# Design and Implementation of an 8 point FFT using Verilog

Kevin Mathew(22446; kevinmathew@iisc.ac.in), Joshua Thomas Mathew( 22254, joshuamathew@iisc.ac.in) , DESE, IISc


## ABSTRACT

***This paper presents the design and implementation of an 8-point Decimation in Time (DIT) Fast Fourier Transform (FFT) algorithm using verilog. The FFT is a crucial algorithm in signal processing applications, and its efficient implementation on FPGA promises high-speed and real-time signal analysis. The paper discusses the algorithmic considerations, design methodology, and optimization techniques employed in realizing the FFT on an FPGA platform. Performance metrics, such as chip area, timing constraints and power consumption, are evaluated to demonstrate the effectiveness of the proposed implementation. The implemented code is run on various open source tools and a proper analysis is conducted.***


## I INTRODUCTION

The Discrete Fourier Transform (DFT) is a mathematical technique used to analyze the frequency content of discrete signals. It transforms a sequence of complex numbers, which represent the samples of a time-domain signal, into another sequence of complex numbers representing the signal's frequency components. The DFT is widely employed in various fields, including signal processing, communications, and image processing, to extract valuable information about the spectral characteristics of a signal.

## II MATHEMATICAL ANALYSIS

The DFT is a linear transformation of the vector xn (the time domain signal samples) to the vector Xm (the set of coefficients of component sinusoids of time domain signal) using

$$X_m = \sum_{n=0}^{N-1} x_n w^{nm} \tag{1}$$

where N is the size of the vectors and w are the roots of unity or twiddle factors as we call them in this paper[2]. For such a computation of DFT, N*N computations are required. When the size of N increases, this leads to lot of computations and the system get's slow and complex. So Cooley and Tuckey came up with an algorithm which is known as the Fast Fourier Transform (FFT), which is significantly faster and can be done with Nlog(N) computations. The idea is that the DFT can be split into smaller DFT's. We apply our computation to this smaller DFT, which significantly reduces the computation time. Also using some tricks regarding the twiddle factor, we just need upto the third power of w for any calculation that we will undergo. There are multiple algorithms to split the DFT . Here we will use the Decimation in Time (DIT) split which is given by:

$$X_m = \sum_{n=0}^{N/2-1} x_{2n} w^{2nm} + \sum_{n=0}^{N/2-1} x_{2n+1} w^{2nm} \tag{2}$$

There are other algorithms used for splitting DIT like the Winograd Algorithm, but they are mathematically very complex and difficult to implement and therefore we will stick with the simple Cooley Tuckey Algorithm. The Matrix Form of a DIT is given by:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & w^8 & w^{10} & w^{12} & w^{14} \\ 1 & w^3 & w^6 & w^9 & w^{12} & w^{15} & w^{16} & w^{21} \\ 1 & w^4 & w^8 & w^{12} & w^{16} & w^{20} & w^{24} & w^{28} \\ 1 & w^5 & w^{10} & w^{15} & w^{20} & w^{25} & w^{30} & w^{35} \\ 1 & w^6 & w^{12} & w^{18} & w^{24} & w^{30} & w^{36} & w^{42} \\ 1 & w^7 & w^{14} & w^{21} & w^{28} & w^{35} & w^{42} & w^{49} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

Now we re-order the matrix according to the DIT split and we seperate the odd and even indices and we continue this process till we get a 2X2 block. Now the higher powers of w can be manipulated and written as lower powers of w by the equation:

$$w^n = w^{n+Nk} \tag{3}$$

Where N=8 and k=0,1,2..

$$w^n = -w^{n+N/2} \tag{4}$$

Thus the simplified matrix can be obtained as:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & w^2 & -w^2 & w & -w & w^3 & -w^3 \\ 1 & 1 & -1 & -1 & w^2 & w^2 & -w^2 & -w^2 \\ 1 & -1 & -w^2 & w^2 & w^3 & -w^3 & w & -w \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & w^2 & -w^2 & -w & w & -w^3 & w^3 \\ 1 & 1 & -1 & -1 & -w^2 & -w^2 & w^2 & w^2 \\ 1 & -1 & -w^2 & w^2 & -w^3 & w^3 & -w & w \end{pmatrix} \begin{pmatrix} x_0 \\ x_4 \\ x_2 \\ x_6 \\ x_1 \\ x_5 \\ x_3 \\ x_7 \end{pmatrix}$$

Thus we need only till the third power of w and not the 49th. Also we notice that the input matrix is now in bit-reversed order. This is illustrated with the help of the table given below:

| Index | Binary | Bit reversed Index | Binary |
|---|---|---|---|
| 0 | 000 | 0 | 000 |
| 1 | 001 | 4 | 100 |
| 2 | 010 | 2 | 010 |
| 3 | 011 | 6 | 110 |
| 4 | 100 | 1 | 001 |
| 5 | 101 | 5 | 101 |
| 6 | 110 | 3 | 011 |
| 7 | 111 | 7 | 111 |

Table 1: Bit Reversal of Indices

The implementation of the matrix operation can be explained well with the help of the signal flow graph given below:

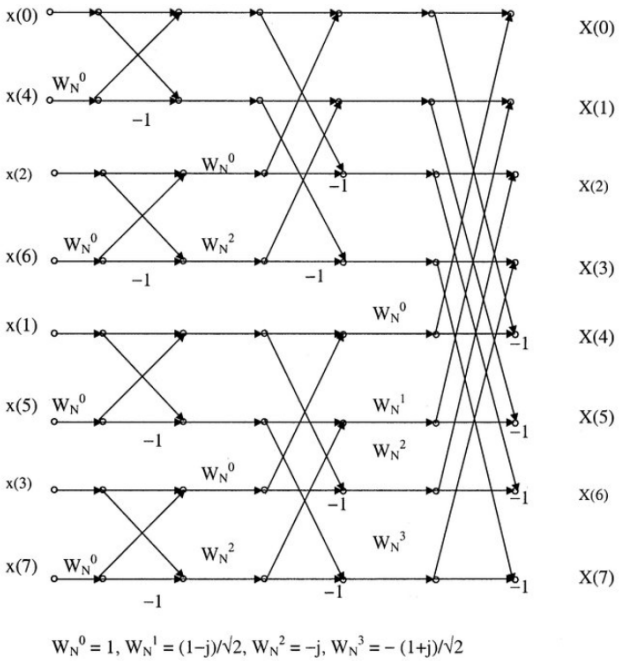


Figure 1: Signal Flow graph of an 8 point DIT FFT (Source: [6]

## III ALGORITHM AND HARDWARE IMPLEMENTATION

In this section we will describe in detail the different hardware blocks required for the computation of FFT. From the Signal flow digram given above, we can see that there are addition and multiplication operations happening multiple times. In other words, there is butterfly operation that is taking place where the inputs are multipled with complex twiddle factors and then it undergoes addition or subtraction. So for this project, we have made 1 module for Additon,which is a 4 bit carry Bypass adder, and a multipler, which helps in multiplying any number with 0.707 (twiddle factor) and we have a butterfly unit, which is the combination of multiplication and addition operations. So in a big picture,in the FFT module, we are instantiating these modules as per requirement.

### *3.1 4 bit Carry Bypass adder*

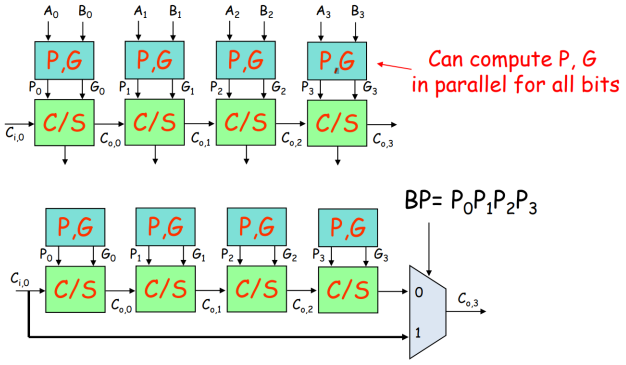


Figure 2: Block diagram of a 4 bit Carry bypass adder ( Source:6.111: Introductory Digital Systems Labaratory, MiT, Fall 2019

In our code, we have used 4 bit adders.We have four such adders present in our 16 bit adder module. From 4 bits and above, we see the carry bypass adder to have higher performance when comapared to the traditional ripple carry adder. Also a 4 bit carry bypass adder takes less area and consumes less power when compared to a 16 bit carry bypass adder. Since our project here is only dealing with 16 bit numbers, 4 bit adder is sufficient. When the number of points in an fft becomes large like 1024, then we definitely need bigger adders.

