Fast Fourier Transform high level synthesis implementation

Aleksei Rostov

Munich, 2022

Contents

1 Overview and background

Discrete Fourier Transform (DFT) is the basic transform in RADAR signal processing for variety applications. For example, the applications are not only for signal from time to frequency transformation (Doppler processing, Imaging RADARs), but for speeding up some digital signal processing (DSP) algorithms like convolution in frequency domain. The last application requires optimized version of DFT - Fast Fourier Transform (FFT).

The current lesson covers some mathematical background and detailed explanation of implementation FFT in FPGA by using High Level Synthesis Tool. Despite the fact that any FPGA vendor very often has its own FFT IP core, the lesson can be used as a start point for implementation similar algorithms (filtering, convolution), customization FFT for certain FPGA project (for ex. 2 points per clock calculation) or developing IP cores by using HLS advantages.

Discrete Fourier Transform of the finite-length x_n sequence of length N is

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, \qquad k = 0, 1, ..., N-1 \tag{1.1}$$

where $W_N^{kn} = e^{-j(2\pi kn/N)}$.

Fast Fourier Transform (FFT) is *exactly* the same DFT with optimization by reducing number of computations. Algorithm FFT *decimation-in-time* is based on decomposition the input sequence x_n into smaller sub-sequences. For example, Eq. 1.1 can be separated into even- and odd-numbered points

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} e^{-j(2\pi k 2n/N)} + \sum_{n=0}^{(N/2)-1} x_{2n+1} e^{-j(2\pi k(2n+1)/N)} \tag{1.2}$$

with substitution

1. $e^{-j(2\pi k2n/N)} = e^{-j(2\pi kn/(N/2))}$
2. $e^{-j(2\pi k(2n+1)/N)} = e^{-j(2\pi k/N)} e^{-j(2\pi kn/(N/2))}$

we obtain

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} e^{-j(2\pi kn/(N/2))} + e^{-j(2\pi k/N)} \sum_{n=0}^{(N/2)-1} x_{2n+1} e^{-j(2\pi kn/(N/2))} \tag{1.3}$$

or with $W_{N/2}^{kn} = e^{-j(2\pi kn/(N/2))}$ and $W_N^k = e^{-j(2\pi k/N)}$

$$X_k = \sum_{n=0}^{(N/2)-1} x_{2n} W_{N/2}^{kn} + W_N^k \sum_{n=0}^{(N/2)-1} x_{2n+1} W_{N/2}^{kn} \tag{1.4}$$

Both sums in Eq. 1.4 could be rewritten into

$$X_k = G_k + W_N^k H_k \tag{1.5}$$

where G_k is $(N/2)$ - point DFT of the even sequence x_{2n} in the first sum and H_k is $(N/2)$ - point DFT of the odd sequence x_{2n+1} in the second sum.

Eq. 1.5 means, that for computation X_k it is necessary to calculate DFT of two sub-sequences - the even- and odd-numbered points of x_n. Furthermore G_k and H_k $(N/2)$ - points DFT can be decomposed in sum of two $(N/4)$ - points DFT, then in sum of two $(N/8)$ - point DFT and so on. Let's consider an example of 4-points DFT with following transformations of coefficients:

1. $W_N^0 = e^{-j2\pi 0/N} = 1$

2. $W_N^{N/2} = e^{-j2\pi(N/2)/N} = e^{-j\pi} = -1$

3. $W_{N/2}^{nk} = e^{-j2\pi nk/(N/2)} = e^{-j2\pi 2nk/N} = W_N^{2nk}$

4. $W_N^{N/2+k} = -W_N^k$

Let's consider an example for 4-point DFT. With Eq. 1.4 and 1.5 and coefficients transformation we obtain:

