# EE5516: VLSI Architectures for Signal Processing and Machine Learning



## Mini Project

# Design and implementation of IFFT Algorithm (DIT, DIF)

R.Sai Rohan 122101030

Kasi Viswanath 122101020

## 1 Abstract

In this project, an 8-point radix 2 decimation in time (DIT) and decimation in frequency (DIF) method will be used to create the structure for an inverse fast fourier transform (IFFT). The design is pipelined to shorten the crucial period, further increasing the IFFT's efficiency. In conclusion, Verilog, a hardware description language used for digital circuit design, is utilized to create both the pipelined and IFFT versions.

#### 2 Introduction

The Inverse Fast Fourier Transform (IFFT) is a fundamental algorithm used to convert frequency-domain representations of signals back to their original time-domain form. It serves as the inverse operation of the Fast Fourier Transform (FFT), which transforms time-domain signals into their frequency-domain representations.

The IFFT plays a crucial role in various fields such as digital signal processing, telecommunications, audio processing, image processing, and many others. It enables the reconstruction of signals after processing in the frequency domain, facilitating tasks such as filtering, modulation, demodulation, equalization, and spectral analysis.

## 3 Theory

## 3.1 Discrete Fourier Transform (DFT)

Given a discrete-time signal x[n] of length N, the DFT X[k] is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi nk/N}$$

where:

• x[n] is the input signal,

- X[k] is the frequency domain representation of the signal,
- $\bullet$  e is the base of the natural logarithm,
- j is the imaginary unit,
- n is the index of the time domain signal,
- k is the index of the frequency domain signal,
- $\bullet$  N is the total number of samples in the input signal.

The DFT maps a discrete-time signal from the time domain to the frequency domain. Each bin X[k] represents the amplitude and phase of a specific frequency component in the signal.

## 3.2 Inverse Discrete Fourier Transform (IDFT)

The IDFT is the reverse process of the DFT, used to reconstruct the original time-domain signal from its frequency domain representation. It's defined as:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{j2\pi nk/N}$$

where:

- $\bullet$  x[n] is the reconstructed time-domain signal,
- $\bullet$  X[k] is the frequency domain signal (obtained from the DFT),
- $\bullet$  N is the total number of samples in the signal.

## 3.3 Fast Fourier Transform (FFT)

#### 3.3.1 Algorithm:

The FFT algorithm, particularly the radix-2 FFT, recursively divides the DFT into smaller DFTs and combines them using a technique called butterfly operations. These operations exploit the symmetry of the complex exponentials in the Fourier transform to reduce the number of computations.

## Steps:

- 1. **Decomposition:** Divide the input sequence into smaller subsequences.
- 2. Computation: Compute the DFT of each sub-sequence.
- 3. **Combination:** Combine the results using twiddle factors and butterfly operations.
- 4. **Recombination:** Repeat the process recursively until the entire DFT is computed.

## 3.4 Inverse Fast Fourier Transform (IFFT)

#### 3.4.1 Algorithm:

The IFFT algorithm is essentially the FFT algorithm run in reverse. It applies the FFT algorithm to the conjugate of the input sequence, and then the resulting sequence is conjugated again. This process effectively reverses the transformation done by the FFT.

## 3.5 Algorithm

#### Steps to Compute IFFT Using Radix-2 DIT Algorithm

- 1. **Input**: Start with a frequency-domain sequence  $X = [X_0, X_1, \dots, X_{N-1}]$ .
- 2. **Bit-Reversal Permutation**: Rearrange elements of X for radix-2 DIT FFT.
- 3. Compute Twiddle Factors: Calculate  $W_N^{nk} = e^{j2\pi nk/N}$  for each stage.
- 4. Radix-2 DIT FFT in Reverse Order: Apply the FFT algorithm from final stage to first stage using twiddle factors.
- 5. **Output**: Obtain the time-domain sequence  $x = [x_0, x_1, \dots, x_{N-1}]$ , representing the inverse FFT of X.