### *3.2 Multiplier unit*

While multiplying with twiddle factors, a multiplication of 0.707 is present. So a module need to be made for this multiplication. 0.707 can be approximated by the the equation given below:

$$0.707 = \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} + \frac{1}{2^{10}} \quad (5)$$

This multiplication is done thus with a series of right shift operation and then adding the results together.

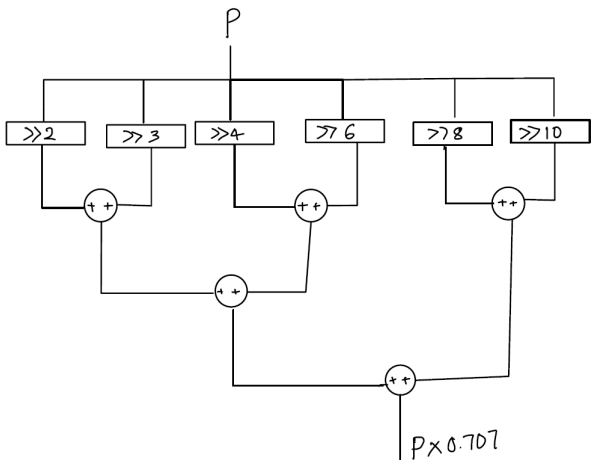


Figure 3: Block diagram of multiplier

### 3.3 *Butterfly unit*

The block diagram for a butterfly unit is given below in Fig.4

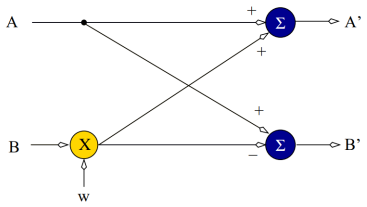


Figure 4: Butterfly Unit Block Diagram ( Source: [2])

We can see addition , subtraction and multiplication operations taking place. We have used 5 such units for our project. We have stored the twiddle factors and based on a select input the appropriate twiddle factor is selected for that particular stage in the FFT computation.

## IV RESULTS AND VALIDATION

So intitially the verilog code (FFT.v and FFT_tb.v are given in the appendix) is run on Icarus Verilog ( an Open source verilog compiler). The files FFT.v and FFT_tb.v are stored in a text file named ”filelist”. Then the following commands are run:
a) iverilog -o FFT FFT.v
b) vvp FFT
The output is then displayed on the command prompt. The output for Example appears as 965 instead of 9.65. This is due to the display function we are using. In reality, the 8 LSB bits are kept as fractional parts of the number.

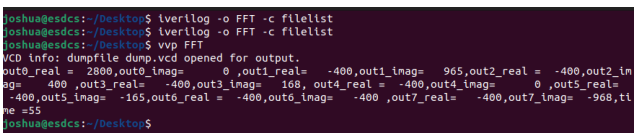


Figure 5: FFT output sample

For Fig.5, the inputs were [0,1,2,3,4,5,6,7], as given in the project statement. The same inputs were tested in Matlab, it's results were obtained. The comparision of the 2 results are given in the table below:

| Output | Matlab | FFT hardware | Error( %) |
|---|---|---|---|
| OUT0 | 28+0i | 28+0i | 0 |
| OUT1 | -4+9.6569i | -4+9.65i | 0.07 |
| OUT2 | -4+4i | -4+4i | 0 |
| OUT3 | -4+1.6569i | -4+1.68i | -0.01 |
| OUT4 | -4+0i | -4+0i | 0 |
| OUT5 | -4-1.6569i | -4-1.65i | 0.41 |
| OUT6 | -4-4i | -4-4i | 0 |
| OUT7 | -4-9.6569i | -4-9.68i | -0.23 |

Table 2: Comparision with results obtained from MATLAB

We can see that the errors are very less and the FFT hardware is giving satisfactory performance. We shall consider one more example where the input is :[ 1+2i,2+3i,3+4i,4+5i,5+6i,6+7i,7+8i,8+9i]

| Output | Matlab | FFT hardware | Error( %) |
|---|---|---|---|
| OUT0 | 36+44i | 36+44i | 0 |
| OUT1 | -13.6569+5.6569i | -13.68+5.65i | -0.126 |
| OUT2 | -8+0i | -8+0i | 0 |
| OUT3 | -5.6569-2.3431i | -5.68-2.32i | -0.205 |
| OUT4 | -4-4i | -4-4i | 0 |
| OUT5 | -2.3431-5.6569i | -2.32-5.65i | 0.248 |
| OUT6 | 0-8i | 0-8i | 0 |
| OUT7 | 5.6569-13.6569i | 5.68-13.68i | -0.204 |

Table 3: Comparision with results obtained from MATLAB

Next, we wanted to analyse the timing by running this file in GTKwave, where we can see all the pulses we have in our project. The following command had to be typed:
gtkwave FFT_tb.vcd. Then the GTKwave window opens and select the required signals.

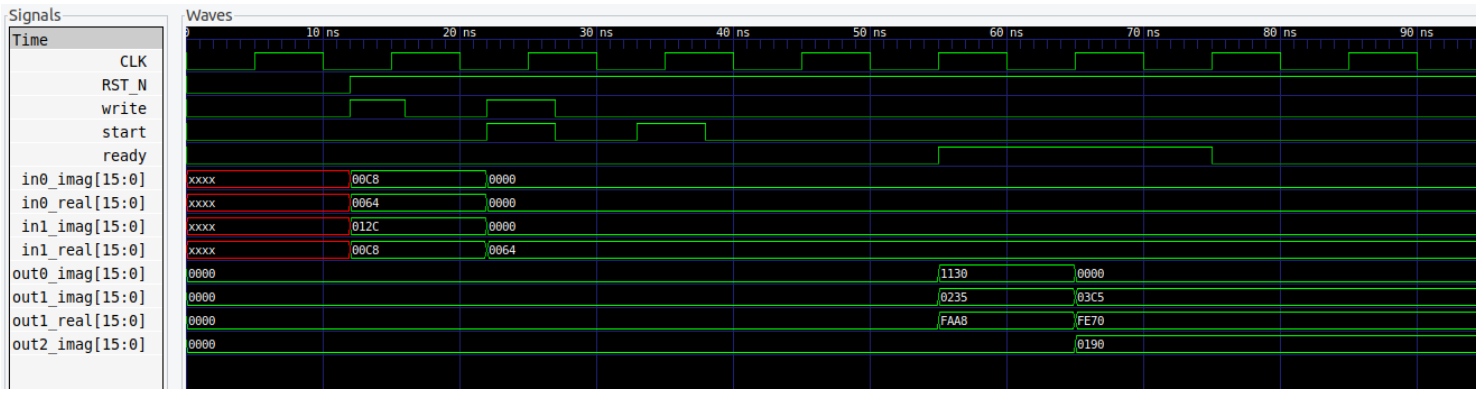


Figure 6: Simulation result from GTK wave

In the above figure, we can see 2 inputs being processed in a pipeline. At first RESET_N pulse is low and all the input and output registers are intialised to zero.Then the RESET_N pulse is made High. Then the Write pulse is made High and the first set of inputs are stored in the input registers. Then the Start pulse is high, indicating the beginning of FFT computation for the first input. AT this same time, the Write pulse goes High for storing the second set of inputs to the input registers. **After 3 clock cycles, the Ready input is High indicating that the output is ready. So 3 clock cycles are required for one FFT Computation**.After one more clock cycle, the 2nd output has also arrived.

