**Any-Point DFT Overview**

**Introduction**

This document describes the hardware implementation of an algorithm for computing the Discrete Fourier Transform (DFT) for arbitrary numbers of points, including non-2^n points, using Verilog HDL. The algorithm efficiently and accurately approximates theoretical DFT results.

**Principle Explanation**

For 2^n point DFTs, the Fast Fourier Transform (FFT) is a well-known efficient method and will not be detailed here.

For non-2^n point DFTs, the algorithm first upsamples the input data to a suitable 2^n point format. This upsampled data can then be processed using a 2^n point FFT. After the FFT, the results are trimmed and compensated in the frequency domain to obtain the final non-2^n point DFT results.

The upsampling process involves low-pass filtering to prevent aliasing. A polyphase filter is used for this purpose. Since the upsampling from non-2^n points to 2^n points is not an integer multiple, interpolation is necessary. Linear interpolation is typically sufficient. It is important to note that after the non-2^n point input data ends, cyclic reconstruction must continue during upsampling instead of simply padding with zeros.

The FFT results for non-2^n point DFTs contain additional frequency band data, which can be discarded. The upsampling filter and interpolation process may distort the frequency domain data. To compensate for this distortion, the frequency response of the upsampling process is precomputed. The FFT results are then multiplied by the inverse of this frequency response to obtain accurate non-2^n point DFT results.

**Design Methodology**

The design process involves the following steps:

1. **Algorithm Design**: This phase typically uses tools like Matlab or Python. In this project, Matlab was chosen for its convenience in quickly testing various parameters and methods. The successful design is validated when the DFT results match those of Matlab's built-in fft function (with a relative error below a certain threshold, such as -60dB) and the structure is hardware-friendly.

FIR Low-pass Filter Design: A critical step is designing an FIR low-pass filter with ideal frequency response characteristics and a manageable number of taps for hardware implementation. The frequency response of the upsampling process is computed using Matlab to derive compensation coefficients.

After the algorithm design is completed, the tap coefficients of the low-pass filter and the frequency-domain compensation coefficients need to be saved in files for subsequent use in the C-model design.

1. **C-model Design**: After finalizing the algorithm, the functionality is implemented in C/C++. The C-model structure must align with the hardware design. A floating-point C-model is developed first and verified against the Matlab model. Then, a fixed-point C-model is created for RTL implementation, ensuring consistency with the floating-point version.
2. **Verilog HDL Implementation**: This includes RTL and Testbench development, referencing the fixed-point C-model.

**Usage Guide**

The project directory contains two main folders: "algorithm" and "lte\_dft". The "algorithm" folder contains Matlab scripts for exploring and optimizing any-point DFT algorithms. The "lte\_dft" folder contains the complete implementation from Matlab algorithm to C-model and Verilog RTL for the DFT used in LTE uplink SC-FDMA.

The main contents of the algorithm directory are as follows:

algorithm

├─ doc  *Documentation directory*  
└─ matlab *Directory containing algorithm source files*  
│ ├── amp\_func.m *Frequency compensation amplitude fitting function*  
│ ├── ang\_func.m *Frequency compensation phase fitting function*  
│ ├── comp\_factor\_fit.m *Calculates the fitting coefficients and outputs to a file*  
│ ├── cooley\_tukey *Verification of the Cooley-Tukey algorithm for 3\*2^n points FFT*  
│ │   ├── cooley\_tukey\_1st.m  
│ │   ├── cooley\_tukey.m  
│ │   └── cooley\_tukey\_main.m  
│ ├── dft\_fit\_main.m *Performs fitting comparison of frequency compensation coefficients*  
│ ├── dft\_freq\_fit.m *Frequency compensation fitting function*  
│ ├── dft\_main\_lut.m *Designs filters on the fly, calculates DFT for arbitrary points, and directly obtains frequency compensation coefficients from a lookup table*  
│ ├── filter\_design.m *Design of low-pass filters for upsampling*  
│ ├── fit\_any\_point\_dft.m *Uses designed filters to calculate DFT for arbitrary points, with frequency compensation coefficients obtained through fitting*  
│ ├── lut\_any\_point\_dft.m *Uses designed filters to calculate DFT for arbitrary points, with frequency compensation coefficients obtained from a lookup table (LUT)*  
│ ├── make\_comp\_lut.m *Generates a lookup table for frequency compensation coefficients*  
│ ├── preproc\_fft\_postproc.m *Main function for any-point DFT, including sample rate conversion, 2^n-point FFT, and output of trimmed frequency-domain data*  
│ ├── reference *Reference directory, whose subdirectories contain various attempts at Matlab algorithms for different scenarios*  
│ ├...  
│ │  
│ ├── sample\_rate\_conv.m *Sample rate conversion for upsampling non-2^n point inputs to 2^n point inputs*  
│ ├── src\_freq\_response.m *Attempts to calculate the frequency response of the upsampling process*

