Ha Noi University of Science and Technology

School of Electronic

Report

Digital Design Using VHDL

Topic:

Design FFT/IFFT 128 points IP core

Instructor: **Dr. Vo Le Cuong**

Group: **4**

Dang Tran Nhat Minh 20193231

Chu Dinh Hai 20193214

Le Tri Trung Kien 20193225

Ha Noi, 1-2023

Preface

In order to revise the knowledge that we have learned at school as well as solving some realistic problems, we chose the project: “Design FFT/IFFT 128 points IP core”. Fast Fourier Transform is an algorithm for the effective Discrete Fourier Transform calculation. We will operate this system directly in FPGA using Verilog with some requirements that we will discuss in the next chapter.

During the process of the project, because of the limited time and unawareness, we are inevitable of mistakes. We hope that we can realize and apply some new ideas from the project from the instructor. This will make our project become more clearly.

Thank Dr. Vo Le Cuong very much for instructing and providing many important documents, give us a lot of chances in researching and finding new things in DFT.

Hanoi, 18th January 2023

TABLE OF CONTENTS

LIST OF FIGURES

DUTY ROSTER

TASKS	TEAM MEMBERS		
	Dang Tran Nhat Minh	Chu Dinh Hai	Le Tri Trung Kien
Part 1. First thought about the project			
Discover the FFT8 and ROTATOR128 block			x
Discover the FFT16 and CNORM block		x	
Discover the BUFRAM128 and CT128 block	x		
Part 2. Coding			
Wave running	x		
Result verification using Python	x		
Part 3. Finish the report and slide			
Write the report	x	x	x
Do the slide	x	x	x

CHAPTER 1. INTRODUCTION

In this chapter, we will introduce the idea, purpose of the project.

1.1 Overview

In our electronics and telecommunications industry, spectrum analysis such as energy spectrum, amplitude spectrum, phase spectrum of signals in general and spectrum of digital signals in particular in the frequency domain plays a very important role. It tells us how the frequency components contribute to the signal, how their energy is, how to use energy effectively…

From that we have a way to handle that signal appropriately. The problem is how to transform the digital signal from the time domain to the frequency domain to observe its spectrum. The simplest answer is to use the Discrete Fourier Transform (DFT).

DFT is used in many fields, it is used in speech processing, image processing,… It would not be an exaggeration to say that anything related to digital signal processing requires Fourier transforms.

However, the use of DFT has a problem, that is, the computation is relatively complicated when the data length to be calculated increases. But as we know an image file, or any signal, is usually quite long, so if you just calculate the DFT normally, the execution time will be very long and complicated, so it won't satisfied time requirements. Although the DFT machine produces good products, but the speed is too slow, the manufacturer will certainly not be satisfied at all. That is why the Fast Fourier Transform (FFT) algorithm was created.

1.2 Main idea of the project

The idea of the FFT algorithm is the divide – and – conquer technique. Instead of calculating the DFT for an entire signal with a large length, we will perform a DFT calculation for each smaller signal segment in that signal and then from the obtained result we calculate the DFT of the original signal to be calculated first.

FFT has a very important role:

- FFT has improved the speed and accuracy of digital signal processing.
- FFT opens up a very wide field of spectrum analysis: telecommunications, astronomy, geophysics management, medical diagnosis,…
- The FFT has rekindled the interests of many branches of mathematics that were previous fully exploited.
- FFT has laid the foundation for computing other transformations such as Walsh transform, Hamadard transform, Haar transform,…

⇒Idea: Center's goal is a FFT algorithm/architecture with the programmability necessary to meet the variety of functional FFT demands of future wireless and other signal processing applications.

So, our project of the FFT128 core architecture to explain its proper use. FFT 128 soft core is the unit to perform the FFT. It performs one dimensional 128 – complex point FFT. The data and coefficient widths are adjustable in the range 8 to 16.

1.3 Features

- 128 – point radix-8 FFT
- Forward and inverse FFT.
- Pipelined mode operation, each result is outputted in one clock cycle, the latent delay from input to output is equal to 310 clock cycles (440 clock cycles when the direct output data order), simultaneous loading/downloading supported.
- Input data, output data, and coefficient widths are parametrizable in range 8 to 16 and more.
- Two and three data buffers are selected.
- FFT for 10 bit data and coefficient width is calculated on Xilinx XC4SX35-12 FPGA at 215 MHz clock cycle, and on Xilinx XC5VLX30-3 FPGA at 260 MHz clock cycle, respectively.
- FFT unit for 10 bit data and coefficients, and 3 data buffers occupies 4147 CLB slices, 4 DSP48 blocks, and 5 kbit of RAM in Xilinx XC4SX35 FPGA, and 1254 CLB slices 4 DSP48E blocks, and 5 kbit of RAM in Xilinx XC5VLX30 FPGA.
- Overflow detectors of intermediate and resulting data are present.

- Two normalizing shifter stages provide the optimum data magnitude bandwidth.
- Structure can be configured in Xilinx, Altera, Actel, Lattice FPGA devices, and ASIC. Can be used in OFDM modems, software defined radio, multichannel coding, wideband spectrum analysis.

1.4 Requirement of the project

- Build Specification of the design
  - System modeling / design by using diagram, FSM, ASM, FSMD, ASMD,…
  - Describe the input / output signals of each module / block
- Implement Algorithm on C++ / Python
- Verify the RTL design
- Evaluate the results of the device when deployed on FPGA: by simulation, FPGA board, evaluate resource consumption.

CHAPTER 2. THEORETICAL BACKGROUND

This chapter will represent the theoretical background of Fourier Transform and brief explaination of IP core.

2.1. Discrete Fourier Transform

Discrete Fourier Transform (DFT) is a fundamental digital signal processing algorithm used in many applications, including frequency analysis and frequency domain processing. DFT is the decomposition of a sampled signal in terms of sinusoidal (complex exponential) components.

The symmetry and periodicity properties of the DFT are exploited to significantly lower its computational requirements. The resulting algorithms are known as Fast Fourier Transforms (FFTs).

2.2. Introduction to Fast Fourier Transform

The main purpose in digital signal processing is to reduce the requirements of hardware and increased the computation. The Fast Fourier Transform (FFT) is an algorithm which samples the signal over some space and splits them into frequency points. This method is used to get the DFT of the signal and it converts the signal into its frequency domain.

2.3. Intellectual property core (IP core)

An intellectual property core (IP core) is a functional block of logic or data used to make a field-programmable gate array (FPGA) or application-specific integrated circuit for a product.

Commonly used in semiconductors, an IP core is a reusable unit of logic or integrated circuit (IC) layout design. It is the IP of one party and may be licensed by others for use in their own ICs and semiconductors.

CHAPTER 3. SYSTEM DESIGN ON FPGA

In this chapter, we will discuss the core of the design.

3.1 Design Features

An 128 – point DFT computes a sequence x(n) of 128 complex – valued numbers given another sequence of data X(k) of length 128 according to the formula

$$X(k) = \sum_{n=0}^{127} x(n)\, e^{-j2\pi nk/128};\ \text{k=0 to 127}$$

To simplify the notation, the complex – valued phase factor $e^{-j2\pi nk/128}$ is usually defined as W_{128}^{n} where W_{128} = cos(2n/128) – jsin(2n/128). The FFT algorithms take advantage of calculations that the DFT requires. In an FFT implementation the real and imaginary components of W_nN are called twiddle factors.

In the processor FFT128 a mixed radix 8 and 16 FFT algorithm is used. It divides DFT into two smaller DFTs of the length 8 and 16:

$$X(k) = X(16r + s) = \sum_{m=0}^{15} W_{16}{}^{mr} W_{128}{}^{ms} \sum_{l=0}^{7} x(16l + m) W_8{}^{sl}$$

r=0 to 15, s=0 to 7

This formula shows that 128-point DFT is divided into two smaller 8- and 16-point DFTs.

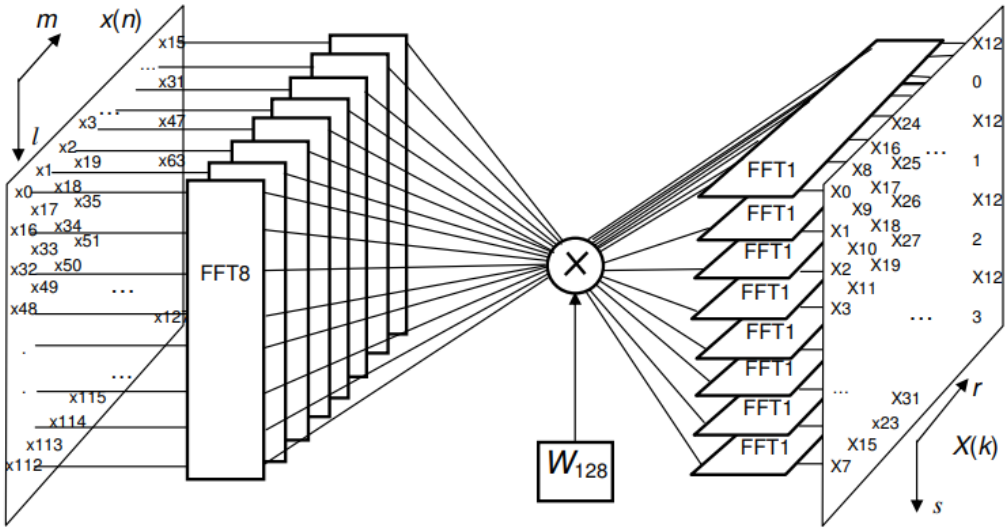

Figure 1 FFT128 illustration

According to Figure 1, the input complex data x(n) are represented by the 2-dimensional array of data x(16l+m). The columns of this array are computed by 8-point DFTs. The results of them are multiplied by the twiddle factors W_{128}^{ms}. And the resulting array of data X(16r+s) is derived by 16-point DFTs of rows of the intermediate result array.

3.1.1 Highly pipelined calculations

Each base FFT operation is computed by the datapaths called FFT8 and FFT16. FFT8 and FFT16 calculates the 8- and 16-point DFTs in the high pipelined mode. Therefore in each clock cycle one complex number is read from the input data buffer RAM and the complex result is written in the output buffer RAM. The 8- and 16-point DFT algorithm is divided into several stages which are implemented in the stages of the FFT8 and FFT16 pipelines. This supports the increasing the clock frequency up to 200 MHz and higher. The latent delay of the FFT8 unit from input of the first data to output of the first result is equal to 30 clock cycles. The latent delay of the FFT16 unit from input of the first data to output of the first result is equal to 30 clock cycles.