Next we will use Yosys (Yosys Open SYnthesis Suite) which is an open-source RTL synthesis tool for digital design descriptions written in Verilog-2005.[5]

We have to write a synthesis file synth.ys (which is given in the appendix). The top level module has to be selected in the synth.ys file which is "FFT_wrapper" in our project. Also use the NangateOpenCellLibrary_typical.lib. Then type the following command in command prompt:

yosys synth.ys

Fig.7 shows the results obtained after synthesis **If we open the file "NangateOpenCellLibrary_typical.lib. " in a text editor, we can see the contents of the file. From the file we get the following data: the process corner is given as "TypTyp", which implies typical. We are neither using fast nor slow. The operating voltage is 1.1V. The operating temperature is 25 $^\circ$ C.**

```
=== design hierarchy ===

   FFT_wrapper                     1
     FFT                           1
       BFU                         5
         carry_look_ahead_16bit    8
           carry_look_ahead_4bit     4
         complex_mult              2
           carry_look_ahead_16bit      5
             carry_look_ahead_4bit       4
       carry_look_ahead_16bit       28
         carry_look_ahead_4bit       4

   Number of wires:              31325
   Number of wire bits:          31325
   Number of public wires:       22260
   Number of public wire bits:   22260
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:              14619
     $_DLATCH_P_                   320
     AND2_X1                       949
     AOI21_X1                      944
     BUF_X1                          5
     DFFR_X1                      1280
     INV_X1                         10
     MUX2_X1                       866
     NAND2_X1                     1736
     NAND3_X1                      320
     NAND4_X1                      160
     NOR2_X1                       477
     NOR3_X1                       472
     OAI21_X1                      944
     OR2_X1                        472
     XNOR2_X1                     1888
     XOR2_X1                      3776

   Chip area for top module '\FFT_wrapper': 23813.384000

Warnings: 2077 unique messages, 2141 total
End of script. Logfile hash: d9de87047a
CPU: user 1.99s system 0.13s, MEM: 41.75 MB total, 35.74 MB resident
Yosys 0.9 (git sha1 1979e0b)
Time spent: 40% 2x write_verilog (0 sec), 13% 10x opt_clean (0 sec), ...
joshua@esdcs:~/Desktop$
```

Figure 7: Yosys Synthesis results

**From Fig.7, we can see that the total chip area is 23813.384 $\mu m^2$**

Finally we will run the code in OpenSTA.OpenSTA is an open-source gate-level static timing analyzer which can be used to verify timing characteristics of a synthesized digital

design. It uses various commands to read the design, specify timing constraints and print timing reports.[5] The following commands are to be typed:
sta
read_liberty NangateOpenCellLibrary_typical.lib
read_verilog synth.v
link_design FFT_wrapper
create_clock -name CLK -period 2 { CLK }
set_power_activity -input -activity 0.5
set_power_activity -global -activity 0.5
report_checks
report_power Fig.8 shows the results of timing analysis in OpenSTA.

```
Delay    Time   Description
---------------------------------------------------------
  0.00    0.00   clock CLK (rise edge)
  0.00    0.00   clock network delay (ideal)
  0.00    0.00 ^ fft/_3136_/CK (DFFR_X1)
  0.09    0.09 v fft/_3136_/Q (DFFR_X1)
  0.06    0.15 v fft/cla_fft_1/_03_/Z (XOR2_X1)
  0.03    0.18 v fft/cla_fft_1/cla1/_19_/ZN (AND2_X1)
  0.05    0.24 ^ fft/cla_fft_1/cla1/_22_/ZN (AOI21_X1)
  0.03    0.27 v fft/cla_fft_1/cla1/_23_/ZN (OAI21_X1)
  0.04    0.31 ^ fft/cla_fft_1/cla1/_25_/ZN (AOI21_X1)
  0.02    0.33 v fft/cla_fft_1/cla1/_28_/ZN (OAI21_X1)
  0.07    0.40 v fft/cla_fft_1/_00_/Z (MUX2_X1)
  0.05    0.45 ^ fft/cla_fft_1/cla2/_22_/ZN (AOI21_X1)
  0.03    0.48 v fft/cla_fft_1/cla2/_23_/ZN (OAI21_X1)
  0.04    0.52 ^ fft/cla_fft_1/cla2/_25_/ZN (AOI21_X1)
  0.02    0.54 v fft/cla_fft_1/cla2/_28_/ZN (OAI21_X1)
  0.07    0.61 v fft/cla_fft_1/_01_/Z (MUX2_X1)
  0.05    0.66 ^ fft/cla_fft_1/cla3/_22_/ZN (AOI21_X1)
  0.03    0.69 v fft/cla_fft_1/cla3/_23_/ZN (OAI21_X1)
  0.04    0.73 ^ fft/cla_fft_1/cla3/_25_/ZN (AOI21_X1)
  0.02    0.75 v fft/cla_fft_1/cla3/_28_/ZN (OAI21_X1)
  0.07    0.81 v fft/cla_fft_1/_02_/Z (MUX2_X1)
  0.05    0.86 ^ fft/cla_fft_1/cla4/_22_/ZN (AOI21_X1)
  0.03    0.89 v fft/cla_fft_1/cla4/_23_/ZN (OAI21_X1)
  0.04    0.94 ^ fft/cla_fft_1/cla4/_25_/ZN (AOI21_X1)
  0.04    0.98 ^ fft/cla_fft_1/cla4/_32_/ZN (XNOR2_X1)
  0.00    0.98 ^ fft/_2831_/D (DFFR_X1)
          0.98   data arrival time

  2.00    2.00   clock CLK (rise edge)
  0.00    2.00   clock network delay (ideal)
  0.00    2.00   clock reconvergence pessimism
          2.00 ^ fft/_2831_/CK (DFFR_X1)
 -0.04    1.96   library setup time
          1.96   data required time
---------------------------------------------------------
          1.96   data required time
         -0.98   data arrival time
---------------------------------------------------------
          0.98   slack (MET)
```

Figure 8: OpenSTA timing results

**In Fig.8 we can see a positive slack of 0.98. We have ran this at a clock speed of 500 MHz. So Maximum clock speed in which our timing constraints won't we violated is given by:**

$$\frac{1}{2-0.98} = 980MHz \tag{6}$$

Fig 9 shows the power analysis from openSTA

```
OpenSTA> set_power_activity -input -activity 0.5
OpenSTA> report_power
Group                  Internal  Switching    Leakage      Total
                          Power      Power      Power      Power
----------------------------------------------------------------
Sequential             2.76e-02   1.87e-03   1.10e-04   2.95e-02  54.4%
Combinational          1.99e-02   4.50e-03   3.81e-04   2.48e-02  45.6%
Macro                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
Pad                    0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
----------------------------------------------------------------
Total                  4.75e-02   6.36e-03   4.91e-04   5.43e-02 100.0%
                          87.4%      11.7%       0.9%
OpenSTA>
```

Figure 9: OpenSTA power results

**From the above Figure, the total power consumed is 54.3 mW. The power calculation was done at the maximum possible operating frequency (980 MHz).Total Internal Power is 47.5 mW. Switching power is around 6.36mW. Leakage power is at the least at 0.491 mW. The total number of clock cycles required for 1 FFT computation is 3. So the energy consumed per FFT computation is:**
**Energy= Total Power* 3* (1/fmax)=0.166 nJ**
**As the frequency decreases, the energy consumption increases, whereas the total power consumption decreases. The same analysis was conducted again with frequency as 10MHz. Total Power consumption became 1.05mW, but the energy consumed for 1 FFT computation became 0.315 nJ.**

## V CONCLUSIONS AND FUTURE EXTENSIONS OF THE PROJECT

8 point FFT has been implemented in verilog and tested using various open source tools like Icarus Verilog and synthesis was conducted and thus were able to conduct a proper timing and power analysis of the entire chip. Normal applications used very large point FFT, like 1024. This work can be further extended by increasing the number of points. When the number of points, we need a separate address generation mechanism which we have not used for this application. Also the pipelining and other timing constraints will become more complicated. **Another extension to the work is to obtain the inverse FFT. Inverse FFT is obtaining the time domain data back from frequency domain. We can employ the same signal flow diagram with few changes. The complex conjugates of the twiddle factors have to be used. Scaling factors of N, has to applied to compensate what happened during FFT. Also for the butterfly unit, the addition and subtraction are reversed.**
**The main aim in our design was to increase the throughput. We have tested for multiple streams of inputs and as seen in Fig.6, we can see that 3 clock cycles are needed for the first output to come out. After that,we will receive an output for every clock cycle in the case of a stream of inputs. Area is much for this design. We have used 5 butterfly units running parallely and this will cost lot of power and area. If we would have used just one stage of FFT hardware and then sequentially executed the other stages of FFT computation using the same hardware, that would have resulted**