$$\begin{aligned}
X_0 &= x_0 + W_4^0 x_2 + W_4^0[x_1 + W_4^0 x_3] = x_0 + x_2 + W_4^0[x_1 + x_3] = x_0 + x_2 + x_1 + x_3\\
X_2 &= x_0 + W_4^0 x_2 + W_4^2[x_1 + W_4^0 x_3] = x_0 + x_2 - W_4^0[x_1 + x_3] = x_0 + x_2 - x_1 - x_3\\
X_1 &= x_0 + W_4^2 x_2 + W_4^1[x_1 + W_4^2 x_3] = x_0 - W_4^0 x_2 + W_4^1[x_1 - W_4^0 x_3] = x_0 - x_2 + W_4^1[x_1 - x_3]\\
X_3 &= x_0 + W_4^2 x_2 + W_4^3[x_1 + W_4^2 x_3] = x_0 - W_4^0 x_2 + W_4^3[x_1 - W_4^0 x_3] = x_0 - x_2 + W_4^3[x_1 - x_3]
\end{aligned}$$

The flow graph based on the computation of 4-points FFT is depicted on Fig. 1.1.

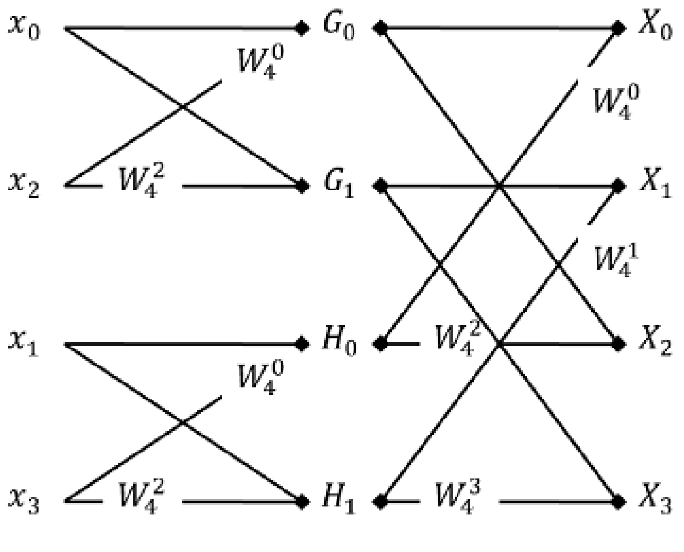


Figure 1.1: The flow graph of decomposition and computation of 4-point DFT

In the example above can be seen that only half of the coefficients are used in the output computation (W_4^1 and W_4^3) and there are only two multiplications are required. All coefficients were reorganized according transformation principle described above. Advantage of the principle will be shown further. Furthermore by applying the same principle as was

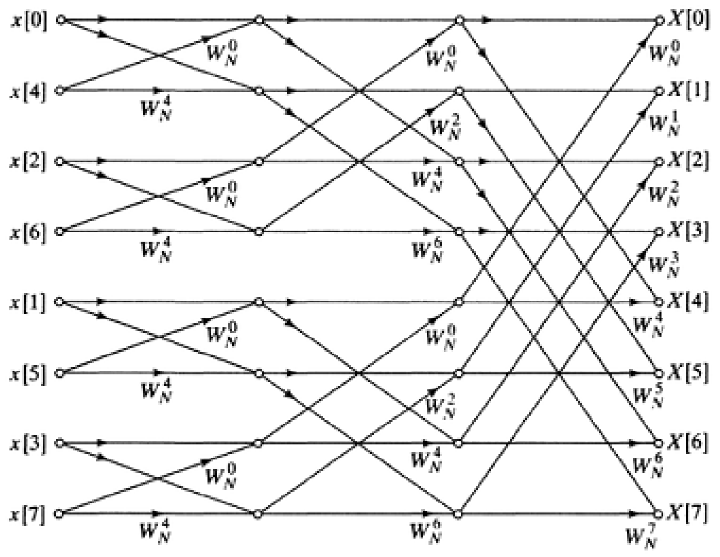


Figure 1.2: The flow graph of decomposition and computation of 8-point DFT

done for 4-point DFT computation for 8-point DFT the flow graph depicted on Fig. 1.2 can be obtained.

From flow graphs on Fig. 1.1 and 1.2 it can be concluded that:

- decomposition of the input sequence and coefficients transformation are essential part of FFT computation
- the basic elements of FFT are butterflies and stages
- number of butterflies and stages depend on number of DFT points
- just one coefficient per butterfly is required

Decomposition of input sequence and coefficients transformation are the key optimization for obtaining FFT, that involves number of multiplications and additions is equal to $N \log_2 N$, what is significantly less then for direct implementation of DFT N^2 and the reduction extremely grows up with the increasing number of DFT points.