3.1.2 High precision computations

In the core the inner data bit width is higher to 4 digits than the input data bit width. The main error source is the result truncation after multiplication to the factors W_{64}^{ms}. Because the most of base FFT operation calculations are additions, they are calculated without errors. The FFT results have the data bit width which is higher in 3 digits than the input data bit width, which provides the high data range of results when the input data is the sinusoidal signal. The maximum result error is less than the 1 least significant bit of the input data. Besides, the normalizing shifters are attached to the outputs of FFT8 pipelines, which provide the proper bandwidth of the resulting data. The overflow detector outputs provide the opportunity to input the proper shift left bit number for these shifters.

3.1.3 Low hardware volume

The FFT128 processor has the minimum multiplier number which is equal to 4. This fact makes this core attractive to implement in ASIC. When configuring in Xilinx FPGA, these multipliers are implemented in 4 DSP48 units respectively. The customer can select the input data, output data, and coefficient widths which provide application dynamic range needs. This can minimize both logic hardware and memory volume.

3.2. Block diagram and functions of each block

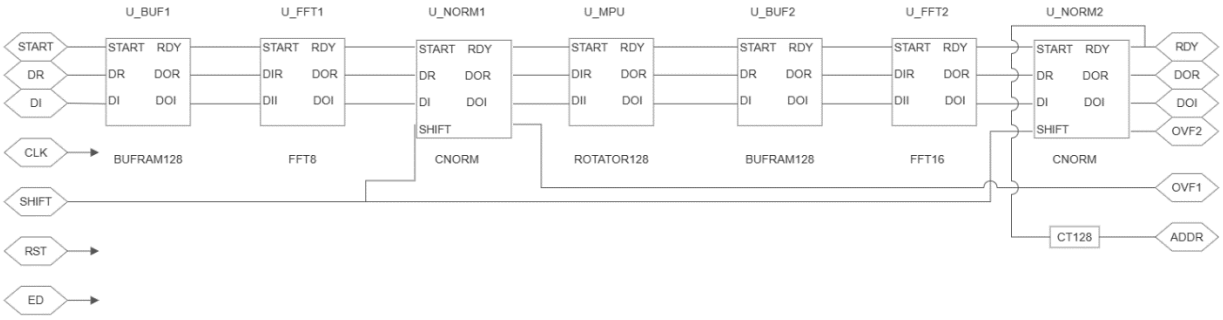

Figure 2 Block diagram of the FFT128 core with two data buffers

In this block diagram, there are some important components:

Block	Description	File described
BUFRAM128	Data buffer with row writing and column reading	BUFRAM128C.v RAM2x128C.v RAM128.v
FFT8	Datapath, which calculates the 8-point DFT	FFT8.v MPUC707.v
FFT16	Datapath, which calculates the 16-point DFT	FFT16.v MPUC707.v MPUC383.v MPUC1307.v MPUC541.v
CNORM	Shifter to 0, 1, 2, 3 bit left shift	CNORM.v
ROTATOR128	Complex multiplier with twiddle factor ROM	ROTATOR128.v WROM128.v
CT128	Counter modulo 128	

3.2.1. BUFRAM128

BUFRAM 128 is the data buffer consisting of the 2 port synchronous RAM of the volume 512 complex data, and the write-read address counter. The real and imaginary parts of the data are stored in the natural ascending order as in the diagram. By the START impulse the address counter is reset and then starts to count , which begins from the signal addrw. The input data DR and DI are stored to the respective address place by the rising edge of the clock signal.

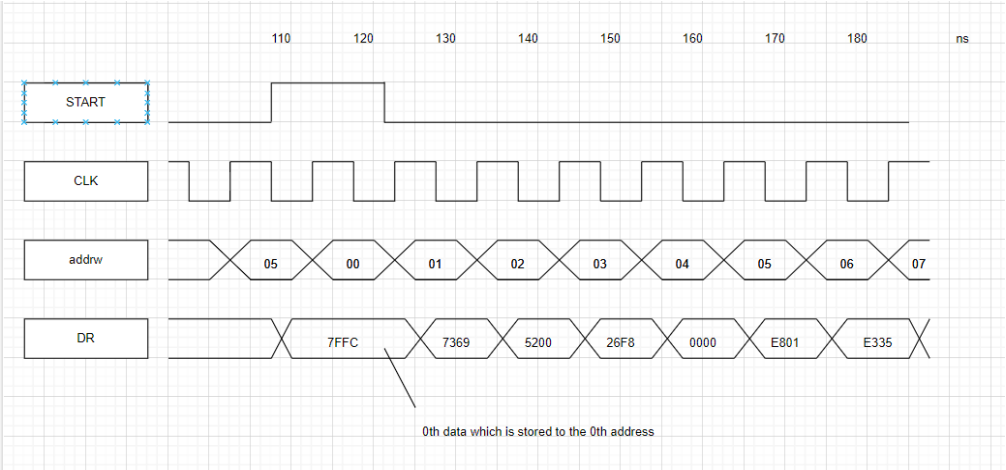

Figure 3 Data writing BUFRAM128 waveforms

After writing 128 data beginning at the START signal, the unit outputs the ready signal RDY and starts to write the next 128 data to the second half of the memory. At this period of time it outputs the data stored in the 1st half of the memory. When this data reading is accomplished, the reading of the next array is starting. This process is continued until the next START signal or RST signal are entered. The reading address sequence is 8-6th inverse order. The reading address is derived from the writing address by swapping 4 LSB and 4 MSB address bits.

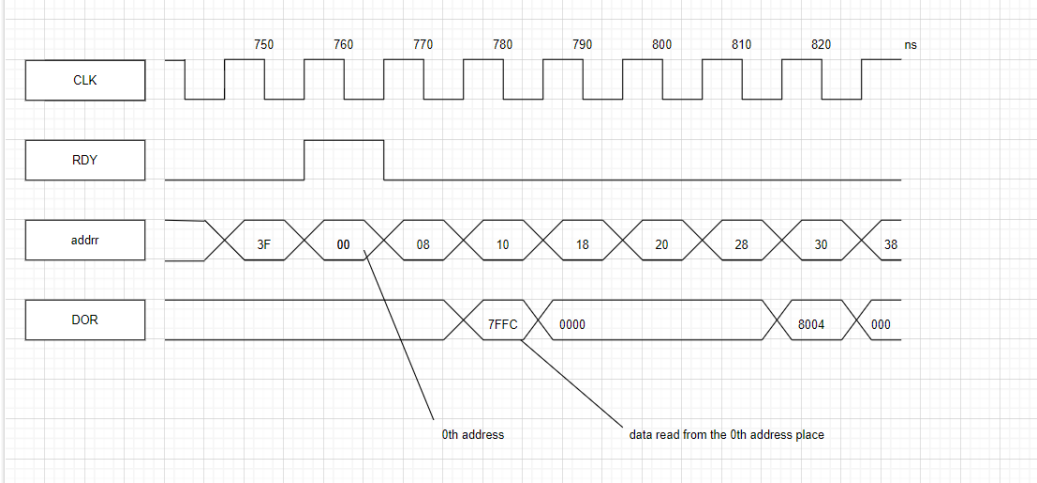

Figure 4 Data reading BUFRAM128 waveforms

There are 2 ways to implement the BUFRAM 128 unit. The 1st way consists the use of the usual one-port synchronous RAMs. Then BUFRAM128 consists of 2 parts, 1st one data array is stored into one part of the buffer, and another data array is read from the 2nd part of the buffer. Then these parts are substituted by each other. Such a BUFRAM 128 is implemented through BUFRAM128C.v as root model of the buffer, RAM2x128C.v – dual ported synchronous RAM and RAM128.v 0 single ported synchronous RAM model. This buffer is implemented when the FFT128bufferports1 parameter is recommended in the FFT128_config.inc file.

The 2nd way consists in use of the usual 2-port synchronous RAM with a single clock input. Such a RAM is usually instantiated as the BlockRAM or the dual ported Distributed Ram in the Xilinx FPGAs. In the case of FFT128bufferports1 parameter is commented or excluded in the FFT128_config.inc file. Then the file RAM128.v, which describes the simple model of the registered synchronous RAM is not used. The configuration in Xilinx FPGAs 2 or 3 BlockRAMs are instantiated depending on the parameter FFT128Parambuffers3.

3.2.2. FFT8

This is the first stage of FFT 128 processor. The datapath FFT8 implements the 8-point FFT algorithm in the pipelined mode. Calculations of this algorithm are:

t1=D(0) + D(4) t5=D(3) + D(7)

t2=D(6) + D(2) t6=D(3) – D(7)

t3=D(1) + D(5)	t7=t1 + t2
t4=D(1) – D(5)	t8=t5 + t3

m0=t7 + t8	m4=sin(π/4)*(t4-t6)
m1=t7 – t8	m5=j*(t5 – t3)
m2=t1 – t2	m6=j*(D(6) – D(2))
m3=D(0) – D(4)	m7=-j*sin(π/4)*(t4 + t6)

s1=m3 + m4	s3=m6 + m7
s2=m3 – m4	s4=m6 – m7

These are 22 clock cycles calculate for 8 input complex data.

DO(0) = m0	DO(4) = m1
DO(1) = s1 + s3	DO(5) = s2 + s4
DO(2) = m2 + m5	DO(6) = m2 – m5
DO(3) = s2 – s4	DO(7) = s1 – s3

These are 8 clock cycles output 8 complex results. The algorithm contains only 4 multiplications to the untrivial coefficient sin(π/4) = 0.7071. The multiplication to a coefficient j means the negation the imaginary part and swapping real and imaginary parts.

The FFT8 unit starts its operation by the START impulse. The first result is preceded by the RDY impulse which is delayed from the START impulse to 17 clock impulses.