#### 3.6 Architecture

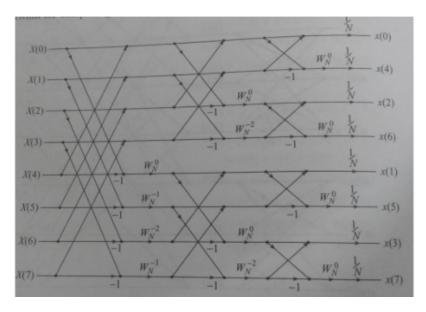


Figure 1: Architecture of DIT IFFT

#### 3.7 Code

```
module IFFT(x0,x1,x2,x3,x4,x5,x6,x7,reset,y0,y1,y2,y3,y4,y5,y6
      ,y7,input clk
             );
    parameter integer width = 16; // total width of fixed-point
        representation
    parameter integer I_width = 1; // integer part width of
6
        fixed-point representation
    parameter integer F_width = 15; // fractional part width of
       fixed-point representation
    parameter QP = 15;
8
9
10
11
    typedef struct {
      logic signed [width-1:0] re;
13
      logic signed [width-1:0] im;
14
    } complex_grow;
15
16
17
18
    parameter complex_grow w0new='{re: 16'h7fff, im: 16'h0000};
19
    parameter complex_grow w1new='{re: 16'h5a82, im: 16'ha57e};
20
    parameter complex_grow w2new='{re: 16'h0000, im: 16'h8000};
21
    parameter complex_grow w3new='{re: 16'ha57e, im: 16'ha57e};
    typedef struct {
25
26
      logic signed [31:0] re;
27
      logic signed [2*(width)-1:0] im;
28
    } complex_temp_grow;
30
31
32
    function complex_grow complex_mult_grow(complex_grow a,
33
       complex_grow b);
       complex_temp_grow full;
36
      complex_grow full1;
37
      full.re = ($signed(a.re) * $signed(b.re)) - ($signed(a.im)
38
           * $signed(b.im));
```

```
full.im = ($signed(a.re) * $signed(b.im)) + ($signed(a.im)
39
           * $signed(b.re));
40
       full1.re=full.re[30:15];
       full1.im=full.im[30:15];
42
43
       return full1;
44
     endfunction
47
    function complex_grow complex_add_grow(complex_grow a,
48
        complex_grow b);
49
       complex_grow result;
       result.re = a.re + b.re;
       result.im = a.im + b.im;
       return result;
52
     endfunction
53
54
     input mode;
     input complex_grow x0,x1,x2,x3,x4,x5,x6,x7;
     input reset;
     output complex_grow y0, y1, y2, y3, y4, y5, y6, y7;
58
    reg[width-1:0] a00,a01,a10,a11,a20,a21,a30,a31,a40,a41,a50,
59
        a51,a60,a61,a70,a71;
    wire [width-1:0] a0new_re,a1new_re,a2new_re,a3new_re,
        a4new_re, a5new_re, a6new_re, a7new_re;
    wire [width-1:0] a0new_im, a1new_im, a2new_im, a3new_im,
61
        a4new_im, a5new_im, a6new_im, a7new_im;
    wire complex_grow y0a,y1a,y2a,y3a,y4a,y5a,y6a,y7a;
     wire complex_grow y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7;
63
    wire complex_grow
                        b0, b1, b2, b3, b4, b5, b6, b7, b8;
    wire complex_grow c0,c1,c2,c3,c4,c5,c6,c7;
    wire complex_grow
                          d0, d1, d2, d3, d4, d5, d6, d7;
67
    wire complex_grow
                         e0,e1,e2,e3,e4,e5,e6,e7;
68
                         f0,f1,f2,f3,f4,f5,f6,f7;
    wire complex_grow
    reg [width-1:0] b0new_re,b1new_re,b2new_re,b3new_re,b4new_re
        ,b5new_re,b6new_re,b7new_re;
    reg [width-1:0] b0new_im,b1new_im,b2new_im,b3new_im,b4new_im
71
        , b5new_im, b6new_im, b7new_im;
    reg [width-1:0] dOnew_re,dlnew_re,d2new_re,d3new_re,d4new_re
72
        , d5new_re, d6new_re, d7new_re;
    reg [width-1:0] d0new_im,d1new_im,d2new_im,d3new_im,d4new_im
        , d5new_im, d6new_im, d7new_im;
74
75
76
```

```
always@(posedge clk)
77
          begin
            if(reset)
79
               begin
80
81
                  a70 <= 0;
82
                  a60<=0;
83
                  a50<=0;
                  a40 <= 0;
85
                  a30 <= 0;
86
                  a20 <= 0;
87
                  a10 <= 0;
88
                  a00<=0;
89
                  a71 <= 0;
                  a61<=0;
91