The basic elements of FFT is a *stages*, which consists of set of 2-points DFT or *butterfly*. Number of stages and butterflies depends on amount of FFT points.

Associated equations for a butterfly from Fig. 1.3

$$X_m[p] = X_{m-1}[p] + W_N^r X_{m-1}[q]$$
$$X_m[q] = X_{m-1}[p] - W_N^r X_{m-1}[q]$$

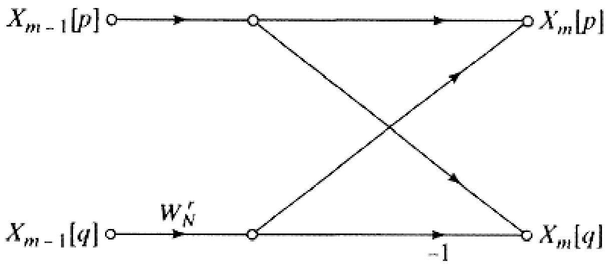


Figure 1.3: The flow graph of basic butterfly computation

The butterfly requires only one complex multiplication $W_N^r X_{m-1}[q]$ and overall there are $N \log_2 N$ multiplication for FFT computation. For example, 8-point FFT requires 24 complex multipliers as can be seen from Fig. 1.2. Apart of butterfly calculation it is necessary to reorder input data for the first stage. Reordering involves *bit-reversal* algorithm, when every bit of input data index binary form is reversed. For example, indexes for $N = 8$ will be reordered like:

Normal	Binary form of normal	Binary form of reserved	Reserved
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

A input data reordering it is an essential part of a FFT algorithm and should be applied either on the input (decimation-in-time) or on the output (decimation-in-frequency).

Summarizing the information above, FFT implementation requires (decimation-in-time Cooley–Tukey FFT algorithm):

1. set of complex pre-computed coefficients (only half of FFT points)
2. decomposition of the input sequence on odd- and even ordered sub-sequences
3. butterfly for complex multiplication on every stage

2 HLS Implementation

High-Level-Synthesis implementation of FFT IP core is based on the following hardware design

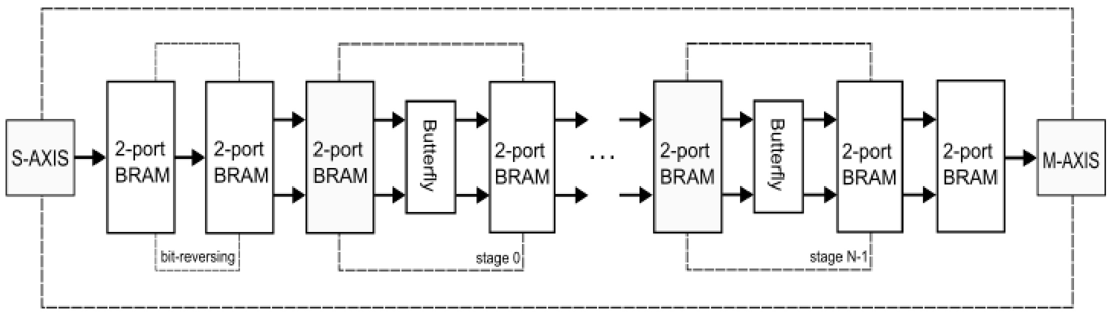


Figure 2.1: Hardware design of FFT IP core

The hardware design consists of the following main units

1. AXI STREAM input/output interfaces
2. One bit-reversing stage
3. Sequence of computation stages

Input/Output interfaces are fully AXI STREAM compliant, bit-reversing stage in the beginning implements reordering of the input sequence, the computation stages are used for further calculation.

The hardware design has the straightforward dataflow. The input sequence is flushed into BRAM via AXI STREAM slave interface (S-AXIS) and stored into 2-ports BRAM in bit-reversal order (bit-reversing stage). Afterwards data are passed stage by stage via computation stages and their butterflies. After the last computation stage data are stored in the output BRAM and are ready for flushing out via AXI STREAM master interface (M-AXIS).