3.2.3. FFT16

This is the second stage of FFT 128 processor. The datapath FFT16 implements the 16-point FFT algorithm in the pipelined mode. Calculations of this algorithm are:

t1:=x(0) + x(8);

t2:=x(4) + x(12);

t3:=x(2) + x(10);

t4:=x(2) – x(10);

t5:=x(6) + x(14);

t6:=x(6) – x(14);

t7:=x(1) + x(9);

t8:=x(1) – x(9)

t9:=x(3) + x(11)

t10:=x(3) – x(11)

t11:=x(5) + x(13)

t12:=x(5) – x(13)

t13:=x(7) + x(15);

t14:=x(7) – x(15);

t15:=t1 + t2

t16:=t3 + t5

t17:=t15 + t16

t18:=t7 + t11

t19:= t7 – t11

t20:=t9 + t13;

t21:= t9 – t13

t22:=t18 + t20

t23:=t8 + t14;

t24:= t8 – t14;

t25:=t12 + t10;

t26:= t12 – t10;

m0:=t17 + t22;

m1:=t17 – t22;

m2:= t15 – t16;

m3:= t1 – t2;

m4:=x(0) – x(8);

m5:= cos(π/4)*(t19 – t21);

m6:= cos(π/4)*(t4 – t6);

m7:= cos(3π/8)*(m24+m26);

m9:=– (cos(π/8) - cos(3π/8))*t26;

m10:= – j*(t18 – t20);

m11:= –j*(t3 – t5);

m12:= – j*(x(4)–x(12));

m13:= –j* sin(π/4)*(t19 + t21);

m14:=–j* sin(π/4)*(t4 + t6);

m15:= –j* sin(3π/8)*(t23 + t25);

m16:= –j* (sin(π/8) – sin(3π/8))*t23;

m8:= (cos(π/8) + cos(3π/8))*t24; m17:= –j*(sin(π/8) + sin(3π/8))*t25

s1:=m3 + m5;	s10:=s5 – s7
s2:=m3 – m5;	s11:=s6 + s8
s3:=m13 + m11;	s12:=s6 – s8
s4:=m13 – m11;	s13:=m12 + m14
s5:=m4 + m6;	s14:=m12 – m14
s6:=m4 – m6;	s15:= m15 – m16
s7:= m8 – m7	s16:= m15 – m17
s8:= m9 – m7	s17:=s13 + s15
s9:=s5 + s7;	s18:=s13 – s15
s19:=s14 + s16;	s20:=s14 – s16;

These are 46 clock cycles calculate for 16 input complex data.

y(0):=m0;	y(8):=m1;
y(1):=s9 + s17;	y(15):=s9 – s17;
y(2):=s1 + s3;	y(14):=s1 – s3;
y(3):=s12 – s20;	y(13):=s12 + s20;
y(4):=m2 + m10;	y(12):=m2 – m10;
y(5):=s11 + s19;	y(11):=s11 – s19;
y(6):=s2 + s4;	y(10):=s2 – s4;
y(7): =s10 – s18;	y(9):=s10 + s18;

These are 16 clock cycles output 16 complex results. The algorithm contains only 20 real multiplications to the untrivial coefficients:

$\sin(\pi/4) = 0.7071;$

$\sin(3\pi/8) = 0.9239;$

$\cos(3\pi/8) = 0.3827;$

$(\cos(\pi/8) + \cos(3\pi/8)) = 1.3066;$

$(\sin(\pi/8) - \sin(3\pi/8)) = 0.5412$ and 156 real additions and subtractions.

The counter ct counts the working clock cycles from 0 to 15. So a single inferred adder adds x(0) + x(8) in one cycle, x(1) + x(9) in the next cycle, D(1) + D(5) in another cycle and so on, and x(7) + x(15) in the final cycle of the sequence of cycles deriving the results t1,t7,t9,…,t13 respectively.

Four constant multipliers are used to derive the multiplication to 5 different coefficients. So the unit in MPUC707.v implements the multiplication to the coefficient 0.7071 in the pipelined manner. Note that the unit MPUC924_383.v implements the multiplication both to 0.9239 and to 0.3827. The multipliers use the adder tree, which adds the multiplicand shifted to different bit numbers. For example, for short input bit width the coefficient 0.7071 is approximated as 0.101101012, for long input bit width it is approximated as 0.1011010100000001012. The long coefficient bit width is set by the parameter FFT128bitwidth_coef_high. The first kind of the constant multiplier occupies 3 adders, and the second one occupies 4 adders.

The FFT16 unit implements both FFT and inverse FFT depending on the parameter FFT128paramifft. Practically the inverse FFT is implemented on the base of the direct FFT by the inversion of operations in the final stage of computations for all the results except y(0), y(8). For example, y(1):=s9 + s17; is substituted to y(1):=s9 – s17;

The FFT16 unit starts its operation by the START impulse. The first result is preceded by the RDY impulse which is delayed from the START impulse to 30 clock impulses. The output results have the bit width which is in 4 higher than the input data

bit width. That means that all the calculations except multiplication by coefficients like 0.7071 are implemented without truncations, and therefore, the FFT128 results have the minimized errors comparing to other FFT processors.

3.2.4. CNORM

During computations in FFT8 and FFT16 the data magnitude increases up to 8 and 16 times, respectively, and the FFT128 result can increase up to 128 times depending on the spectrum properties of the input signal. Therefore, to prevent the signal dynamic bandwidth loose, the output signal bit width must be at least in 8 bits higher than the input signal bit width. To prevent this bit width increase, to provide the proper signal dynamic bandwidth, and to ease the next computation of the derived spectrum, the CNORM units are attached to the outputs of the FFT16 units.

CNORM unit provides the data shift left to 0,1,2, and 3 bits depending on the code SHIFT. The input data width is nb+3 and the output data width is nb+2, where nb is the given processor input bit width.

The overflow occurs in CNORM unit when the SHIFT code is given too high. The SHIFT code must be set by the customer to prevent the data overflow and to provide the proper dynamic bandwidth. The CNORM unit contains the overflow detector with the output OVF. When FFT128 core in operation, a 1 at the output OVF signals that for some input data an overflow occurred. OVF flag is resetted by the RST or START signal.

The SHIFT inputs of two CNORM stages are concatenated to the 4-bit input SHIFT of the FFT128 core, 2 LSB bits control the first stage, and 2 MSB bits do the second stage.

The selection of the proper SHIFT code depends on the spectrum property of the input signal. When the input signal is the sinusoidal one or contains a few of sinusoids, and the noise level is small then SHIFT =0000, or 0001, or 0010. When the input signal is a noisy signal then SHIFT can be 1100 and higher.

When the input signal has the stable statistic properties then the code SHIFT can be set as a constant. Then the OVF outputs can be not in use, and the CNORM units will be removed from the project by the hardware optimization when the core is synthesized.

3.2.5. ROTATOR128

The unit ROTATOR implements the complex vector rotating to the angles W_{128}^{ms}. The complex twiddle factors are stored in the unit WROM128. Here the ROM contains the table of coefficients

(w0, w0, w0, w0, w0, w0, w0, w0, w0, w0, w0 , w0 , w0 , w0 ,w0,

w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13,w14,w15,

w0,w3, w6,w9, w12, w15, w18, w21, w24, w27, w30, w33, w36, w39, w42,w45,

……

w0,w7,w15,w23,w31,w39,w47,w55,w63,w71,w79,w97,w103,w111,w119,w127),

where $wi = W_{128}^{i}$. Here the row and column indexes are m and s respectively. These coefficients are read in the natural order address by the 7-bit counter addrw. The complex vector rotating is implemented by the usual schema of the complex number multiplier which contains 4 multiply units and 2 adders.

3.2.6. CT128

CT128 (Counter process module 128) is used to generate the address sequence to the UG unit starting after the START impulse, where the UG units outputs the testing complex signal to the UUT unit (FFT128) with the period of 128 clock cycles.

3.3. Verilog code

Followed the blocks being demonstrated, we have build a system using Verilog language. The code for each blocks is shown as followed:

- **Input module (Wave_ROM128)**

The data input is all indicated in Wave_ROM128.v.

```
module Wave_ROM128 ( ADDR ,DATA_RE,DATA_IM,DATA_REF );
        output [15:0] DATA_RE,DATA_IM,DATA_REF ;
        input [6:0]    ADDR ;
        reg [15:0] cosi[0:127];
        initial   begin
```