Special Notes:

(1) When you first start, you can ignore the parts about calculating frequency-domain compensation coefficients with fitting methods. Focus on these key files: preproc\_fft\_postproc.m, sample\_rate\_conv.m, and dft\_main\_lut.m.

(2) The calculation process simulates the communication process of OFDM signal transmission/reception. Here's the workflow: non-2^n point frequency-domain data → modulated via Matlab's ifft function (IDFT) → non-2^n point time-domain data → transformed with the custom DFT algorithm (demodulation) → restored non-2^n point frequency-domain data. If the input and restored frequency-domain data match, the DFT algorithm is valid. Readers can verify this by using Matlab's built-in fft function on the non-2^n point time-domain data to confirm the consistency.

(3) In digital filter design, frequency is a relative concept. A well-designed upsampling filter can be used for all non-2^n point DFT calculations, eliminating the need to design separate filters for each point count. This simplifies the sample rate converter circuit design.

The main contents of the lte\_dft directory are as follows:   
lte\_dft  
├── c\_model *C/C++ reference model directory*  
│   ├── fft\_r22 *2^n-point FFT reference model directory*  
│   │   ├── cpp *C/C++ source code directory*  
│   │   │   ├── fixed\_point *Fixed-point model source code directory*  
│   │   │   └── float\_point *Floating-point model source code directory*  
│   │   ├── data *Input/output data directory*  
│   │   └── matlab *Matlab source file directory for comparison, used to generate input data and compare results between C-Model and Matlab built-in functions*  
│   │    
│   ├── lte\_dft *LTE DFT reference model directory*  
│   │   ├── cpp *C/C++ source code directory*  
│   │   │   ├── fixed\_point *Fixed-point model source code directory*  
│   │   │   └── float\_point *Floating-point model source code directory*  
│   │   ├── data *Input/output data directory*  
│   │   └── matlab *Matlab source file directory for comparison, used to generate input data and compare results between C-Model and Matlab DFT model*  
│   └── reference *Reference directory*  
│   
├── doc *Reference documentation directory*  
│  
├── hdl *Verilog HDL design directory*  
│   ├── rtl *RTL design directory*  
│   │   ├── fft *2^n-point FFT RTL implementation directory*  
│   │   ├── header *Header file directory, defining relevant macros*  
│   │   ├── postproc *Post-processing unit, for frequency-domain data trimming and compensation to obtain final DFT results*  
│   │   ├── preproc *Pre-processing unit, for sample rate conversion*  
│   │   └── top *Top-module directory*  
│   │  
│   ├── sim *Simulation directory*  
│   │   ├── dft *Simulation for LTE DFT*  
│   │   └── fft *Simulation for 2^n-point FFT*  
│   │  
│   └── tb *Testbench directory*  
│   ├── dft *Testbench for LTE DFT*  
│   └── fft *Testbench for 2^n-point FFT*  
│  
└── matlab *Matlab reference model for LTE DFT*

Due to the project's modular design, some identical Matlab files appear in multiple directories to facilitate lower-module verification.

If you're only interested in algorithm design, focus on the algorithm directory.

To understand the entire design process from Matlab to RTL, you need to delve into the lte\_dft directory.

#### Disclaimer

This design is freely available for use without any cost.

However, the author assumes no responsibility for the results or any legal liabilities arising from its use.

Users must agree to these terms before using the design.