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
- Function inlining with **HLS INLINE** pragma

- Loop Pipelining with **HLS PIPELINE** pragma
- Loop Unrolling with **HLS UNROLL** pragma
- Array Reshaping with **HLS ARRAY_PARTITION** pragma
- Task-level pipelining with **HLS DATAFLOW** pragma

Bit-reversing

Bit-reversing algorithm implements data reading from input BRAM and storing to another one in bit-reversal order.

```
/**
 * Bit reversal operation
 *
 * @param       Addr            U-bits address for reversal
 * @return      reversal        address
 *
 */
template <typename T>
T revBits(T Addr)
{
#pragma HLS INLINE
        T revAddr = 0;
        rb_L:for(uint8_t idx = 0; idx < FFTRADIX; idx ++)
        {
                revAddr <<= 1;
                revAddr  |= Addr & 1;
                Addr    >>= 1;
        } // rb_L

        return revAddr;
} // revBits


/**
 * Pre-processing stage
 *
 * @param       x               input  data array
 * @return      y               output data array with bit-revesed indexes
 *
 */

template <typename T>
void reverse_stage(T x[NPOINTS], T y[NPOINTS])
{
        T                       temp = 0;
        uint16_t        idx_r= 0;

        revst_L:for(uint16_t idx_d = 0; idx_d < NPOINTS; idx_d ++)
        {
                idx_r   = revBits<uint16_t>(idx_d);
                y[idx_r]= x[idx_d];
        } // revst_L
} // reverse_stage
```

Here the directive **HLS INLINE** for *T revBits(T Addr)* function was applied. The directive removes the function as separate entity in the HLS hierarchy - in other words, the directive builds the function into the *void reverse_stage(T x[NPOINTS], T y[NPOINTS])* and represents both functions as one entity. Two BRAM blocks are involved in the pre-processing stage as interfaces.

Butterfly

The Butterfly implements 2-points decimation-in-time DFT algorithm. The algorithm uses complex multiplier, addition and subtraction arithmetic operation.

```
/**
 *  butterfly dit (decimation-in-time) implementation
 *
 *  @param x_0 point in complex value converted in word cmpx_t<int16_t> ->
     uint32_t
 *  @param y_0 point in complex value converted in word cmpx_t<int16_t> ->
     uint32_t
 *  @param w_0 coeff in complex value converted in word cmpx_t<int16_t> ->
     uint32_t
 *  @return x_1, y_1 points in complex value converted in word cmpx_t<int16_t> ->
     uint32_t
 *            ------             -----
 *  x_0---->| z^-2 |--+-----+->|  +  |------> x_1
 *            ------    \    /    -----
 *                       \ /
 *                        X
 *            ------    /  \      -----
 *  y_0---->| cmpx |--+----+-->|  -  |------> y_1
 *            -+----              -----
 *            /
 *  w_0---->
 *
 *   x_1 = x_0 - y_0 * w_0
 *   y_1 = x_0 + y_0 * w_0
 */
template <typename T, typename U, typename V, uint8_t F>
void butter_dit(T x0, T y0, T w0, T *x1, T *y1)
{
#pragma HLS INLINE
        cmpx_t<V>   x_0 = {0, 0}, y_0 = {0, 0}, w_0 = {0, 0};

        uint2cmpx.uint = x0;
        x_0 = uint2cmpx.cmpx;

        uint2cmpx.uint = y0;
        y_0 = uint2cmpx.cmpx;

        uint2cmpx.uint = w0;
        w_0 = uint2cmpx.cmpx;
        cmpx_t<V>  x_1 = {0, 0};
        cmpx_t<V>  y_1 = {0, 0};

        cmpx_t<U> cmpx_mlt = {0, 0};
        cmpx_mlt.re_  = sum_pair<V, U>(y_0) * (U)w_0.re_ - sum_pair<V, U>(w_0) * (
            U)y_0.im_;
        cmpx_mlt.im_  = sum_pair<V, U>(y_0) * (U)w_0.re_ + sub_pair<V, U>(w_0) * (
            U)y_0.re_;

        cmpx_t<V> scaled_mlt = scl_pair<V, U, F>(cmpx_mlt);

        cmpx_t<U> dout_0 = cnv_pair<U, V>(x_0) + cnv_pair<U, V>(scaled_mlt);
        cmpx_t<U> dout_1 = cnv_pair<U, V>(x_0) - cnv_pair<U, V>(scaled_mlt);

        x_1 = scl_pair<V, U, 1>(dout_0);
        y_1 = scl_pair<V, U, 1>(dout_1);

        *x1 = (T &) x_1;
        *y1 = (T &) y_1;
} // butter_dit
```