The input including DATA_RE, DATA_IM and DATA_REF :

```
        initial   begin
cosi[0]=16'h7FFF; cosi[1]=16'h7FD7; cosi[2]=16'h7F61; cosi[3]=16'h7E9C; cosi[4]=16'h7D89; cosi[5]=16'h7C29; cosi[6]=16'h7A7C; cosi[7]=16'h7883; cosi[8]=16'h7640; cosi[9]=16'h
cosi[16]=16'h5A81; cosi[17]=16'h55F4; cosi[18]=16'h5133; cosi[19]=16'h4C3F; cosi[20]=16'h471C; cosi[21]=16'h41CD; cosi[22]=16'h3C56; cosi[23]=16'h36B9; cosi[24]=16'h30FB; cos
cosi[32]=16'h0000; cosi[33]=16'hF9B9; cosi[34]=16'hF375; cosi[35]=16'hED39; cosi[36]=16'hE708; cosi[37]=16'hE0E7; cosi[38]=16'hDAD9; cosi[39]=16'hD4E2; cosi[40]=16'hCF05; cos
cosi[48]=16'hA57F; cosi[49]=16'hA12A; cosi[50]=16'h9D0F; cosi[51]=16'h9932; cosi[52]=16'h9594; cosi[53]=16'h9237; cosi[54]=16'h8F1F; cosi[55]=16'h8C4B; cosi[56]=16'h89C0; cos
cosi[64]=16'h8001; cosi[65]=16'h8029; cosi[66]=16'h809F; cosi[67]=16'h8164; cosi[68]=16'h8277; cosi[69]=16'h83D7; cosi[70]=16'h8584; cosi[71]=16'h877D; cosi[72]=16'h89C0; cos
cosi[80]=16'hA57F; cosi[81]=16'hAA0C; cosi[82]=16'hAECD; cosi[83]=16'hB3C1; cosi[84]=16'hB8E4; cosi[85]=16'hBE33; cosi[86]=16'hC3AA; cosi[87]=16'hC947; cosi[88]=16'hCF05; cos
cosi[96]=16'h0000; cosi[97]=16'h0647; cosi[98]=16'h0C8B; cosi[99]=16'h12C7; cosi[100]=16'h18F8; cosi[101]=16'h1F19; cosi[102]=16'h2527; cosi[103]=16'h2B1E; cosi[104]=16'h30FB
cosi[112]=16'h5A81; cosi[113]=16'h5ED6; cosi[114]=16'h62F1; cosi[115]=16'h66CE; cosi[116]=16'h6A6C; cosi[117]=16'h6DC9; cosi[118]=16'h70E1; cosi[119]=16'h73B5; cosi[120]=16'h
    end
```

```
        reg [15:0] sine[0:127];
        initial   begin
sine[0]=16'h0000; sine[1]=16'h0647; sine[2]=16'h0C8B; sine[3]=16'h12C7; sine[4]=16'h18F8; sine[5]=16'h1F19; sine[6]=16'h2527; sine[7]=16'h2B1E; sine[8]=16'h30FB; sine[9]=16'h
sine[16]=16'h5A81; sine[17]=16'h5ED6; sine[18]=16'h62F1; sine[19]=16'h66CE; sine[20]=16'h6A6C; sine[21]=16'h6DC9; sine[22]=16'h70E1; sine[23]=16'h73B5; sine[24]=16'h7640; sin
sine[32]=16'h7FFF; sine[33]=16'h7FD7; sine[34]=16'h7F61; sine[35]=16'h7E9C; sine[36]=16'h7D89; sine[37]=16'h7C29; sine[38]=16'h7A7C; sine[39]=16'h7883; sine[40]=16'h7640; sin
sine[48]=16'h5A81; sine[49]=16'h55F4; sine[50]=16'h5133; sine[51]=16'h4C3F; sine[52]=16'h471C; sine[53]=16'h41CD; sine[54]=16'h3C56; sine[55]=16'h36B9; sine[56]=16'h30FB; sin
sine[64]=16'h0000; sine[65]=16'hF9B9; sine[66]=16'hF375; sine[67]=16'hED39; sine[68]=16'hE708; sine[69]=16'hE0E7; sine[70]=16'hDAD9; sine[71]=16'hD4E2; sine[72]=16'hCF05; sin
sine[80]=16'hA57F; sine[81]=16'hA12A; sine[82]=16'h9D0F; sine[83]=16'h9932; sine[84]=16'h9594; sine[85]=16'h9237; sine[86]=16'h8F1F; sine[87]=16'h8C4B; sine[88]=16'h89C0; sin
sine[96]=16'h8001; sine[97]=16'h8029; sine[98]=16'h809F; sine[99]=16'h8164; sine[100]=16'h8277; sine[101]=16'h83D7; sine[102]=16'h8584; sine[103]=16'h877D; sine[104]=16'h89C0
sine[112]=16'hA57F; sine[113]=16'hAA0C; sine[114]=16'hAECD; sine[115]=16'hB3C1; sine[116]=16'hB8E4; sine[117]=16'hBE33; sine[118]=16'hC3AA; sine[119]=16'hC947; sine[120]=16'h
    end
```

```
        reg [15:0] deltas[0:127];
        initial   begin
 deltas[0]=16'h0000; deltas[1]=16'h0000; deltas[2]=16'h0000; deltas[3]=16'h0000;
 deltas[4]=16'h0000; deltas[5]=16'h0000; deltas[6]=16'h0000; deltas[7]=16'h0000;
 deltas[8]=16'h0000; deltas[9]=16'h0000; deltas[10]=16'h0000; deltas[11]=16'h0000;
 deltas[12]=16'h0000; deltas[13]=16'h0000; deltas[14]=16'h0000; deltas[15]=16'h0000;
 deltas[16]=16'h0000; deltas[17]=16'h0000; deltas[18]=16'h0000; deltas[19]=16'h0000;
 deltas[20]=16'h0000; deltas[21]=16'h0000; deltas[22]=16'h0000; deltas[23]=16'h0000;
 deltas[24]=16'h0000; deltas[25]=16'h0000; deltas[26]=16'h0000; deltas[27]=16'h0000;
 deltas[28]=16'h0000; deltas[29]=16'h0000; deltas[30]=16'h0000; deltas[31]=16'h0000;
 deltas[32]=16'h0000; deltas[33]=16'h0000; deltas[34]=16'h0000; deltas[35]=16'h0000;
 deltas[36]=16'h0000; deltas[37]=16'h0000; deltas[38]=16'h0000; deltas[39]=16'h0000;
 deltas[40]=16'h0000; deltas[41]=16'h0000; deltas[42]=16'h0000; deltas[43]=16'h0000;
 deltas[44]=16'h0000; deltas[45]=16'h0000; deltas[46]=16'h0000; deltas[47]=16'h0000;
 deltas[48]=16'h0000; deltas[49]=16'h0000; deltas[50]=16'h0000; deltas[51]=16'h0000;
 deltas[52]=16'h0000; deltas[53]=16'h0000; deltas[54]=16'h0000; deltas[55]=16'h0000;
 deltas[56]=16'h0000; deltas[57]=16'h0000; deltas[58]=16'h0000; deltas[59]=16'h0000;
 deltas[60]=16'h0000; deltas[61]=16'h0000; deltas[62]=16'h0000; deltas[63]=16'h0000;
 deltas[64]=16'h0000; deltas[65]=16'h0000; deltas[66]=16'h0000; deltas[67]=16'h0000;
 deltas[68]=16'h0000; deltas[69]=16'h0000; deltas[70]=16'h0000; deltas[71]=16'h0000;
 deltas[72]=16'h0000; deltas[73]=16'h0000; deltas[74]=16'h0000; deltas[75]=16'h0000;
 deltas[76]=16'h0000; deltas[77]=16'h0000; deltas[78]=16'h0000; deltas[79]=16'h0000;
 deltas[80]=16'h0000; deltas[81]=16'h0000; deltas[82]=16'h0000; deltas[83]=16'h0000;
 deltas[84]=16'h0000; deltas[85]=16'h0000; deltas[86]=16'h0000; deltas[87]=16'h0000;
 deltas[88]=16'h0000; deltas[89]=16'h0000; deltas[90]=16'h0000; deltas[91]=16'h0000;
 deltas[92]=16'h0000; deltas[93]=16'h0000; deltas[94]=16'h0000; deltas[95]=16'h0000;
 deltas[96]=16'h0000; deltas[97]=16'h0000; deltas[98]=16'h0000; deltas[99]=16'h0000;
 deltas[100]=16'h0000; deltas[101]=16'h0000; deltas[102]=16'h0000; deltas[103]=16'h0000;
 deltas[104]=16'h0000; deltas[105]=16'h0000; deltas[106]=16'h0000; deltas[107]=16'h0000;
 deltas[108]=16'h0000; deltas[109]=16'h0000; deltas[110]=16'h0000; deltas[111]=16'h0000;
 deltas[112]=16'h0000; deltas[113]=16'h0000; deltas[114]=16'h0000; deltas[115]=16'h0000;
```

- **FFT data buffer**

Including 2 data buffer is used to relase 8th and 16th inverse output address.

BUFRAM128C_1

```
module BUFRAM128C_1 ( CLK ,RST ,ED ,START ,DR ,DI ,RDY ,DOR ,DOI );
        `FFT128paramnb
        output RDY ;
        reg RDY ;
        output [nb-1:0] DOR ;
        wire [nb-1:0] DOR ;
        output [nb-1:0] DOI ;
        wire [nb-1:0] DOI ;

        input CLK ;
        wire CLK ;
        input RST ;
        wire RST ;
        input ED ;
        wire ED ;
        input START ;
        wire START ;
        input [nb-1:0] DR ;
        wire [nb-1:0] DR ;
        input [nb-1:0] DI ;
        wire [nb-1:0] DI ;

        wire odd, we;
        wire [6:0] addrw,addrr;
        reg [7:0] addr;
        reg [8:0] ct2;          //counter for the RDY signal


assign  addrw=  addr[6:0];
assign  odd=addr[7];                        // signal which switches the 2 parts of the buffer
assign  addrr={addr[2 : 0], addr[6 : 3]};        // 16-th inverse output address
assign  we = ED;

        RAM2x128C #(nb) URAM(.CLK(CLK),.ED(ED),.WE(we),.ODD(odd),
        .ADDRW(addrw),  .ADDRR(addrr),
        .DR(DR),.DI(DI),
        .DOR(DOR),      .DOI(DOI));

endmodule
```

BUFRAM128C_2

```
module BUFRAM128C_2 ( CLK ,RST ,ED ,START ,DR ,DI ,RDY ,DOR ,DOI );
        `FFT128paramnb
        output RDY ;
        reg RDY ;
        output [nb-1:0] DOR ;
        wire [nb-1:0] DOR ;
        output [nb-1:0] DOI ;
        wire [nb-1:0] DOI ;

        input CLK ;
        wire CLK ;
        input RST ;
        wire RST ;
        input ED ;
        wire ED ;
        input START ;
        wire START ;
        input [nb-1:0] DR ;
        wire [nb-1:0] DR ;
        input [nb-1:0] DI ;
        wire [nb-1:0] DI ;

        wire odd, we;
        wire [6:0] addrw,addrr;
        reg [7:0] addr;
        reg [8:0] ct2;          //counter for the RDY signal


assign  addrw=  addr[6:0];
assign  odd=addr[7];                        // signal which switches the 2 parts of the buffer
assign  addrr={addr[3 : 0], addr[6 : 4]};        // 8-th inverse output address
assign  we = ED;

       RAM2x128C #(nb) URAM(.CLK(CLK),.ED(ED),.WE(we),.ODD(odd),
       .ADDRW(addrw),  .ADDRR(addrr),
       .DR(DR),.DI(DI),
       .DOR(DOR),      .DOI(DOI));

endmodule
```

For the full operation the 2 BUFRAM requires the running of the RAM128 RAM2x128

RAM2x128

```
module RAM2x128C ( CLK ,ED ,WE ,ODD ,ADDRW ,ADDRR ,DR ,DI ,DOR ,DOI );
        `FFT128paramnb


        output [nb-1:0] DOR ;
        wire [nb-1:0] DOR ;
        output [nb-1:0] DOI ;
        wire [nb-1:0] DOI ;
        input CLK ;
        wire CLK ;
        input ED ;
        wire ED ;
        input WE ;          //write enable
        wire WE ;
        input ODD ;       // RAM part switshing
        wire ODD ;
        input [6:0] ADDRW ;
        wire [6:0] ADDRW ;
        input [6:0] ADDRR ;
        wire [6:0] ADDRR ;
        input [nb-1:0] DR ;
        wire [nb-1:0] DR ;
        input [nb-1:0] DI ;
        wire [nb-1:0] DI ;
```