**in lesser power and area, but the total number of clock cycles required for FFT computation will increase.**

## REFERENCES


[1] An Algorithm for the machine calculation of complex Fourier Series - J Cooley, J Tuckey

[2] The Fast Fourier Transform in Hardware:A Tutorial Based on an FPGA Implementation- G William Slade (MiT)

[3] How to implement an FFT with an FPGA from scratch- The EE View

[4] Digital Integrated Circuits- J M Rabaey, Chandrakasan, Nikolic

[5] K. Maharatna, E. Grass and U. Jagdhold, "A 64-point Fourier transform chip for high-speed wireless LAN application using OFDM," in IEEE Journal of Solid-State Circuits, vol. 39, no. 3, pp. 484-493, March 2004, doi: 10.1109/JSSC.2003.822776.

[6] Verilog simulation and synthesis Tutorial ( ESDCS-2023, DESE, IISc)

[7] FPGA Implementation of 8-Point FFT by Dr. Shirshendu Roy


## A APPENDIX

The main Verilog file used for this project is given bellow

```
////////////////////////////////

//16-bit Carry Look Ahead Adder

////////////////////////////////


`timescale 1ns / 1ps

module carry_look_ahead_16bit(a,b, carry_in,
    sum,cout);

  input signed[15:0] a,b;

input carry_in;

output signed[15:0] sum;

output cout;

wire c1,c2,c3,c4,c5,c6,c7;

  wire [15:0] b_int;

  wire [6:0]p;

  wire c1_p,c2_p,c3_p,c4_p,c5_p,c6_p,c7_p;


  assign b_int = (b ^ {16{carry_in}});//
      negate b for subtraction

  carry_look_ahead_4bit cla1 (.a(a[3:0]),
      .b(b_int[3:0]), .cin((carry_in)),
      .sum(sum[3:0]),
      .cout(c1),.p0p1p2p3(p[0]));


  assign c1_p = (p[0] & carry_in) || ((!p[0])
      & c1);//mux for carry propogation

  carry_look_ahead_4bit cla2 (.a(a[7:4]),
      .b(b_int[7:4]), .cin(c1_p),
      .sum(sum[7:4]),
      .cout(c2),.p0p1p2p3(p[1]));


  assign c2_p = (p[1] & c1) || ((~p[1])&c2);

  carry_look_ahead_4bit cla3(.a(a[11:8]),
      .b(b_int[11:8]), .cin(c2_p),
      .sum(sum[11:8]),
      .cout(c3),.p0p1p2p3(p[2]));


  assign c3_p = (p[2] & c2) || ((~p[2])&c3);

  carry_look_ahead_4bit cla4(.a(a[15:12]),
      .b(b_int[15:12]), .cin(c3_p),
      .sum(sum[15:12]),
      .cout(c4),.p0p1p2p3(p[3]));


endmodule

////////////////////////////////////////////////////

//4-bit Carry Look Ahead Adder

////////////////////////////////////////////////////


module carry_look_ahead_4bit(a,b, cin,
    sum,cout,p0p1p2p3);

input [3:0] a,b;

input cin;

output [3:0] sum;

output cout;

  output p0p1p2p3;

  wire [3:0] p,g,c;
```

```
assign p=a^b;//propagate

assign g=a&b; //generate



//carry=gi + Pi.ci



assign c[0]=cin;

assign c[1]= g[0]|(p[0]&c[0]);

assign c[2]= g[1] | (p[1]&g[0]) |
    p[1]&p[0]&c[0];

assign c[3]= g[2] | (p[2]&g[1]) |
    p[2]&p[1]&g[0] | p[2]&p[1]&p[0]&c[0];

assign cout= g[3] | (p[3]&g[2]) |
    p[3]&p[2]&g[1] | p[3]&p[2]&p[1]&g[0] |
    p[3]&p[2]&p[1]&p[0]&c[0];

assign sum=p^c;

  assign p0p1p2p3=p[0]&p[1]&p[2]&p[3];



endmodule

////////////////////////////////////////////////////////////
//Complex multiplier
////////////////////////////////////////////////////////////

//`timescale 1ns / 1ps
module complex_mult(multiplicant,product);


  input signed [15:0]multiplicant;

  output signed [15:0]product;

  wire [15:0]sum1,sum2,sum3,sum4,sum5;

  wire cout1,cout2,cout3,cout4,cout5;



  carry_look_ahead_16bit cla_16_1
 (.a({multiplicant[15],multiplicant[15:1]}),
.b({multiplicant[15],multiplicant[15],
multiplicant[15],multiplicant[15:3]}),

 .carry_in(1'b0), .sum(sum1),.cout(cout1));


  carry_look_ahead_16bit cla_16_2
      (.a({multiplicant[15],multiplicant[15],
multiplicant[15],multiplicant[15],
multiplicant[15:4]}),
.b({multiplicant[15],
multiplicant[15],multiplicant[15],
multiplicant[15],multiplicant[15],
multiplicant[15],multiplicant[15:6]}),
 .carry_in(1'b0), .sum(sum2),.cout(cout2));


    carry_look_ahead_16bit cla_16_3
 (.a({multiplicant[15],multiplicant[15],
multiplicant[15],
multiplicant[15],multiplicant[15]
,multiplicant[15],multiplicant[15],
multiplicant[15],multiplicant[15:8]}),
.b({multiplicant[15],
multiplicant[15],multiplicant[15],
multiplicant[15],
multiplicant[15],multiplicant[15]
,multiplicant[15],
multiplicant[15],multiplicant[15]
,multiplicant[15],
multiplicant[15:10]}),
 .carry_in(1'b0), .sum(sum3),.cout(cout3));

   carry_look_ahead_16bit cla_16_4 (.a(sum1),
.b(sum2), .carry_in(1'b0),
    .sum(sum4),.cout(cout4));

  carry_look_ahead_16bi
t cla_16_5 (.a(sum3),.b(sum4),
    .carry_in(1'b0), .sum(sum5),.cout(cout5));


   assign product=sum5;


endmodule

////////////////////////////

//Butterfly Unit

////////////////////////////

//`timescale 1ns / 1ps

module BFU(A_real,A_imag,B_real,
B_imag,sel_w,X0_real, X0_imag, X1_real,
    X1_imag);


  input signed[15:0]
```

```
     A_real,A_imag,B_real,B_imag;

  input [1:0]sel_w;

  output signed [15:0] X0_real,
X0_imag, X1_real, X1_imag;

  wire [15:0] sum1,diff1,
B_sum_prod,B_sum_prod_n,
B_diff_prod,B_real_n;

  reg signed [15:0] B_real_0,
B_imag_0,B_real_1,B_imag_1,
Bxw_real,Bxw_imag;

  wire cout1,cout2,cout3,cout4;



  always@(B_real,B_imag,sel_w)

    begin //Decoder to direct B based on w
        (sel_w)

      case(sel_w)

        2'b00,2'b10 :

          begin

          B_real_0 <= B_real;

          B_imag_0 <= B_imag;

          end

        2'b01,2'b11 :

          begin

          B_real_1 <= B_real;

          B_imag_1 <= B_imag;

          end

        endcase

       end

  carry_look_ahead_16bit cla_bfu_1
(.a(B_real_1),.b(B_imag_1), .carry_in(1'b0),
.sum(sum1),.cout(cout1));//b_real+b_imag



  carry_look_ahead_16bit cla_bfu_2
 (.a(B_imag_1),.b(B_real_1), .carry_in(1'b1),
.sum(diff1),.cout(cout2));//b_imag-b_real
```

```
 complex_mult cmult_1
(.multiplicant(sum1),.product(B_sum_prod));//0.707*(s



 complex_mult cmult_2
(.multiplicant(diff1),.product(B_diff_prod));//0.707*



 carry_look_ahead_16bit cla_bfu_3
     (.a(16'b0),.b(B_sum_prod),
.carry_in(1'b1),
    .sum(B_sum_prod_n),.cout(cout3));//-0.707*(sum)



 carry_look_ahead_16bit cla_bfu_4
     (.a(16'b0),.b(B_real_0),
.carry_in(1'b1),
    .sum(B_real_n),.cout(cout3));//-b_real



 always@(*)

   begin

     case(sel_w)

       2'b00:

         begin

           Bxw_real <= B_real_0;

           Bxw_imag <= B_imag_0;

         end

       2'b01:

         begin

           Bxw_real <= B_sum_prod;

           Bxw_imag <= B_diff_prod;

         end

       2'b10:

         begin

           Bxw_real <= B_imag_0;

           Bxw_imag <= B_real_n;

         end
```

```
      2'b11:
        begin
          Bxw_real <= B_diff_prod;
          Bxw_imag <= B_sum_prod_n;
        end
     endcase
    end


 carry_look_ahead_16bit cla_bfu_5
     (.a(A_real),.b(Bxw_real),
     .carry_in(1'b0),
     .sum(X0_real),.cout());//A'_real

 carry_look_ahead_16bit cla_bfu_6
     (.a(A_imag),.b(Bxw_imag),
     .carry_in(1'b0),
     .sum(X0_imag),.cout());//A'_imag

 carry_look_ahead_16bit cla_bfu_7
     (.a(A_real),.b(Bxw_real),