                  a51 <= 0;
92
                  a41 <= 0;
93
94
                  a31 <= 0;
                  a21 <= 0;
                  a11<=0;
                  a01<=0;
97
98
               end
99
            else
100
101
102
                     begin
103
104
                        a00 \le x0.re;
105
                        a10 \le x4.re;
106
                        a20 <= x2.re;
107
                        a30 \le x6.re;
108
                        a40 \le x1.re;
109
                        a50 <= x5.re;
110
                        a60 \le x3.re;
111
                        a70 \le x7.re;
112
                        a01 \le x0.im;
113
                        a11 \le x4.im;
114
                        a21 <= x2.im;
115
                        a31 <= x6.im;
116
                        a41 <= x1.im;
117
                        a51 <= x5.im;
118
                        a61 \le x3.im;
119
                        a71 <= x7.im;
120
                     end
121
122
123
          end
```

```
124
     assign a0new_re = {{3{a00[width-1]}}}, a00[15:3]};
125
     assign a1new_re = {{3{a10[width-1]}}}, a10[15:3]};
126
     assign a2new_re = \{\{3\{a20[width-1]\}\}, a20[15:3]\};
127
     assign \ a3new_re = \{\{3\{a30[width-1]\}\}, \ a30[15:3]\};
128
     assign a4new_re = \{\{3\{a40[width-1]\}\}, a40[15:3]\};
129
     assign a5new_re = \{\{3\{a50[width-1]\}\}, a50[15:3]\};
130
     assign a6new_re = {{3{a60[width-1]}}}, a60[15:3]};
     assign a7new_re = \{\{3\{a70[width-1]\}\}, a70[15:3]\};
     assign a0new_im = {{3{a01[width-1]}}}, a01[15:3]};
134
     assign alnew_im = {{3{a11[width-1]}}}, a11[15:3]};
135
     assign \ a2new_im = \{\{3\{a21[width-1]\}\}, \ a21[15:3]\};
136
     assign a3new_im = \{\{3\{a31[width-1]\}\}, a31[15:3]\};
     assign \ a4new_im = \{\{3\{a41[width-1]\}\}, \ a41[15:3]\};
138
     assign \ a5new_im = \{\{3\{a51[width-1]\}\}, \ a51[15:3]\};
139
     assign \ a6new_im = \{\{3\{a61[width-1]\}\}, \ a61[15:3]\};
140
     assign a7new_im = \{\{3\{a71[width-1]\}\}, a71[15:3]\};
141
142
144
145
146
     assign e0 = complex_add_grow('{re: d0new_re, im: d0new_im},
147
         '{re: d1new_re, im: d1new_im});
     assign e1 = complex_add_grow('{re: d0new_re, im: d0new_im},
         '{re: -d1new_re, im: -d1new_im});
     assign e2 = complex_add_grow('{re: d2new_re, im: d2new_im},
149
         '{re: d3new_re, im: d3new_im});
     assign e3 = complex_add_grow('{re: d2new_re, im: d2new_im},
150
         '{re: -d3new_re, im: -d3new_im});
     assign e4 = complex_add_grow('{re: d4new_re, im: d4new_im},
         '{re: d5new_re, im: d5new_im});
     assign e5 = complex_add_grow('{re: d4new_re, im: d4new_im},
         '{re: -d5new_re, im: -d5new_im});
     assign e6 = complex_add_grow('{re: d6new_re, im: d6new_im},
153
         '{re: d7new_re, im: d7new_im});
     assign e7 = complex_add_grow('{re: d6new_re, im: d6new_im},
         '{re: -d7new_re, im: -d7new_im});
     always@(posedge clk)
156
       begin
157
          b0new_re <= {{4{b0.re[width-1]}}, b0.re};
          b1new_re <= {{4{b1.re[width-1]}}}, b1.re};
159
          b2new_re <= {{4{b2.re[width-1]}}}, b2.re};
          b3new_re <= {{4{b3.re[width-1]}}, b3.re};
161
          b4new_re <= {{4{b4.re[width-1]}}, b4.re};
162
```

```
b5new_re <= {{4{b5.re[width-1]}}, b5.re};
          b6new_re <= {{4{b6.re[width-1]}}}, b6.re};
164
          b7new_re <= {{4{b7.re[width-1]}}, b7.re};
          b0new_im <= \{\{4\{b0.im[width-1]\}\}, b0.im\};
167
          b1new_im <= {{4{b1.im[width-1]}}, b1.im};
168
          b2new_im <= {{4{b2.im[width-1]}}, b2.im};