Changing the ADDR type based on the addr and we, which defines the state of reading or writing of the buffer.

```
  reg     oddd,odd2;
  always @( posedge CLK) begin //switch which reswiches the RAM parts
                if (ED) begin
                                oddd<=ODD;
                                odd2<=oddd;
                        end
        end
  `ifdef  FFT128bufferports1
  //One-port RAMs are used
  wire we0,we1;
  wire    [nb-1:0] dor0,dor1,doi0,doi1;
  wire    [6:0] addr0,addr1;



  assign  addr0 =ODD?  ADDRW: ADDRR;              //MUXA0
  assign  addr1 = ~ODD? ADDRW:ADDRR;      // MUXA1
  assign  we0   =ODD?  WE: 0;                  // MUXW0:
  assign  we1   =~ODD? WE: 0;                     // MUXW1:


//1-st half - write when odd=1   read when odd=0
RAM128 #(nb) URAM0(.CLK(CLK),.ED(ED),.WE(we0), .ADDR(addr0),.DI(DR),.DO(dor0)); //
RAM128 #(nb) URAM1(.CLK(CLK),.ED(ED),.WE(we0), .ADDR(addr0),.DI(DI),.DO(doi0));

//2-d half
RAM128 #(nb) URAM2(.CLK(CLK),.ED(ED),.WE(we1), .ADDR(addr1),.DI(DR),.DO(dor1));//
RAM128 #(nb) URAM3(.CLK(CLK),.ED(ED),.WE(we1), .ADDR(addr1),.DI(DI),.DO(doi1));


        `else
        //Two-port RAM is used
        wire [7:0] addrr2 = {ODD,ADDRR};
        wire [7:0] addrw2 = {~ODD,ADDRW};
        wire [2*nb-1:0] di= {DR,DI} ;

        //wire [2*nb-1:0] doi;
        reg [2*nb-1:0] doi;

        reg [2*nb-1:0] ram [255:0];
        reg [7:0] read_addra;
        always @(posedge CLK) begin
                        if (ED)
                                begin
                                        if (WE)
                                                ram[addrw2] <= di;
                                        read_addra <= addrr2;
                                   doi = ram[read_addra];
                                end
                end
        //assign

        assign  DOR=doi[2*nb-1:nb];             // Real read data
        assign  DOI=doi[nb-1:0];                // Imaginary read data


        `endif
endmodule
```

RAM128

```
module RAM128 ( CLK, ED,WE ,ADDR ,DI ,DO );
        `FFT128paramnb

        output [nb-1:0] DO ;
        reg [nb-1:0] DO ;
        input CLK ;
        wire CLK ;
        input ED;
        input WE ;
        wire WE ;
        input [6:0] ADDR ;
        wire [6:0] ADDR ;
        input [nb-1:0] DI ;
        wire [nb-1:0] DI ;
        reg [nb-1:0] mem [127:0];
        reg [6:0] addrrd;

        always @(posedge CLK) begin
                        if (ED) begin
                                        if (WE)         mem[ADDR] <= DI;
                                        addrrd <= ADDR;          //storing the address
                                        DO <= mem[addrrd];         // registering the read datum
                                end
                end

endmodule
```

- **CNORM**

```
40  // DESCRIPTION  :       Normalization unit
41  // FUNCTION:            shifting left up to 3 bits
42  //      FILES:                  CNORM.v
43  //  PROPERTIES: 1) shifting left up to 3 bits controlled by the 2-bit code SHIFT
44  //                                                  2) Is registered
45  //                                                  3) Overflow detector detects the overflow event
46  //                          by the given shift condition. The detector is zeroed by the START signal
47  //                         4) RDY is the START signal delayed to a single clock cycle
48  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
49  // File        :
50  // Generated   :
51  //
52  // Description :
53  //
54  //--------------------------------------------------------------------------------
55  `timescale 1 ns / 1 ps
56  `include "FFT128_CONFIG.inc"
57
58  module CNORM ( CLK ,ED ,START ,DR ,DI ,SHIFT ,OVF ,RDY ,DOR ,DOI );
59          `FFT128paramnb
60
61          output OVF ;
62          reg OVF ;
63          output RDY ;
64          reg RDY ;
65          output [nb+1:0] DOR ;
66          wire [nb+1:0] DOR ;
67          output [nb+1:0] DOI ;
68          wire [nb+1:0] DOI ;
69
```

```
70          input CLK ;
71          wire CLK ;
72          input ED ;
73          wire ED ;
74          input START ;
75          wire START ;
76          input [nb+2:0] DR ;
77          wire [nb+2:0] DR ;
78          input [nb+2:0] DI ;
79          wire [nb+2:0] DI ;
80          input [1:0] SHIFT ;                         //shift left code to 0,1,2,3 bits
81          wire [1:0] SHIFT ;
82
83          wire signed [nb+3:0]     diri,diii;
84          assign diri = DR << SHIFT;
85          assign diii = DI << SHIFT;
86
87          reg [nb+2:0]    dir,dii;
88
89          `ifdef FFT128round                          //rounding
90          always @( posedge CLK )    begin
91                          if (ED)   begin
92                                          if (diri[nb+2] && ~diri[0])     // <0 with LSB=00
93                                                  dir<=diri;
94                                          else   dir<=diri+2;
95                                          if (diii[nb+2] && ~diii[0])
96                                                  dii<=diii;
97                                          else   dii<=diii+2;
98                                  end
99                  end
100
101         `else                                                               //truncation
102         always @( posedge CLK )    begin
```

```
103                         if (ED)   begin
104                                         dir<=diri;
105                                         dii<=diii;
106                                 end
107             end
108
109
110         `endif
111
112         always @( posedge CLK )         begin
113                         if (ED)   begin
114                                         RDY<=START;
115                                         if (START)
116                                                 OVF<=0;
117                                         else
118                                                 case (SHIFT)
119                                                   2'b01 : OVF<= (DR[nb+3] != DR[nb+2]) || (DI[nb+3] != DI[nb+2]);
120                                                   2'b10 : OVF<= (DR[nb+3] != DR[nb+2]) || (DI[nb+3] != DI[nb+2]) ||
121                                                           (DR[nb+3] != DR[nb+1]) || (DI[nb+3] != DI[nb+1]);
122                                                   2'b11 : OVF<= (DR[nb+3] != DR[nb+2]) || (DI[nb+3] != DI[nb+2])||
123                                                           (DR[nb+3] != DR[nb]) || (DI[nb+3] != DI[nb]) ||
124                                                           (DR[nb+3] != DR[nb+1]) || (DI[nb+3] != DI[nb+1]);
125                                                 endcase
126                                 end
127             end
128
129         assign DOR= dir[nb+3:2];
130         assign DOI= dii[nb+3:2];
131
132 endmodule
```

- **ROTATOR 128**

```
54  module ROTATOR128 (CLK ,RST,ED,START, DR,DI, DOR, DOI,RDY  );
55          `FFT128paramnb
56          `FFT128paramnw
57
58          input RST ;
59          wire RST ;
60          input CLK ;
61          wire CLK ;
62          input ED ; //operation enable
63          input [nb-1:0] DI;  //Imaginary part of data
64          wire [nb-1:0]  DI ;
65          input [nb-1:0]  DR ; //Real part of data
66          input START ;              //1-st Data is entered after this impulse
67          wire START ;
68
69          output [nb-1:0]  DOI ; //Imaginary part of data
70          wire [nb-1:0]  DOI ;
71          output [nb-1:0]  DOR ; //Real part of data
72          wire [nb-1:0]  DOR ;
73          output RDY ;       //repeats START impulse following the output data
74          reg RDY ;
```

```
77           reg [6:0] addrw;
78           reg sd1,sd2;
79           always  @( posedge CLK)    //address counter for twiddle factors
80                   begin
81                           if (RST) begin
82                                                   addrw<=0;
83                                                   sd1<=0;
84                                                   sd2<=0;
85                                   end
86                           else if (START && ED)  begin
87                                                   addrw[6:0]<=0;
88                                                   sd1<=START;
89                                                   sd2<=0;
90                                   end
91                           else if (ED)       begin
92                                                   addrw<=addrw+1;
93                                                   sd1<=START;
94                                                   sd2<=sd1;
95                                                   RDY<=sd2;
96                                   end
97                   end
98
99                   wire     [6:0] addrwi;
```

```
100         assign    addrwi={addrw[4 : 0], addrw[6 : 3] };
101                 wire signed [nw-1:0] wr,wi; //twiddle factor coefficients
102         //twiddle factor ROM
103         WROM128 UROM( .ADDR(addrwi),     .WR(wr),.WI(wi) );
104
105
106         reg signed [nb-1 : 0] drd,did;
107         reg signed [nw-1 : 0] wrd,wid;
108         wire signed [nw+nb-1 : 0] drri,drii,diri,diii;
109         reg signed [nb:0] drr,dri,dir,dii,dwr,dwi;
110
111         assign          drri=drd*wrd;
112         assign  diri=did*wrd;
113         assign  drii=drd*wid;
114         assign  diii=did*wid;
```

```
116         always @(posedge CLK)    //complex multiplier
117                 begin
118                         if (ED) begin
119                                         drd<=DR;
120                                         did<=DI;
121                                         wrd<=wr;
122                                         wid<=wi;
123                                         drr<=drri[nw+nb-1 :nw-1]; //msbs of multiplications are stored
124                                         dri<=drii[nw+nb-1 : nw-1];
125                                         dir<=diri[nw+nb-1 : nw-1];
126                                         dii<=diii[nw+nb-1 : nw-1];
127                                         dwr<=drr - dii;
128                                         dwi<=dri + dir;
129                                 end
130                 end
131         assign DOR=dwr[nb:1];
132         assign DOI=dwi[nb:1];
133
134 endmodule
```

WROM128

Bấm vào đây để nhập văn bản.

```
96                  `endif
97
98
99          wire [15:0] wri,wii       ;
100         assign wri=cosw[ADDR];
101         assign wii=sinw[ADDR];
102
103         wire [nw:0] wrt,wit;
104
105
106         assign  wrt = wri[15:16-nw];
107         assign  wit = wii[15:16-nw];
108         assign WR= wrt[nw-1:0];
109         assign WI= wit[nw-1:0];
110
111 endmodule
```

3.4. Implement Algorithm on C++ / Python

In order to verify our design, we provide a simulation on different IDE with different indication. We promote a solution using python with the input is the same from the input in the Verilog system.