The butterfly implementation is fully pipelined and requires 3 DSP blocks. The latency of the algorithm (when output is available after applying input) is 4 clocks. In addition butterfly's output is scaled in order to avoid overflowing.

Computation stages

Every stage involves input / output 2-ports BRAM interfaces and only one butterfly entity (this is due to throughput of 2-ports BRAMs). Two complex points are passed to butterfly's input with according coefficient and pushed out to the next BRAM.

```
/**
 *
 * FFT stage (reading array + butterfly processing)
 *
 * @param x[]   input array
 * @param w[]   input coefficients
 * @param casc  stage number
 *
 * @return y[]  output array
 *
 */

template <typename T, typename U, typename V>
void n_stage(T x[NPOINTS], T y[NPOINTS], uint8_t casc)
{
#pragma HLS BIND_STORAGE variable=wcoe type=rom_np impl=lutram latency=1
        T x0 = 0, y0 = 0;
        T x1 = 0, y1 = 0;
        nstage_L:for(uint16_t idx = 0; idx < NPOINTS / 2; idx ++)
        {
#pragma HLS PIPELINE
                uint16_t  d      = ((idx % POW2(casc)) == 0)  ? 2*idx : d + 1;
                uint16_t _idx1   = d + 0;
                uint16_t _idx2   = d + POW2(casc);
                T w0 = wcoe[W_IDX(idx, casc)];

                x0                = x[_idx1];
                y0                = x[_idx2];
                butter_dit<T, U, V, 15>(x0, y0, w0, &x1, &y1);
                y[_idx1] = x1;
                y[_idx2] = y1;
        } // nstage_L
} // n_stage
```

The implementation includes **HLS PIPELINE** directive, that provides ability to calculate 2-point DFT every clock. All coefficients are pre-computed and stored into ROM memory, that is implemented by using LUT resources.This ROM implementation is required since fully pipelined FFT implementation (Several stages should work simultaneously). The directive **HLS BIND_STORAGE** is applied to the global variable *wcoe*. All stages are combined together in one common function

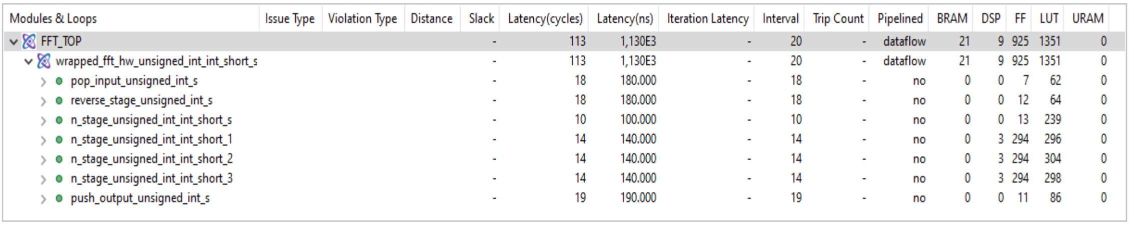
```
/**
 *  FFT CORE
 *
 */
template <typename T, typename U, typename V, int TU, int TI, int TD>
void wrapped_fft_hw (stream<stream_1ch> &in_stream, stream<stream_1ch> &out_stream
    )
{
#pragma HLS DATAFLOW
        T       mem_bram[FFTRAD_1][NPOINTS];
#pragma HLS ARRAY_PARTITION variable=mem_bram dim=1
#pragma HLS BIND_STORAGE variable=mem_bram type=ram_t2p impl=bram
        T       x[NPOINTS];
#pragma HLS BIND_STORAGE variable=x type=ram_t2p impl=bram


        pop_input  <T, TU, TI, TD, NPOINTS>( in_stream, x);
        reverse_stage<T>(x, mem_bram[0]);
        ffthw_L:for(uint8_t casc = 0; casc < FFTRADIX; casc ++)
#pragma HLS UNROLL
                        n_stage    <T,U,V>(  mem_bram[casc], mem_bram[casc + 1],
                            casc);

        push_output<T, TU, TI, TD, NPOINTS>(out_stream, mem_bram[FFTRADIX]);
} // wrapped_fft_hw
```