169
          b3new_im <= \{\{4\{b3.im[width-1]\}\}, b3.im\};
170
          b4new_im <= \{\{4\{b4.im[width-1]\}\}, b4.im\};
171
          b5new_im \le {\{4\{b5.im[width-1]\}\}, b5.im\}};
172
          b6new_im <= \{\{4\{b6.im[width-1]\}\}, b6.im\};
173
          b7new_im <= {{4{b7.im[width-1]}}, b7.im};
174
175
          d0new_re <= d0.re;
          d1new_re <= d1.re;</pre>
177
          d2new_re <=d2.re;
178
          d3new_re <= d3.re;
179
          d4new_re <= d4.re;
180
          d5new_re <= d5.re;
181
          d6new_re <= d6.re;
          d7new_re <= d7.re;
183
184
          dOnew_im <=d0.im;
185
          d1new_im <=d1.im;
186
          d2new_im <=d2.im;
187
          d3new_im <=d3.im;
          d4new_im <=d4.im;
189
          d5new_im <=d5.im;
190
          d6new_im <=d6.im;
          d7new_im <= d7.im;
192
194
        end
195
196
197
198
      assign c0='{re: b0new_re, im: b0new_im};
200
      assign c1='{re: b1new_re, im: b1new_im};
201
      assign c2=complex_mult_grow('{re: b2new_re, im: b2new_im},'{
202
         re: w0new.re, im: w0new.im});
203
204
205
206
207
```

```
assign c3=complex_mult_grow('{re: b3new_re, im: b3new_im},'{
208
        re: w2new.re, im: w2new.im});
     assign c4='{re: b4new_re, im: b4new_im};
209
     assign c5='{re: b5new_re, im: b5new_im};
     assign c6=complex_mult_grow('{re: b6new_re, im: b6new_im},'{
211
        re: w0new.re, im: w0new.im});
     assign c7=complex_mult_grow('{re: b7new_re, im: b7new_im},'{
212
        re: w2new.re, im: w2new.im});
213
214
     assign d0=complex_add_grow(c2,c0);
215
     assign d1=complex_add_grow(c3,c1);
216
     assign d2=complex_add_grow(c0, '{re: -c2.re, im: -c2.im});
217
     assign d3=complex_add_grow(c1, '{re: -c3.re, im: -c3.im});
     assign d4=complex_add_grow(c6,c4);
219
     assign d5=complex_add_grow(c7,c5);
220
     assign d6=complex_add_grow(c4,'{re: -c6.re, im: -c6.im});
221
     assign d7=complex_add_grow(c5,'{re: -c7.re, im: -c7.im});
222
223
     assign b0='{re: a0new_re, im: a0new_im};
225
     assign b1='{re: a1new_re, im: a1new_im};
226
     assign b2='{re: a2new_re, im: a2new_im};
227
     assign b3='{re: a3new_re, im: a3new_im};
228
     assign b4=complex_mult_grow('{re: a4new_re, im: a4new_im},'{
        re: w0new.re, im: -w0new.im});
     assign b5=complex_mult_grow('{re: a5new_re, im: a5new_im},'{
230
        re: w1new.re, im: -w1new.im});
     assign b6=complex_mult_grow('{re: a6new_re, im: a6new_im},'{
231
        re: w2new.re, im: -w2new.im});
     assign b7=complex_mult_grow('{re: a7new_re, im: a7new_im},'{
        re: w3new.re, im: -w3new.im});
234
     assign f0=complex_add_grow(e4,e0);
235
     assign f1=complex_add_grow(e5,e1);
236
     assign f2=complex_add_grow(e6,e2);
     assign f3=complex_add_grow(e7,e3);
     assign f4=complex_add_grow(e0, '{re: -e4.re, im: -e4.im});
239
     assign f5=complex_add_grow(e1,'{re: -e5.re, im: -e5.im});
240
     assign f6=complex_add_grow(e2,'{re: -e6.re, im: -e6.im});
     assign f7=complex_add_grow(e3,'{re: -e7.re, im: -e7.im});
242
     assign y0='{re: -f0.re/8, im: f0.im/8};
244
     assign y1='{re: -f1.re/8, im: f1.im/8};
245
     assign y2='{re: -f2.re/8, im: f2.im/8};
246
     assign y3='{re: -f3.re/8, im: f3.im/8};
247
```

```
assign y4='{re: -f4.re/8, im: f4.im/8};
assign y5='{re: -f5.re/8, im: f5.im/8};
assign y6='{re: -f6.re/8, im: f6.im/8};
assign y7='{re: -f7.re/8, im: f7.im/8};

251
assign y7='{re: -f7.re/8, im: f7.im/8};

252
assign y7='{re: -f7.re/8, im: f7.im/8};

253
assign y7='{re: -f7.re/8, im: f7.im/8};
```