1st step is to convert the input data into different .txt file as real number and imaginary numbers:

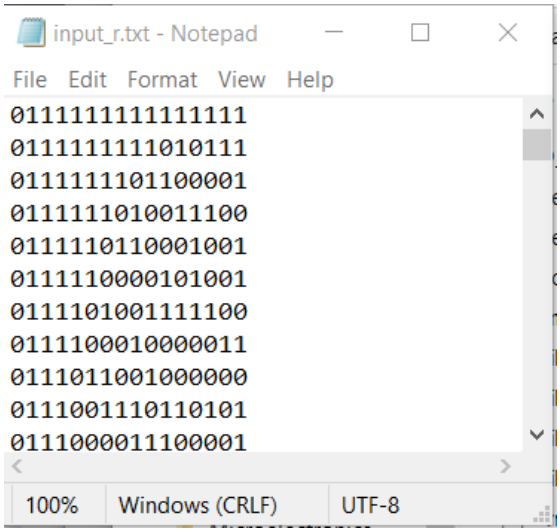

Figure 5 Real data input

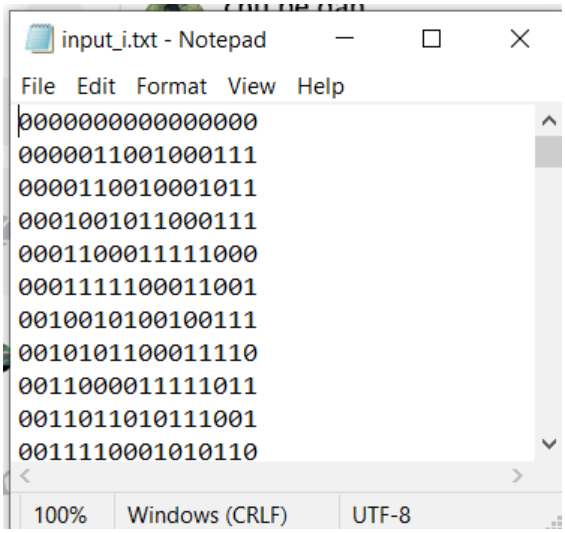

Figure 6 Imaginary data input

We verify the result by using Python with the library numpy , which will help us to calculate the FFT 128.

The code is indicated as below:

In [9]:
```
import numpy as np
import struct
```

In [10]:
```
# create empty lists for decimalr_num and decimali_num
decimalr_nums = []
decimali_nums = []
```

In [11]:
```
# open the .txt file for reading
with open('F:\VHDL\ET5080E - 20221\Big project\pipelined_fft_128-master_ref\work\input_r.txt', 'r') as f:
    # read the file line by line
    for line in f:
        # strip the line of any whitespace
        line = line.strip()
        # convert the binary string to an integer
        binary_num = int(line, 2)
        # use the struct module to convert the binary number to a signed decimal
        decimalr_num = struct.unpack('h', struct.pack('H', binary_num))[0]
        # append the decimal number to the decimalr_nums list
        decimalr_nums.append(decimalr_num)
```

In [12]:
```
# open the .txt file for reading
with open('F:\VHDL\ET5080E - 20221\Big project\pipelined_fft_128-master_ref\work\input_i.txt', 'r') as f:
    # read the file line by line
    for line in f:
        # strip the line of any whitespace
        line = line.strip()
        # convert the binary string to an integer
        binary_num = int(line, 2)
        # use the struct module to convert the binary number to a signed decimal
        decimali_num = struct.unpack('h', struct.pack('H', binary_num))[0]
        # append the decimal number to the decimali_nums list
        decimali_nums.append(decimali_num)
```

In [13]:

```
# use the zip function to iterate over the two lists and create a new list of complex numbers
complex_numbers = [complex(decimalr_num, decimali_num) for decimalr_num, decimali_num in zip(decimalr_nums, decimali_nums)]
print(complex_numbers)
```

```
[(32767+0j), (32727+1607j), (32609+3211j), (32412+4807j), (32137+6392j), (31785+7961j), (31356+9511j), (30851+11038j), (30272+12539j), (29621+14009j),
(28897+15446j), (28105+16845j), (27244+18204j), (26318+19519j), (25329+20787j), (24278+22004j), (23169+23169j), (22004+24278j), (20787+25329j), (19519+
26318j), (18204+27244j), (16845+28105j), (15446+28897j), (14009+29621j), (12539+30272j), (11038+30851j), (9511+31356j), (7961+31785j), (6392+32137j),
(4807+32412j), (3211+32609j), (1607+32727j), 32767j, (-1607+32727j), (-3211+32609j), (-4807+32412j), (-6392+32137j), (-7961+31785j), (-9511+31356j), (-
11038+30851j), (-12539+30272j), (-14009+29621j), (-15446+28897j), (-16845+28105j), (-18204+27244j), (-19519+26318j), (-20787+25329j), (-22004+24278j),
(-23169+23169j), (-24278+22004j), (-25329+20787j), (-26318+19519j), (-27244+18204j), (-28105+16845j), (-28897+15446j), (-29621+14009j), (-30272+12539
j), (-30851+11038j), (-31356+9511j), (-31785+7961j), (-32137+6392j), (-32412+4807j), (-32609+3211j), (-32727+1607j), (-32767+0j), (-32727-1607j), (-326
09-3211j), (-32412-4807j), (-32137-6392j), (-31785-7961j), (-31356-9511j), (-30851-11038j), (-30272-12539j), (-29621-14009j), (-28897-15446j), (-28105-
16845j), (-27244-18204j), (-26318-19519j), (-25329-20787j), (-24278-22004j), (-23169-23169j), (-22004-24278j), (-20787-25329j), (-19519-26318j), (-1820
4-27244j), (-16845-28105j), (-15446-28897j), (-14009-29621j), (-12539-30272j), (-11038-30851j), (-9511-31356j), (-7961-31785j), (-6392-32137j), (-4807-
32412j), (-3211-32609j), (-1607-32727j), -32767j, (1607-32727j), (3211-32609j), (4807-32412j), (6392-32137j), (7961-31785j), (9511-31356j), (11038-3085
1j), (12539-30272j), (14009-29621j), (15446-28897j), (16845-28105j), (18204-27244j), (19519-26318j), (20787-25329j), (22004-24278j), (23169-23169j), (2
4278-22004j), (25329-20787j), (26318-19519j), (27244-18204j), (28105-16845j), (28897-15446j), (29621-14009j), (30272-12539j), (30851-11038j), (31356-95
11j), (31785-7961j), (32137-6392j), (32412-4807j), (32609-3211j), (32727-1607j)]
```

In [14]:

```
# Perform FFT on complex numbers
fft_result = np.fft.fft(complex_numbers)
fft_result = fft_result/16
print(fft_result)
```

```
[ 0.00000000e+00+0.j  2.62131350e+05+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -9.06769123e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -1.59382638e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -3.99004091e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -7.74704912e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -1.02138082e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.42555480e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  2.29793597e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -3.64241723e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.77396306e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  9.98393046e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  4.46651127e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.21767613e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -2.90565723e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.03172203e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -7.77603784e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  1.47040921e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.25797636e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  6.39782408e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -5.04409257e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  8.95531498e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  7.13565401e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -4.98754636e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.59849845e-02+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -5.75710515e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  1.25829451e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  3.59577542e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  7.21501117e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -3.25673467e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  5.81532858e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  1.56575185e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  2.83547289e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j]
```

The final result should look as below, with the same output as it is indicated from the wave in ModelSim:

```
[ 0.00000000e+00+0.j  2.62131350e+05+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -9.06769123e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -1.59382638e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -3.99004091e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -7.74704912e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -1.02138082e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.42555480e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  2.29793597e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
 -3.64241723e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.77396306e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  9.98393046e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  4.46651127e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -2.21767613e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -2.90565723e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.03172203e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -7.77603784e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  1.47040921e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.25797636e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  6.39782408e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -5.04409257e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  8.95531498e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  7.13565401e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -4.98754636e-01+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  9.59849845e-02+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j -5.75710515e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  1.25829451e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  3.59577542e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  7.21501117e-01+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j -3.25673467e-02+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  5.81532858e-01+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  1.56575185e+00+0.j  0.00000000e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j  2.83547289e+00+0.j
  0.00000000e+00+0.j  0.00000000e+00+0.j]
```