There are several important directives for improving throughput and reserving certain resources. For example, **HLS ARRAY_PARTITION** allows to divide 2 dimensional *mem_bram* array into several arrays or 2-ports BRAM blocks, **HLS UNROLL** is for unrolling *for* loop, since every FFT stage can be computed only in sequence, that's why pipelining here is not allowed. Directive **HLS DATAFLOW** plays significant role in order to provide pipelining on task-level and increasing the overall throughput. In other words, the directive arranges all stages for providing computation as soon as possible. For example, every stage can accept new data after the last two points were read out to butterfly.

HLS Summary for Synthesis and Co-Simulation of 16-points FFT is given below. There are $\log_2 N$ or 4 stages of FFT for 16 points. All stages and functions are working sequentially. Because of **HLS DATAFLOW** directive each function is ready to accept new data after processing of previous data is completed. Thus fully pipelined design is provided. Resource and timings estimation for the design is given on Fig. 2.2



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FFT_TOP				-	113	1,130E3	-	20	-	dataflow	21	9	925	1351	0
wrapped_fft_hw_unsigned_int_int_short_s				-	113	1,130E3	-	20	-	dataflow	21	9	925	1351	0
pop_input_unsigned_int_s				-	18	180.000	-	18	-	no	0	0	7	62	0
reverse_stage_unsigned_int_s				-	18	180.000	-	18	-	no	0	0	12	64	0
n_stage_unsigned_int_int_short_s				-	10	100.000	-	10	-	no	0	0	13	239	0
n_stage_unsigned_int_int_short_1				-	14	140.000	-	14	-	no	0	3	294	296	0
n_stage_unsigned_int_int_short_2				-	14	140.000	-	14	-	no	0	3	294	304	0
n_stage_unsigned_int_int_short_3				-	14	140.000	-	14	-	no	0	3	294	298	0
push_output_unsigned_int_s				-	19	190.000	-	19	-	no	0	0	11	86	0

Figure 2.2: 16-points FFT HLS synthesis report

The Hardware Design utilizes 21 block BRAM and 9 DSP. BRAM utilization is caused by FFT stage implementation and **HLS DATAFLOW** directive. Each stage except first one (since multiplication is replaced by inversion) utilizes 3 DSP blocks for complex multiplication implementation. BRAM blocks are implemented with a feature *ram_t2p* for using 2-ports with read / write capability on both. *Latency* means when the last output will be provided after first input (113 clock cycles in the example). *Interval* timing shows when the algorithm will be ready to accept a new data.

3 HLS verification

For HLS verification two Python scripts were created: *fft_generator.py* and *fft_model.py*. First one is used for generating test files and header *parameters.h* for HLS FFT IP core initialization (Fig. 3.1). Second one is used for HLS testbench output verification during Co-Simulation. The script compares and plots three outputs: HLS testbench, Numpy FFT and python FFT decimation-in-time implementation scaled to integer signed register.

