: output of float-point Python OS CFAR implementation and output of HLS Co-Simulation (Fig. 3.2).

The first plot shows OS CFAR output without rounding of input data and threshold. Two targets were detected. The second plot shows outputs adjusted for 16-bit unsigned register. First output (blue line) is the output from Python OS CFAR model, that operates

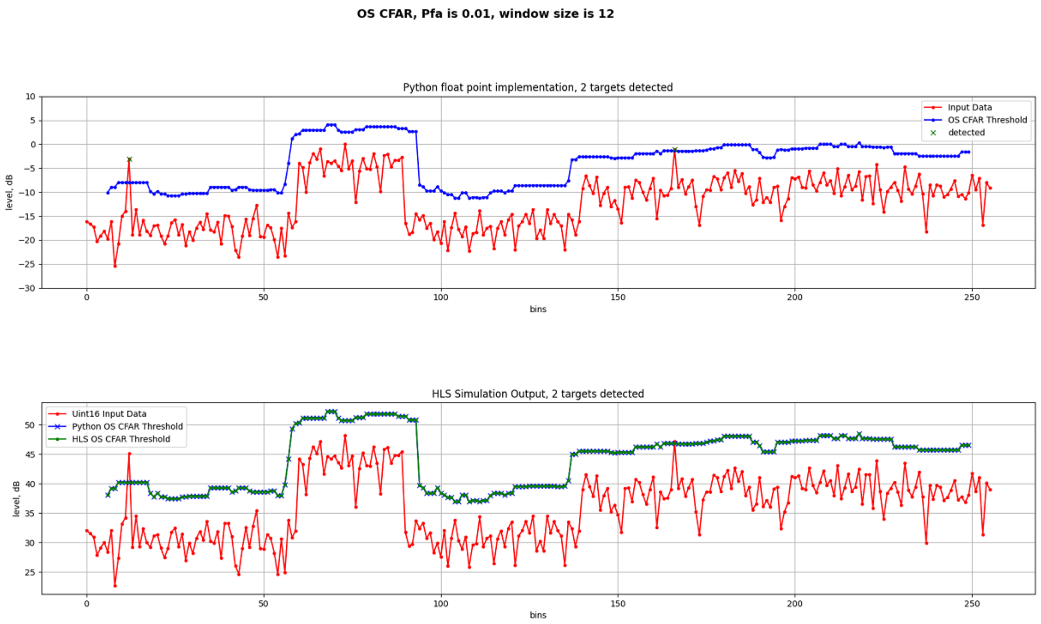


Figure 3.2: Comparison of Python CFAR implementation and HLS Co-Simulation

with 16-bit unsigned input (red line) and threshold also adjusted for 16-bit unsigned value. The second output (green line) is HLS Co-Simulation threshold. Since both Python OS CFAR and HLS implementation operates with the same input and threshold, their outputs are the same as well.

4 References

1. Mark A. Richards - Fundamentals of RADAR Signal Processing - 2005
2. Insertion sort Wikipedia
3. Systolic array Wikipedia