Figure 7 Simulation results from Jupyter Notebook

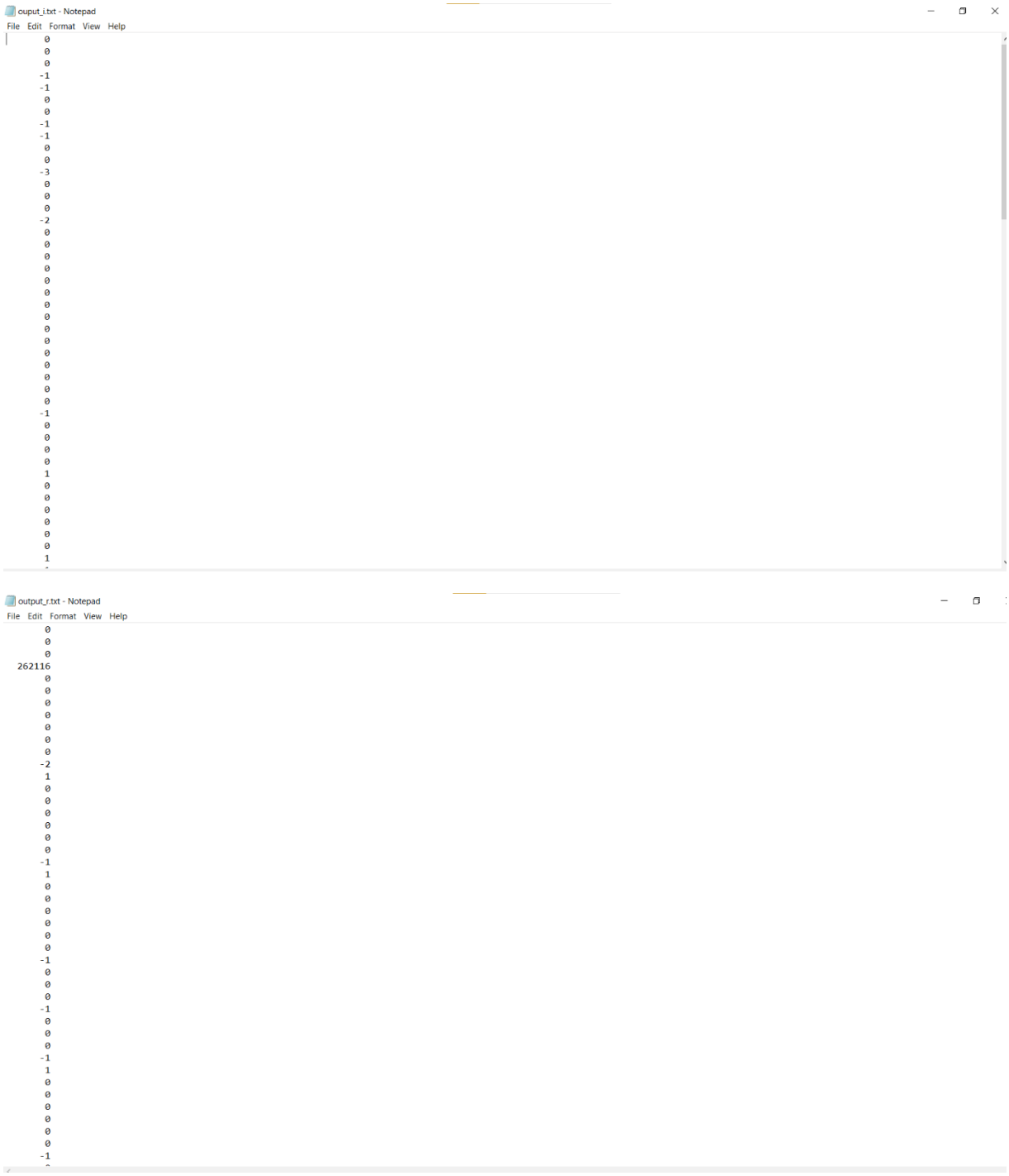

Figure 8 Output in Verilog

```
Entrée [51]: for i in range (0,128):
                 print(fft_result[i].real)

0.0
262131.34953936416
0.0
0.0
0.0
-0.9067691230273303
0.0
0.0
0.0
-1.5938263763554479
0.0
0.0
0.0
-0.39900409125605235
0.0
0.0
0.0
-0.774704911567712
0.0
```

Figure 9 Real output in Python

```
Entrée [50]: for i in range (0,128):
                 print (fft_result[i].imag)

0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
```

Figure 10 Imaginary output in Python

3.5. RTL design

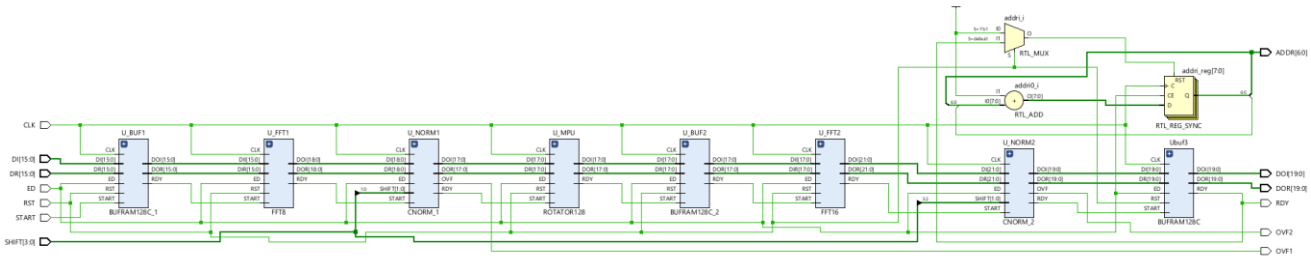

Figure 11 RTL schematic

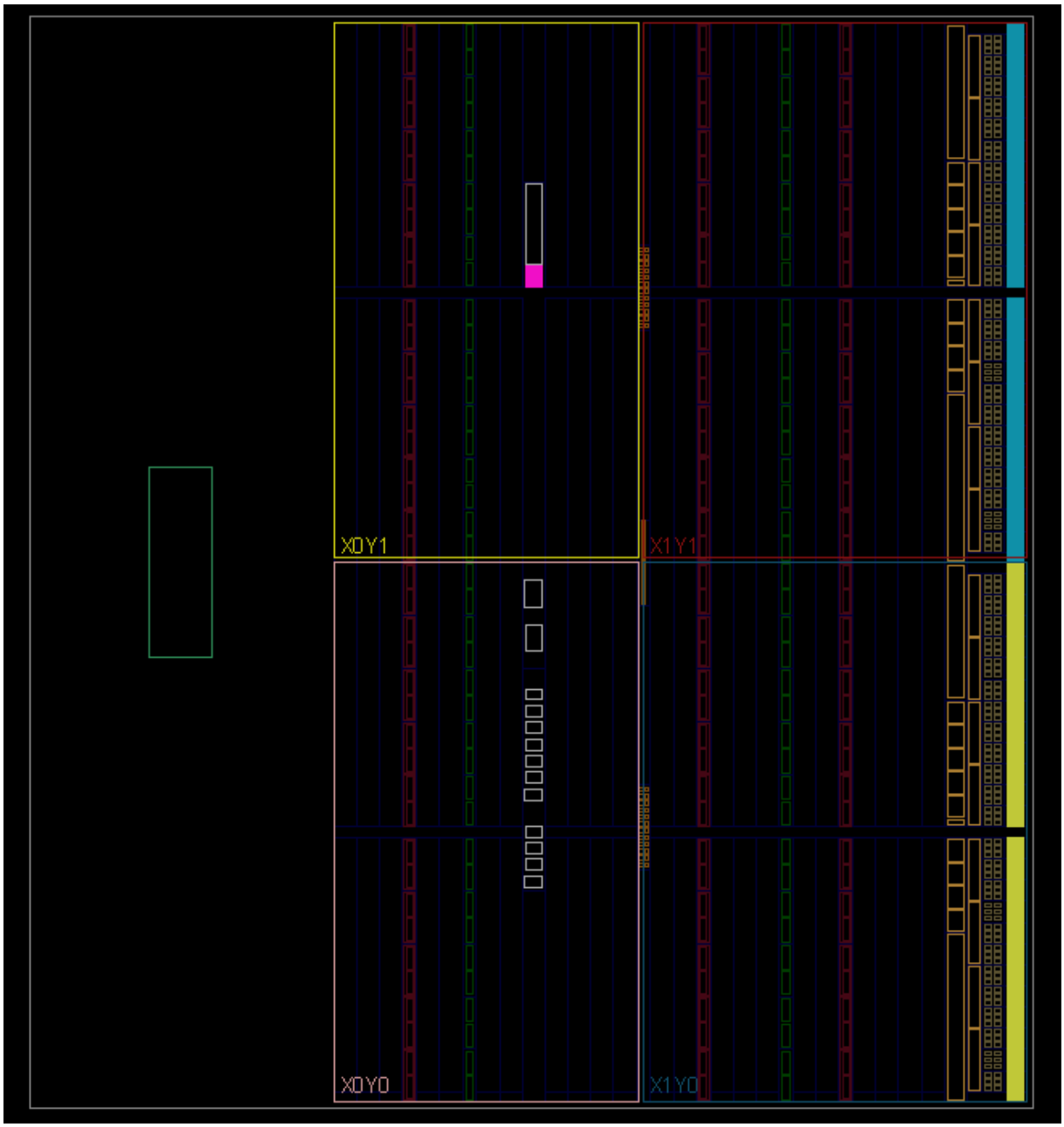

Figure 12 Synthesis overview on chip xc7z010iclg225-1L

3.6. Testbench

3.6.1. Block Diagram

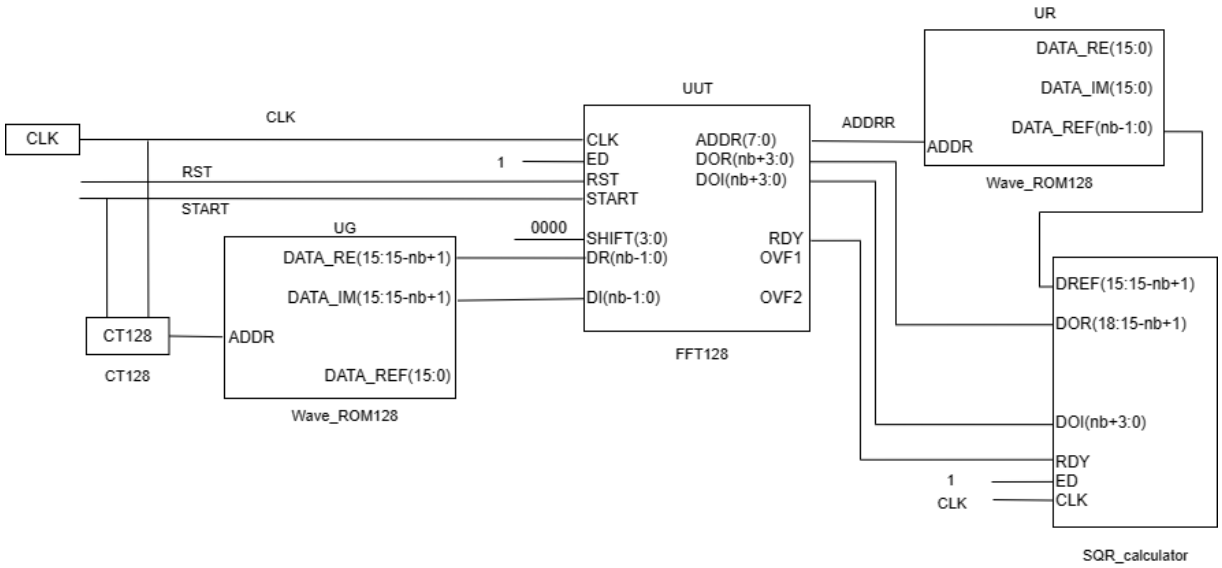

Figure 13 Testbench structure

The units UG and UR are implemented as ROMs which contain the generating waveforms (UG) and the reference waveform (UR). They are instantiated as a component Wave_ROM128 which is described in the file Wave_ROM128.v. This file can be generated by the PERL script sinerom128_gen.pl. In this script the tables of sums of up to 4 sine and cosine waves are generated which frequencies are set by the parameters $f1, $f2, $f3, and $f4. The table of the respective frequency bins is generated too. The table length is set as $n = 128. The samples of these tables are outputted to the outputs DATA_IM, DATA_RE, and DARA_REF of the component Wave_ROM128, respectively.

The counter process CT128 generates the address sequence to the UG unit starting after the START impulse. The UG unit outputs the testing complex signal to the UUT unit (FFT128) with the period of 128 clock cycles.