     .carry_in(1'b1),.sum(X1_real),.cout());//B'_real

 carry_look_ahead_16bit cla_bfu_8
     (.a(A_imag),.b(Bxw_imag),
     .carry_in(1'b1),
     .sum(X1_imag),.cout());//B'_imag



endmodule



////////////////////////////////////

//FFT UNIT

////////////////////////////////////

//`timescale 1ns / 1ps

module FFT(In_real0,In_real1,In_real2,
In_real3,In_real4,
In_real5,In_real6,In_real7,In_imag0,In_imag1,
In_imag2,In_imag3,In_imag4,
In_imag5,In_imag6,In_imag7,reset_n,clk
,write,start_fft,Out_real0,Out_real1,Out_real2
,Out_real3,Out_real4,Out_real5,Out_real6
,Out_real7,Out_imag0,Out_imag1,Out_imag2
,Out_imag3,Out_imag4,Out_imag5,
Out_imag6,Out_imag7,fft_ready);


 input signed
     [15:0]In_real0,In_real1,In_real2,
In_real3,In_real4,In_real5,
In_real6,In_real7,
In_imag0,In_imag1,
In_imag2,In_imag3,
In_imag4,In_imag5,
In_imag6,In_imag7;

 input clk,reset_n,write,start_fft;

 output reg fft_ready;

 output reg signed [15:0]Out_real0,Out_real1,
Out_real2,Out_real3,
Out_real4,Out_real5,
Out_real6,Out_real7,
Out_imag0,Out_imag1,
Out_imag2,Out_imag3,
Out_imag4,Out_imag5,
Out_imag6,Out_imag7;

 //reg signed [15:0]
     stage1_op_real[7:0],stage1_op_imag[7:0],
stage2_ip_real[7:0],
stage2_ip_imag[7:0],stage2_op_real[7:0],
stage2_op_imag[7:0],
stage3_ip_real[7:0],stage3_ip_imag[7:0];

 wire [15:0] stage1_op_real[7:0],
stage1_op_imag[7:0],stage2_op_real[7:0]
,stage2_op_imag[7:0],
stage3_op_real[7:0],
stage3_op_imag[7:0];

 reg signed [15:0]stage1_ip_real[7:0],
stage1_ip_imag[7:0],stage2_ip_real[7:0]
,stage2_ip_imag[7:0],
stage3_ip_real[7:0],
stage3_ip_imag[7:0],
Input_real_reg[7:0],
Input_imag_reg[7:0];

 reg [1:0]i;//To be used to generate ready
     signal when output ready
 reg strt_s1,strt_s2,strt_s3;

 always@(posedge clk or negedge reset_n)
  begin
    if(!reset_n)
     begin
     Input_real_reg[0]<=15'd0;
    Input_real_reg[1]<=15'd0;
    Input_real_reg[2]<=15'd0;
    Input_real_reg[3]<=15'd0;
```

```
    Input_real_reg[4]<=15'd0;
    Input_real_reg[5]<=15'd0;
    Input_real_reg[6]<=15'd0;
    Input_real_reg[7]<=15'd0;
    Input_imag_reg[0]<=15'd0;
    Input_imag_reg[1]<=15'd0;
    Input_imag_reg[2]<=15'd0;
    Input_imag_reg[3]<=15'd0;
    Input_imag_reg[4]<=15'd0;
    Input_imag_reg[5]<=15'd0;
    Input_imag_reg[6]<=15'd0;
    Input_imag_reg[7]<=15'b0;
    strt_s1<=1'b0;
    fft_ready<=1'b0;
     end
     else if (write)
      begin
        Input_real_reg[0]<=In_real0;
        Input_real_reg[1]<=In_real1;
        Input_real_reg[2]<=In_real2;
        Input_real_reg[3]<=In_real3;
        Input_real_reg[4]<=In_real4;
        Input_real_reg[5]<=In_real5;
        Input_real_reg[6]<=In_real6;
        Input_real_reg[7]<=In_real7;
        Input_imag_reg[0]<=In_imag0;
        Input_imag_reg[1]<=In_imag1;
        Input_imag_reg[2]<=In_imag2;
        Input_imag_reg[3]<=In_imag3;
        Input_imag_reg[4]<=In_imag4;
        Input_imag_reg[5]<=In_imag5;
        Input_imag_reg[6]<=In_imag6;
        Input_imag_reg[7]<=In_imag7;
      end

end


always@(posedge clk or negedge reset_n)
begin
  if(!reset_n)
    begin
    stage1_ip_real[0]<=15'd0;
  stage1_ip_real[1]<=15'd0;
  stage1_ip_real[2]<=15'd0;
  stage1_ip_real[3]<=15'd0;
  stage1_ip_real[4]<=15'd0;
  stage1_ip_real[5]<=15'd0;
  stage1_ip_real[6]<=15'd0;
  stage1_ip_real[7]<=15'd0;
  stage1_ip_imag[0]<=15'd0;
  stage1_ip_imag[1]<=15'd0;
  stage1_ip_imag[2]<=15'd0;
  stage1_ip_imag[3]<=15'd0;
  stage1_ip_imag[4]<=15'd0;
  stage1_ip_imag[5]<=15'd0;
  stage1_ip_imag[6]<=15'd0;
  stage1_ip_imag[7]<=15'b0;

    end
  else if (start_fft)
     begin
        stage1_ip_real[0]<=Input_real_reg[0];
        stage1_ip_real[1]<=Input_real_reg[1];
        stage1_ip_real[2]<=Input_real_reg[2];
        stage1_ip_real[3]<=Input_real_reg[3];
        stage1_ip_real[4]<=Input_real_reg[4];
        stage1_ip_real[5]<=Input_real_reg[5];
        stage1_ip_real[6]<=Input_real_reg[6];
        stage1_ip_real[7]<=Input_real_reg[7];
        stage1_ip_imag[0]<=Input_imag_reg[0];
        stage1_ip_imag[1]<=Input_imag_reg[1];
        stage1_ip_imag[2]<=Input_imag_reg[2];
        stage1_ip_imag[3]<=Input_imag_reg[3];
        stage1_ip_imag[4]<=Input_imag_reg[4];
        stage1_ip_imag[5]<=Input_imag_reg[5];
        stage1_ip_imag[6]<=Input_imag_reg[6];
        stage1_ip_imag[7]<=Input_imag_reg[7];
        strt_s1<=1'b1;
      end
   else if(!start_fft)
     strt_s1<=1'b0;

 end

// initially compute stage 1. stage 1
    consists only of basic addition and
    subtraction. For subtraction, carry
    input should be 1


  //stage 1 output 1


carry_look_ahead_16bit cla_fft_1
    (.a(stage1_ip_real[0]),.b(stage1_ip_real[4]),
    .carry_in(1'b0),
    .sum(stage1_op_real[0]),.cout());
    //Real (x0+x4)

carry_look_ahead_16bit cla_fft_2
    (.a(stage1_ip_imag[0]),.b(stage1_ip_imag[4]),
    .carry_in(1'b0),
    .sum(stage1_op_imag[0]),.cout());
    //Imaginary (x0+x4)



  //stage 1 output 2

carry_look_ahead_16bit cla_fft_3
    (.a(stage1_ip_real[0]),.b(stage1_ip_real[4]),
    .carry_in(1'b1),
    .sum(stage1_op_real[1]),.cout());//
    real(x0-x4)

carry_look_ahead_16bit cla_fft_4
    (.a(stage1_ip_imag[0]),.b(stage1_ip_imag[4]),
    .carry_in(1'b1),
    .sum(stage1_op_imag[1]),.cout());//
    Imaginary (x0-x4)
```

```
                                                    .carry_in(1'b1),
                                                    .sum(stage1_op_imag[5]),.cout());//
                                                    Imaginary (x1-x5)

  //stage 1 output 3

carry_look_ahead_16bit cla_fft_5                     //stage 1 output 7
    (.a(stage1_ip_real[2]),.b(stage1_ip_real[6]),
    .carry_in(1'b0),                                carry_look_ahead_16bit cla_fft_13
    .sum(stage1_op_real[2]),.cout());                   (.a(stage1_ip_real[3]),.b(stage1_ip_real[7]),
    //Real (x2+x6)                                      .carry_in(1'b0),
                                                        .sum(stage1_op_real[6]),.cout());
carry_look_ahead_16bit cla_fft_6                        //Real (x3+x7)
    (.a(stage1_ip_imag[2]),.b(stage1_ip_imag[6]),
    .carry_in(1'b0),                                carry_look_ahead_16bit cla_fft_14
    .sum(stage1_op_imag[2]),.cout());                   (.a(stage1_ip_imag[3]),.b(stage1_ip_imag[7]),
    //Imaginary (x2+x6)                                 .carry_in(1'b0),
                                                        .sum(stage1_op_imag[6]),.cout());
                                                        //Imaginary (x3+x7)

  //stage 1 output 4

carry_look_ahead_16bit cla_fft_7                     //stage 1 output 8
    (.a(stage1_ip_real[2]),.b(stage1_ip_real[6]),
    .carry_in(1'b1),                                carry_look_ahead_16bit cla_fft_15
    .sum(stage1_op_real[3]),.cout());//                 (.a(stage1_ip_real[3]),.b(stage1_ip_real[7]),