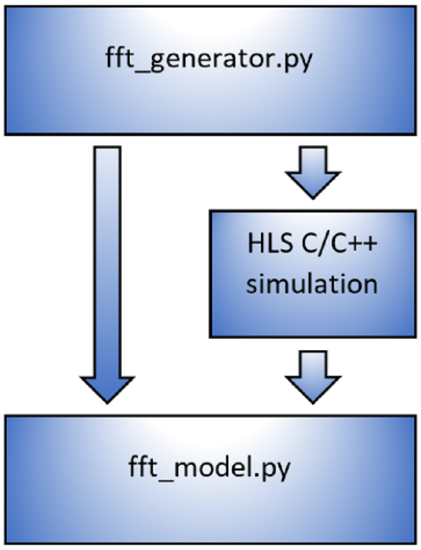


Figure 3.1: Scheme for FFT simulation and HLS output verification

Before starting HLS project test files should be generated. For example *fft_generator.py* script can be launched with the following arguments

```
python3 fft_generator.py 1024 3 30 -50
```

where - the synthetic signal consist of 1024 samples and is composed from 3 signals with Signal-To-Noise ratio 30 dB, 20 dB, 10 dB and noise floor equal to -50 dB.

The arguments in the script are *Npoints* is a number of FFT points, *Nsignals* - is an amount of harmonics with random frequencies in the synthetic signal and *SNR_dB* - is a max value of Signal-To-Noise ratio of the first harmonic in the synthetic signal, SNR of other harmonics will be decreased on 10 dB for every one, *Noise_dB* - noise floor.

The complex synthetic signal is represented as follows

```
...

for k in range(Nsignals):
    nbin = Npoints // 8 * (k + 1)
    x   += 10**((SNR_q + SNR_dB - 10*k)/20) * np.exp(2*1j*np.pi*nbin*np.arange(
        Npoints)/Npoints) + 10**(SNR_q/20)*np.random.randn(Npoints) #

...
```

The signal is the sum of *Nsignals* harmonics (position of every is calculated by *nbin*) with amplitude *SNR_dB* for first one and -10 dB for every next one. Noise is calculated with a given noise floor (set by *Noise_dB*).

After calculation the amplitude of synthetic sum signal is scaled to *-1 ... +1* and *-32768 ... +32768* and samples are saved to files. The script will generate several files:

```
    nonscaled_re.txt, nonscaled_im.txt, scaled_re.txt,
    scaled_im.txt, parameters.h
```

Header *parameters.h* consist of FFT parameters and array of scaled coefficients. Files *nonscaled_re.txt*, *nonscaled_im.txt* are complex float point input for Numpy FFT implementation, that will be used for comparison HLS FFT implementation. Files *scaled_re.txt*, *scaled_im.txt* are scaled to 16-bit signed register complex input for HLS FFT. The *fft_model.py* python script consists of scaled to 16-bits signed register implementation of FFT and FFT from Numpy package. The script reads out *scaled_re.txt* and *nonscaled_im.txt* data, output *cmpx_hls.txt* from HLS Co-Simulation and plots results of three FFT.

Python FFT decimation-in-time implementation is based on decomposition DFT during computation and is implemented stage-by-stage with pre-computed coefficients.

```python
def fft_dit(x, w):
    """
    FFT decimation-in-time implementation
    @param  complex int16 x: input data
    @param  complex int16 w: twiddling coefficients
    @return complex int16 y: output data
    """
    Np = np.size(x)
    Ns = int(np.log2(Np))
    Y_int16 = np.zeros((Ns + 1, Np), dtype=complex)

    # reverse input
    for k in range(Np):
        Y_int16[0, k] = x[revBits(k, Ns)]


    for casc in range(Ns):
        d = 0
        for k in range(Np // 2):
            idx_w = int(np.mod(k, 2**casc))*2**(Ns - 1 - casc)
            if np.mod(k, 2**casc) == 0:
                d = 2*k
            idx_1 = d
            idx_2 = d + 2**casc
            d = d + 1
            Y_int16[casc + 1, idx_1], Y_int16[casc + 1, idx_2] = butter_time_int16
                (Y_int16[casc, idx_1], Y_int16[casc, idx_2], w[idx_w])
    return np.round(Y_int16[Ns, :])
def butter_time_int16(x, y, w):
    """
    Radix 2 butterfly implementation (decimation-in-time) with rounding to int16
        and scaling factor for output
    @param complex int16 x:  FFT complex point (sample)
    @param complex int16 y:  FFT complex point (sample)
    @param complex int16 w:  Complex coefficient
    @return x_t, y_t complex  int16 samples
    """
    y_w = np.round((y * w)/Ampl) # rounding back to 16 bits after multiplication
    y_t = x - y_w
    x_t = x + y_w
    return x_t/2, y_t/2
```