When the FFT result is ready then UUT generates the RDY signal after that it generates the address sequence ADDR of the results. This sequence is the input data for the UR unit which outputs the correct real samples (bins) of the spectrum. Note that because the input data is the complex sine wave sum then the imaginary part of the

spectrum must be a sequence of zeros. The process SQR_calculator calculates the sum of square differences between spectrum results and reference samples. It starts after the impulse RDY and finishes after 128 clock cycles. Then the result is divided to 128 and outputted in the message in the console of the simulator. For example, the message: “rms error is 1 lsb” means that the square of the residue mean square error is equal to 1 LSB of the spectrum result.

When the model FFT128 is correct and its bit widths are selected correctly then the rms error not succeeded 1 or 2. When this model is not correct then the message will be a huge positive or negative integer, or ‘X’.

The model correctness can be proven or investigated by looking at the input and output waveforms. Fig.12 illustrates the waveforms of the input signals, and Fig.13 shows the output waveforms. Not that the scale of the waveform DOI is in thousand times higher than one of the waveform DOR.

CHAPTER 4. SIMULATION RESULTS

4.1. WAVEFORM

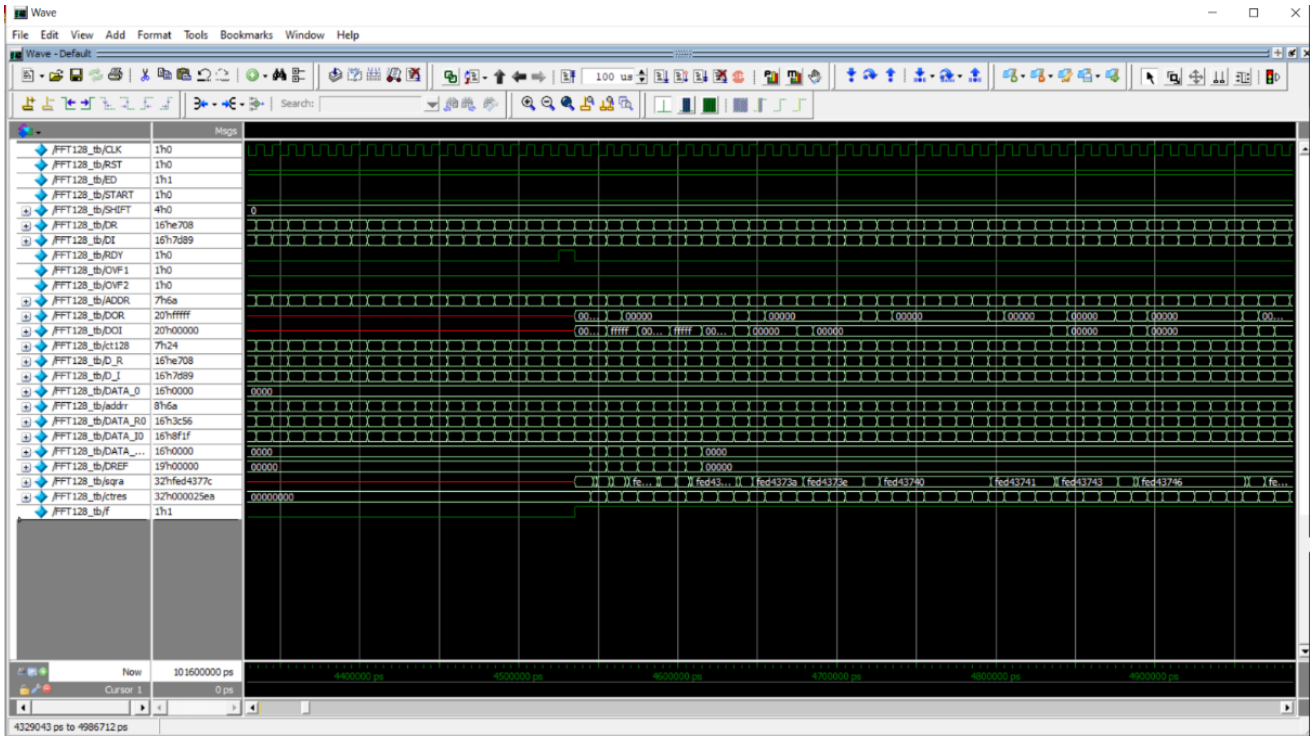

Figure 14 Waveform

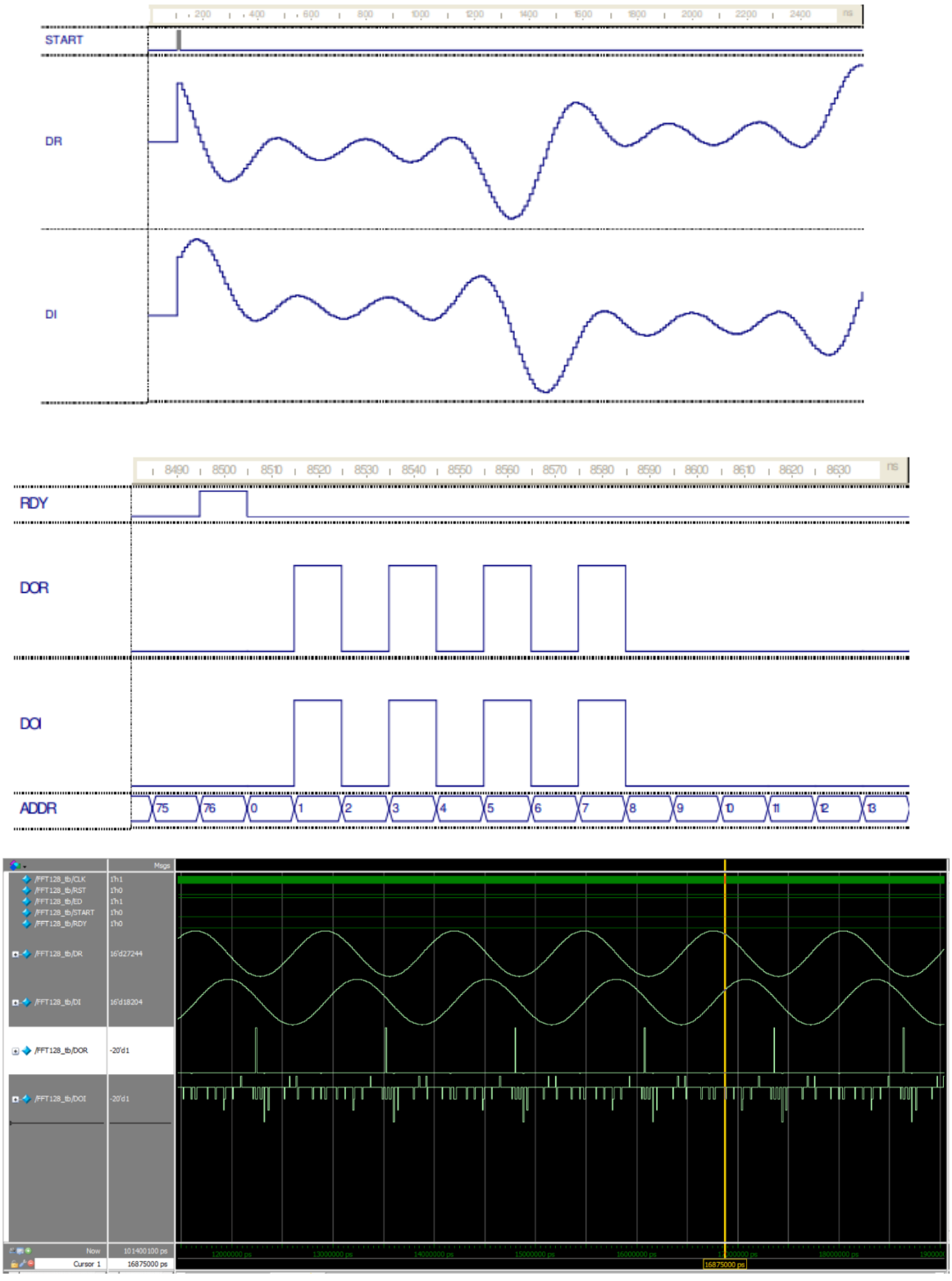

Figure 15 Decimal waveform

4.2. Synthesis

We synthesized the design on the FPGA Cyclone IV GX – EP4CGX22CF19C6 on Quartus II 64-Bit

Flow Summary	
Flow Status	Successful - Mon Jan 30 16:14:05 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	FFT128
Top-level Entity Name	FFT128
Family	Cyclone IV GX
Total logic elements	8,238 / 21,280 (39 %)
Total combinational functions	6,234 / 21,280 (29 %)
Dedicated logic registers	5,156 / 21,280 (24 %)
Total registers	5156
Total pins	90 / 167 (54 %)
Total virtual pins	0
Total memory bits	32,258 / 774,144 (4 %)
Embedded Multiplier 9-bit elements	0 / 80 (0 %)
Total GXB Receiver Channel PCS	0 / 4 (0 %)
Total GXB Receiver Channel PMA	0 / 4 (0 %)
Total GXB Transmitter Channel PCS	0 / 4 (0 %)
Total GXB Transmitter Channel PMA	0 / 4 (0 %)
Total PLLs	0 / 4 (0 %)
Device	EP4CGX22CF19C6
Timing Models	Final

Figure 16 Synthesis Result

The design can run at max frequencies of 83.55 MHz.

Slow 1200mV 85C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	83.55 MHz	83.55 MHz	CLK	

Figure 17 Maximum Operating Frequency

CONCLUSION

After a plenty of time studying and researching with the instruction of Dr. Vo Le Cuong and Mr. Le Van Kieu Quy, we have finish the project: “Design FFT/IFFT 128 points IP core”. Through this project, we have earned some results:

- Understand the structure of Discrete Fourier Transform
- Implement the Fast Fourier Transform in Verilog code and understand the operation of each block and their connections.
- Learn how to verify the results with the use of Python, learn more about coding skills.

We sincerely thank the teacher for having a feedback to our project. This will help us a lot for the further idea in the future.

REFERENCES

[1] "Pipelined FFT/IFFT 128 points (Fast Fourier Transform) IP Core User Manual". Link: pipelined_fft_128/fft128_um.pdf at master · freecores/pipelined_fft_128 · GitHub

[2] Pong P.Chu, "FPGA Prototyping by Verilog Examples"

[3] H.J.Nussbaumer, "Fast Fourier Transform and Convolution Algorithms"

[4] Intel User Guide FFT IP Core

[5] Shuang Zhang, "The design and implementation of FFT algorithm based on the Xilinx FPGA IP Core"