    real(x2-x6)                                         .carry_in(1'b1),
                                                        .sum(stage1_op_real[7]),.cout());//
carry_look_ahead_16bit cla_fft_8                        real(x3-x7)
    (.a(stage1_ip_imag[2]),.b(stage1_ip_imag[6]),
    .carry_in(1'b1),                                carry_look_ahead_16bit cla_fft_16
    .sum(stage1_op_imag[3]),.cout());//                 (.a(stage1_ip_imag[3]),.b(stage1_ip_imag[7]),
    Imaginary (x2-x6)                                   .carry_in(1'b1),
                                                        .sum(stage1_op_imag[7]),.cout());//
                                                        Imaginary (x3-x7)

  //stage 1 output 5                                // always@(*)
                                                    //   begin
carry_look_ahead_16bit cla_fft_9
    (.a(stage1_ip_real[1]),.b(stage1_ip_real[5]),   always@(posedge clk or negedge reset_n)
    .carry_in(1'b0),
    .sum(stage1_op_real[4]),.cout());                 begin
    //Real (x1+x5)

carry_look_ahead_16bit cla_fft_10                      if(!reset_n)
    (.a(stage1_ip_imag[1]),.b(stage1_ip_imag[5]),
    .carry_in(1'b0),                                    begin
    .sum(stage1_op_imag[4]),.cout());
    //Imaginary (x1+x5)                                 stage2_ip_real[0]<=16'b0;

                                                        stage2_ip_imag[0]<=16'b0;

  //stage 1 output 6                                    stage2_ip_real[1]<=16'b0;

carry_look_ahead_16bit cla_fft_11                       stage2_ip_imag[1]<=16'b0;
    (.a(stage1_ip_real[1]),.b(stage1_ip_real[5]),
    .carry_in(1'b1),                                    stage2_ip_real[2]<=16'b0;
    .sum(stage1_op_real[5]),.cout());//
    real(x1-x5)                                         stage2_ip_imag[2]<=16'b0;

carry_look_ahead_16bit cla_fft_12                       stage2_ip_real[3]<=16'b0;
    (.a(stage1_ip_imag[1]),.b(stage1_ip_imag[5]),
```

```
      stage2_ip_imag[3]<=16'b0;                    //stage 2 computation.

      stage2_ip_real[4]<=16'b0;

      stage2_ip_imag[4]<=16'b0;

      stage2_ip_real[5]<=16'b0;                    //stage 2 output 1

      stage2_ip_imag[5]<=16'b0;                   carry_look_ahead_16bit cla_fft_17
                                                      (.a(stage2_ip_real[0]),.b(stage2_ip_real[2]),
      stage2_ip_real[6]<=16'b0;                       .carry_in(1'b0),
                                                      .sum(stage2_op_real[0]),.cout()); //
      stage2_ip_imag[6]<=16'b0;                       real (stage 1 op1 + stage 1 op3)

      stage2_ip_real[7]<=16'b0;                   carry_look_ahead_16bit cla_fft_18
                                                      (.a(stage2_ip_imag[0]),.b(stage2_ip_imag[2]),
      stage2_ip_imag[7]<=16'b0;                       .carry_in(1'b0),
                                                      .sum(stage2_op_imag[0]),.cout()); //
    end                                               imag (stage 1 op1 + stage 1 op3)

    else

     begin                                         //stage 2 output 3

       stage2_ip_real[0]<=stage1_op_real[0];      carry_look_ahead_16bit cla_fft_19
                                                      (.a(stage2_ip_real[0]),.b(stage2_ip_real[2]),
       stage2_ip_imag[0]<=stage1_op_imag[0];          .carry_in(1'b1),
                                                      .sum(stage2_op_real[2]),.cout()); //
       stage2_ip_real[1]<=stage1_op_real[1];          real (stage 1 op1 - stage 1 op3)

       stage2_ip_imag[1]<=stage1_op_imag[1];      carry_look_ahead_16bit cla_fft_20
                                                      (.a(stage2_ip_imag[0]),.b(stage2_ip_imag[2]),
       stage2_ip_real[2]<=stage1_op_real[2];          .carry_in(1'b1),
                                                      .sum(stage2_op_imag[2]),.cout()); //
       stage2_ip_imag[2]<=stage1_op_imag[2];          imag (stage 1 op1 - stage 1 op3)

       stage2_ip_real[3]<=stage1_op_real[3];

       stage2_ip_imag[3]<=stage1_op_imag[3];       //stage 2 output 2 and 4

       stage2_ip_real[4]<=stage1_op_real[4];

       stage2_ip_imag[4]<=stage1_op_imag[4];       BFU bf1 (.A_real(stage2_ip_real[1]),
                                                  .A_imag(stage2_ip_imag[1]),
       stage2_ip_real[5]<=stage1_op_real[5];      .B_real(stage2_ip_real[3]),
                                                  .B_imag(stage2_ip_imag[3]),
       stage2_ip_imag[5]<=stage1_op_imag[5];      .sel_w(2'b10),.X0_real(stage2_op_real[1]),

       stage2_ip_real[6]<=stage1_op_real[6];             .X0_imag(stage2_op_imag[1]),
                                                             .X1_real(stage2_op_real[3]),
       stage2_ip_imag[6]<=stage1_op_imag[6];                 .X1_imag(stage2_op_imag[3])); //
                                                             10 is selected for sel_w as w^2
       stage2_ip_real[7]<=stage1_op_real[7];                 is the twiddle factor

       stage2_ip_imag[7]<=stage1_op_imag[7];

       strt_s2<=strt_s1;                           // stage 2 output 5 and 7
     end

  end
                                                  //stage 2 output 5
```

```
carry_look_ahead_16bit cla_fft_21
    (.a(stage2_ip_real[4]),.b(stage2_ip_real[6]),
    .carry_in(1'b0),
    .sum(stage2_op_real[4]),.cout()); //
    real (stage 1 op5 + stage 1 op7)

carry_look_ahead_16bit cla_fft_22
    (.a(stage2_ip_imag[4]),.b(stage2_ip_imag[6]),
    .carry_in(1'b0),
    .sum(stage2_op_imag[4]),.cout()); //
    imag (stage 1 op5 + stage 1 op7)



  //stage 2 output 7

carry_look_ahead_16bit cla_fft_23
    (.a(stage2_ip_real[4]),.b(stage2_ip_real[6]),
    .carry_in(1'b1),
    .sum(stage2_op_real[6]),.cout()); //
    real (stage 1 op5 - stage 1 op7)

carry_look_ahead_16bit cla_fft_24
    (.a(stage2_ip_imag[4]),.b(stage2_ip_imag[6]),
    .carry_in(1'b1),
    .sum(stage2_op_imag[6]),.cout()); //
    imag (stage 1 op5 - stage 1 op7)



  //stage 2 output 6 and 8



 BFU bf2 (.A_real(stage2_ip_real[5]),
.A_imag(stage2_ip_imag[5]),
.B_real(stage2_ip_real[7]),
.B_imag(stage2_ip_imag[7]),
.sel_w(2'b10),
.X0_real(stage2_op_real[5])

       , .X0_imag(stage2_op_imag[5]),
           .X1_real(stage2_op_real[7]),
           .X1_imag(stage2_op_imag[7])); //
           10 is selected for sel_w as w^2
           is the twiddle factor



always@(posedge clk or negedge reset_n)

  begin

    if(!reset_n)

     begin

     stage3_ip_real[0]<=16'b0;

     stage3_ip_imag[0]<=16'b0;

     stage3_ip_real[1]<=16'b0;

     stage3_ip_imag[1]<=16'b0;

     stage3_ip_real[2]<=16'b0;

     stage3_ip_imag[2]<=16'b0;

     stage3_ip_real[3]<=16'b0;

     stage3_ip_imag[3]<=16'b0;

     stage3_ip_real[4]<=16'b0;

     stage3_ip_imag[4]<=16'b0;

     stage3_ip_real[5]<=16'b0;