Scaling and rounding are implemented in order to fit to 16-bit signed register during computation. Result for *python3 fft_generator.py 256 3 30 -50* is depicted on Fig. 3.2

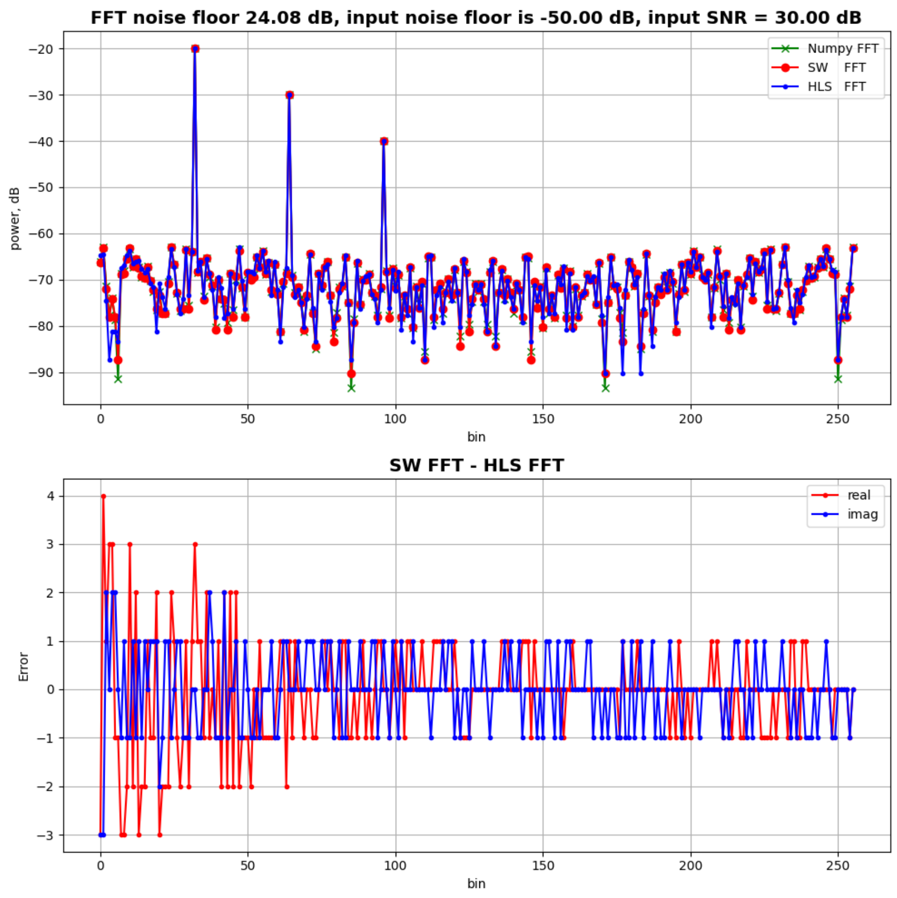


Figure 3.2: Results of Numpy, Python and HLS FFT implementation

First plot is a power of normalized FFT output implemented in Numpy (green), Python (red) and HLS Co-simulation (blue). Since Python and HLS implementation is scaled to 16-bit signed register, this noise floor is restricted by Root Means Square (RMS) value, that depends on length of a register. In case of 16-bit signed register, the minimum level of power spectrum is

$$SNR_{RMS} = 6.02 * 15 + 1.76 = 92.06dB$$

The second plot is an error between software implementation FFT and HLS C/C++ Co-Simulation output. The difference is minor and caused by rounding in Python and Vivado HLS. Despite the minor error the model can be used for Vivado HLS Co-simulation output verification.

4 References

1. Alan V. Oppenheim, Ronald W. Schafer, Discrete-Time Signal Processing, 3rd Edition, 2010
2. W. Kester, Understand SINAD, ENOB, SNR, THD, THD + N, and SFDR so You Don't Get Lost in the Noise Floor, Analog Devices, 2008
3. Vitis High-Level Synthesis User Guide UG1399
4. DFT matrix