#### 3.8 Test Bench

#### Verilog Testbench Code

```
module testbench_FFT;
1
2
     parameter integer width = 16;
3
     reg clk;
     reg reset;
6
     integer write_data;
     localparam SF = 2.0**(-12);
     reg signed [width-1:0] inputs[15:0];
     reg signed [width-1:0] realv[7:0];
     reg signed [width-1:0] imag[7:0];
11
     typedef struct {
13
       logic signed [width-1:0] re;
14
       logic signed [width-1:0] im;
15
     } complex_grow;
16
17
     complex_grow x0, x1, x2, x3, x4, x5, x6, x7;
18
19
     complex_grow y0, y1, y2, y3, y4, y5, y6, y7;
20
     // Instantiate the FFT module
21
     IFFT FFT_instance (
22
       .x0(x0),
       .x1(x1),
       .x2(x2),
25
       .x3(x3),
26
       .x4(x4),
27
       .x5(x5),
28
       .x6(x6),
29
       .x7(x7),
30
       .clk(clk),
31
       .reset(reset),
32
       .y0(y0),
33
34
       .y1(y1),
       .y2(y2),
       .y3(y3),
36
```

```
.y4(y4),
37
       .y5(y5),
38
       .y6(y6),
39
       .y7(y7)
40
41
42
     // Generate clock signal
43
     always #5 clk = ~clk;
45
     initial begin
46
       $dumpfile("dump.vcd");
47
       $dumpvars(0);
49
50
     task open();
51
       $readmemh("input.txt", inputs);
53
       for (integer i = 0; i < 16; i = i + 2) begin
54
         realv[i/2] = inputs[i];
       end
56
       for (integer i = 1; i < 16; i = i + 2) begin
57
         imag[i/2] = inputs[i];
58
       end
59
60
       // Assign input values to inputs of FFT module
61
       x0 = '{re: realv[0], im: imag[0]};
62
       x1 = '{re: realv[1], im: imag[1]};
63
       x2 = '{re: realv[2], im: imag[2]};
64
       x3 = '{re: realv[3], im: imag[3]};
65
       x4 = '{re: realv[4], im: imag[4]};
66
       x5 = '\{re: realv[5], im: imag[5]\};
67
       x6 = '{re: realv[6], im: imag[6]};
68
       x7 = '{re: realv[7], im: imag[7]};
69
     endtask
70
     task drive_reset();
72
       $display("Driving the reset");
73
       clk = 1'b0;
74
       @(negedge clk) reset = 0;
       @(negedge clk) reset = 1;
76
       @(negedge clk) reset = 0;
     endtask
79
     task display_output();
80
       $display("Output values :- \n");
81
       for (integer i = 0; i < 8; i++) begin
82
         display("y\%0d = \%f + \%f j", i, $itor(y[i].re) * SF, $itor(y[i])
83
             ].im) * SF);
       end
84
       $fwrite(write_data, "Output values :- \n");
86
       for (integer i = 0; i < 8; i++) begin
87
         fwrite(write_data, "y\%0d = \%f + \%f j\n", i, $itor(y[i].re) *
88
             SF, $itor(y[i].im) * SF);
```

```
end
89
     endtask
90
91
     initial begin
       write_data = $fopen("output_tracker.txt", "w");
93
       drive_reset();
94
       open();
95
       repeat (30) @(negedge clk);
       display_output();
97
        $fclose(write_data);
       $finish;
     \verb"end"
   endmodule
```