     stage3_ip_imag[5]<=16'b0;

     stage3_ip_real[6]<=16'b0;

     stage3_ip_imag[6]<=16'b0;

     stage3_ip_real[7]<=16'b0;

    stage3_ip_imag[7]<=16'b0;

     end

    else

     begin

       stage3_ip_real[0]<=stage2_op_real[0];

       stage3_ip_imag[0]<=stage2_op_imag[0];

       stage3_ip_real[1]<=stage2_op_real[1];

       stage3_ip_imag[1]<=stage2_op_imag[1];

       stage3_ip_real[2]<=stage2_op_real[2];

       stage3_ip_imag[2]<=stage2_op_imag[2];

       stage3_ip_real[3]<=stage2_op_real[3];

       stage3_ip_imag[3]<=stage2_op_imag[3];

       stage3_ip_real[4]<=stage2_op_real[4];

       stage3_ip_imag[4]<=stage2_op_imag[4];

       stage3_ip_real[5]<=stage2_op_real[5];

       stage3_ip_imag[5]<=stage2_op_imag[5];

       stage3_ip_real[6]<=stage2_op_real[6];

       stage3_ip_imag[6]<=stage2_op_imag[6];

       stage3_ip_real[7]<=stage2_op_real[7];
```

```
        stage3_ip_imag[7]<=stage2_op_imag[7];

        strt_s3<=strt_s2;                          //stage 3 output 3==X2 and OUTPUT 7 ==X6


      end
   end                                            BFU bf4 (.A_real(stage3_ip_real[2]),
                                                 .A_imag(stage3_ip_imag[2]),
                                                 .B_real(stage3_ip_real[6]),
                                                 .B_imag(stage3_ip_imag[6]),
                                                 .sel_w(2'b10),.X0_real(stage3_op_real[2])
 //stage 3 computation.
                                                        , .X0_imag(stage3_op_imag[2]),
                                                            .X1_real(stage3_op_real[6]),
    //stage 3 output 1 X0                                   .X1_imag(stage3_op_imag[6])); //
                                                            10 is selected for sel_w as w^2
 carry_look_ahead_16bit cla_fft_25                          is the twiddle factor
     (.a(stage3_ip_real[0]),.b(stage3_ip_real[4]),
     .carry_in(1'b0),
     .sum(stage3_op_real[0]),.cout()); //
     real (stage 2 op1 + stage 2 op5)             //stage 3 output 4==X3 and OUTPUT 8 ==X7
                                                 BFU bf5 (.A_real(stage3_ip_real[3]),
 carry_look_ahead_16bit cla_fft_26               .A_imag(stage3_ip_imag[3]),
     (.a(stage3_ip_imag[0]),.b(stage3_ip_imag[4]),.B_real(stage3_ip_real[7]),
     .carry_in(1'b0),                            .B_imag(stage3_ip_imag[7]),
     .sum(stage3_op_imag[0]),.cout()); //         .sel_w(2'b11),.X0_real(stage3_op_real[3])
     imag (stage 2 op1 + stage 2 op5)
                                                        , .X0_imag(stage3_op_imag[3]),
                                                            .X1_real(stage3_op_real[7]),
                                                            .X1_imag(stage3_op_imag[7]));
    //stage 3 output 5 X4                                 // 11 is selected for sel_w as
                                                          w^3 is the twiddle factor
 carry_look_ahead_16bit cla_fft_27
     (.a(stage3_ip_real[0]),.b(stage3_ip_real[4]),
     .carry_in(1'b1),                             always@(posedge clk or negedge reset_n)
     .sum(stage3_op_real[4]),.cout()); //
     real (stage 2 op1 + stage 2 op5)            begin

 carry_look_ahead_16bit cla_fft_28                 if(!reset_n)
     (.a(stage3_ip_imag[0]),.b(stage3_ip_imag[4]),
     .carry_in(1'b1),                              begin
     .sum(stage3_op_imag[4]),.cout()); //
     imag (stage 2 op1 + stage 2 op5)              Out_real0<=16'b0;

                                                   Out_imag0<=16'b0;

    // stage 3 output 2 == X1 and output 6==       Out_real1<=16'b0;
        X5
                                                   Out_imag1<=16'b0;
 BFU bf3 (.A_real(stage3_ip_real[1]),
.A_imag(stage3_ip_imag[1]),                        Out_real2<=16'b0;
.B_real(stage3_ip_real[5]),
.B_imag(stage3_ip_imag[5]),                        Out_imag2<=16'b0;
.sel_w(2'b01),
.X0_real(stage3_op_real[1])                        Out_real3<=16'b0;

        , .X0_imag(stage3_op_imag[1]),             Out_imag3<=16'b0;
            .X1_real(stage3_op_real[5]),
            .X1_imag(stage3_op_imag[5])); //        Out_real4<=16'b0;
            01 is selected for sel_w as w is
            the twiddle factor                     Out_imag4<=16'b0;
```

```
        Out_real5<=16'b0;

        Out_imag5<=16'b0;

        Out_real6<=16'b0;

        Out_imag6<=16'b0;

        Out_real7<=16'b0;

       Out_imag7<=16'b0;

       end

      else

       begin

         Out_real0<=stage3_op_real[0];

         Out_imag0<=stage3_op_imag[0];

         Out_real1<=stage3_op_real[1];

         Out_imag1<=stage3_op_imag[1];

         Out_real2<=stage3_op_real[2];

         Out_imag2<=stage3_op_imag[2];

         Out_real3<=stage3_op_real[3];

         Out_imag3<=stage3_op_imag[3];

         Out_real4<=stage3_op_real[4];

         Out_imag4<=stage3_op_imag[4];

         Out_real5<=stage3_op_real[5];

         Out_imag5<=stage3_op_imag[5];

         Out_real6<=stage3_op_real[6];

         Out_imag6<=stage3_op_imag[6];

         Out_real7<=stage3_op_real[7];

         Out_imag7<=stage3_op_imag[7];

     fft_ready<=strt_s3;

       end

   end

endmodule


module FFT_wrapper(in0_real,in0_imag,in1_real,
in1_imag,in2_real,in2_imag,
in3_real,in3_imag,in4_real,
in4_imag,in5_real,in5_imag,
in6_real,in6_imag,in7_real,
in7_imag,CLK,RST_N,write,
start,out0_real,out0_imag,
out1_real,out1_imag,out2_real,
out2_imag,out3_real,out3_imag,
out4_real,out4_imag,out5_real,
out5_imag,out6_real,out6_imag,
out7_real,out7_imag,ready);

  input signed
      [15:0]in0_real,in1_real,in2_real,in3_real,
in4_real,in5_real,in6_real,
in7_real,in0_imag,
in1_imag,in2_imag,
in3_imag,in4_imag,
in5_imag,in6_imag,
in7_imag;

  input CLK,RST_N,write,start;

  output ready;

  output wire signed
      [15:0]out0_real,out1_real,
out2_real,out3_real,out4_real,
out5_real,out6_real,out7_real,
out0_imag,out1_imag,
out2_imag,out3_imag,
out4_imag,out5_imag,
out6_imag,out7_imag;

  FFT fft(.In_real0(in0_real),
.In_real1(in1_real),.In_real2(in2_real),
.In_real3(in3_real),.In_real4(in4_real),
.In_real5(in5_real),.In_real6(in6_real),
.In_real7(in7_real),.In_imag0(in0_imag),
.In_imag1(in1_imag),
.In_imag2(in2_imag),.In_imag3(in3_imag),
.In_imag4(in4_imag),.In_imag5(in5_imag),
.In_imag6(in6_imag),.In_imag7(in7_imag),
.reset_n(RST_N),.clk(CLK),.write(write),
.start_fft(start),.Out_real0(out0_real),
.Out_real1(out1_real),.Out_real2(out2_real),
.Out_real3(out3_real),.Out_real4(out4_real),
.Out_real5(out5_real),.Out_real6(out6_real),
.Out_real7(out7_real),.Out_imag0(out0_imag),
.Out_imag1(out1_imag),.Out_imag2(out2_imag),
.Out_imag3(out3_imag),.Out_imag4(out4_imag),
.Out_imag5(out5_imag),.Out_imag6(out6_imag),
.Out_imag7(out7_imag),.fft_ready(ready));


endmodule
```

The Testbench file used for the project is given below:

```
`timescale 1ns/1ps
module FFT_tb;
  parameter CLOCK_PERIOD = 10; // 10 MHz clock
  reg [15:0]
```

```
     in0_real,in1_real,in2_real,in3_real,in4_real,
 in5_real,in6_real,in7_real,in0_imag,in1_imag,in2_imag,
 in3_imag,in4_imag,in5_imag,in6_imag,in7_imag;
  reg CLK,RST_N,write,start;
  wire [15:0]
     out0_real,out1_real,out2_real,out3_real,out4_real,
out5_real,out6_real,out7_real,out0_imag,out1_imag,
 out2_imag,out3_imag,out4_imag,out5_imag,out6_imag,out7_imag;
wire ready;
wire [7:0] count_val;

  FFT_wrapper
     ff(.in0_real(in0_real),.in1_real(in1_real),
.in2_real(in2_real),.in3_real(in3_real),.in4_real(in4_real),
.in5_real(in5_real),.in6_real(in6_real),.in7_real(in7_real),
.in0_imag(in0_imag),.in1_imag(in1_imag),.in2_imag(in2_imag),
.in3_imag(in3_imag),.in4_imag(in4_imag),.in5_imag(in5_imag),
.in6_imag(in6_imag),.in7_imag(in7_imag),.RST_N(RST_N),
.CLK(CLK),.write(write),.start(start),.out0_real(out0_real),
.out1_real(out1_real),.out2_real(out2_real),.out3_real(out3_real),
.out4_real(out4_real),.out5_real(out5_real),.out6_real(out6_real),
.out7_real(out7_real),.out0_imag(out0_imag),.out1_imag(out1_imag),
.out2_imag(out2_imag),.out3_imag(out3_imag),.out4_imag(out4_imag),
.out5_imag(out5_imag),.out6_imag(out6_imag),.out7_imag(out7_imag),
.ready(ready));
initial begin
$dumpfile("FFT_tb.vcd");
$dumpvars(0,ff);
$monitor("Time: %t :: Count: %x", $time,
    count_val);
end

initial begin
  CLK=0;

  write=0;
  start=0;
  RST_N=0;
  #12 RST_N=1;
    in0_real=100;
   in1_real=200;
  in2_real=300;
  in3_real=400;
  in4_real=500;
  in5_real=600;
  in6_real=700;
  in7_real=800;

  in0_imag=200;
  in1_imag=300;
  in2_imag=400;
  in3_imag=500;
  in4_imag=600;
  in5_imag=700;
  in6_imag=800;
  in7_imag=900;
  write =1;
  #4 write=0;
  #6 start=1;
       in0_real=0;
                              in1_real=100;
                  in2_real=200;
  in3_real=300;
  in4_real=400;
  in5_real=500;
  in6_real=600;
  in7_real=700;

  in0_imag=0;
  in1_imag=0;
  in2_imag=0;
  in3_imag=0;
  in4_imag=0;
  in5_imag=0;
  in6_imag=0;
  in7_imag=0;

    write=1;
  #5 start =0;
  write=0;

  #6 start=1;
  #5 start=0;

  #80 $finish();
 // #10 a=16'd0; b=16'd0; carry_in=1'd1;
// #10 a=-16'd15; b=16'd1; carry_in=1'd1;
 // #10 a=16'd5; b=16'd23; carry_in=1'd0;
 // #10 a=16'd999; b=16'd7; carry_in=1'd1;
end
  always begin
    #(CLOCK_PERIOD/2) CLK = ~CLK;
   end
  always@(*)
    begin
      if(ready)
        begin
      $display("out0_real =%d,out0_imag= %d
          ,out1_real=
          %d,out1_imag=%d,out2_real
          =%d,out2_imag= %d ,out3_real=
          %d,out3_imag=%d, out4_real
          =%d,out4_imag= %d ,out5_real=
          %d,out5_imag=%d,out6_real
          =%d,out6_imag= %d ,out7_real=
          %d,out7_imag=%d,time =%0d",
          $signed(out0_real),$signed(out0_imag),$signed(
    end
    end

endmodule
```

The Matlab code used for running FFT is given below

```
% Define the input sequence x[n] of length N
    = 8
x = [1+2i, 2+3i, 3+4i, 4+5i, 5+6i, 6+7i,
    7+8i, 8+9i];

% Calculate the FFT using the built-in fft
    function
X = fft(x);
```

```
disp('FFT using built-in fft function:');
disp(X);
```

The code for the synthesis file synth.ys is given below

```
# read design
read_verilog FFT.v
hierarchy -check
hierarchy -top FFT_wrapper
#flatten
# high level synthesis
proc; opt; clean
fsm; opt; clean
memory; opt; clean
# low level synthesis
techmap; opt; clean
# map to target architecture
dfflibmap -liberty
    NangateOpenCellLibrary_typical.lib
abc -liberty
    NangateOpenCellLibrary_typical.lib
# split larger signals
splitnets -ports; opt; clean
# write synthesis output
write_verilog synth.v
write_spice synth.sp
# print synthesis reports
stat
stat -liberty
    NangateOpenCellLibrary_typical.lib
```

The project files are available at github, by clicking on the following link https://github.com/KevinMathewEC/FFT_ESDCS_Project/tree/main