Listing 1: IFFT Testbench

## 3.9 Results and graphs

```
# KERNEL: y0= 1.779785 + -0.219482 j

# KERNEL: y1= 0.219238 + -0.280518 j

# KERNEL: y2= 0.573242 + 0.572998 j

# KERNEL: y3= 0.426758 + -0.072998 j

# KERNEL: y4= -0.280273 + -0.780518 j

# KERNEL: y5= 0.280273 + 1.280518 j

# KERNEL: y6= -0.072754 + 0.427002 j

# KERNEL: y7= -0.926270 + -0.927002 j
```

Figure 2: Output of the DIT-FFT



Figure 3: Waveform diagram of the DIT-FFT

## 4 IFFT - DIF

## 4.1 Algorithm

Input: Frequency-domain sequence  $X = [X_0, X_1, ..., X_{N-1}]$ . Twiddle Factor Computation:

• Compute twiddle factors  $W_N^{nk} = e^{-j\frac{2\pi nk}{N}}$  for each stage.

#### Radix-2 DIF IFFT:

- Apply radix-2 DIF FFT algorithm.
- Perform butterfly operations in reverse order.

#### **Bit-Reversal Permutation**:

• Perform bit-reversal permutation on the output sequence.

#### Normalization:

 $\bullet$  Normalize the output by dividing by N

## 4.2 Architecture

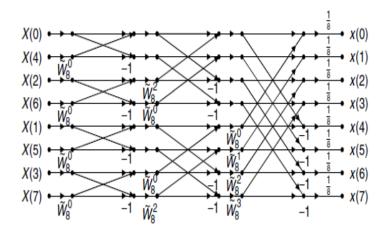


Figure 4: Output of the DIF-IFFT

#### 4.3 Code

```
module FFT(x0,x1,x2,x3,x4,x5,x6,x7,reset,y0,y1,y2,y3,y4,y5,y6,y7,
     input clk
            );
    parameter integer width = 16; // total width of fixed-point
     representation
    parameter integer I_width = 1; // integer part width of fixed-
     point representation
    parameter integer F_width = 15; // fractional part width of fixed-
     point representation
    parameter QP = 15;
10
    typedef struct {
12
13
      logic signed [width-1:0] re;
      logic signed [width-1:0] im;
14
    } complex_grow;
15
16
17
18
    parameter complex_grow w0new='{re: 16'h7fff, im: 16'h0000};
19
    parameter complex_grow w1new='{re: 16'h5a82, im: 16'ha57e};
20
    parameter complex_grow w2new='{re: 16'h0000, im: 16'h8000};
21
    parameter complex_grow w3new='{re: 16'ha57e, im: 16'ha57e};
22
23
24
    typedef struct {
25
26
      logic signed [31:0] re;
27
      logic signed [2*(width)-1:0] im;
28
29
    } complex_temp_grow;
30
32
    function complex_grow complex_mult_grow(complex_grow a,
33
     complex_grow b);
34
35
      complex_temp_grow full;
36
      complex_grow full1;
37
      full.re = ($signed(a.re) * $signed(b.re)) - ($signed(a.im) *
38
     $signed(b.im));
      full.im = ($signed(a.re) * $signed(b.im)) + ($signed(a.im) *
39
     $signed(b.re));
40
      full1.re=full.re[30:15];
41
42
      full1.im=full.im[30:15];
43
      return full1;
44
    endfunction
```

```
46
47
    function complex_grow complex_add_grow(complex_grow a,
48
     complex_grow b);
      complex_grow result;
49
      result.re = a.re + b.re;
50
      result.im = a.im + b.im;
      return result;
    endfunction
53
54
55
56
    input complex_grow x0,x1,x2,x3,x4,x5,x6,x7;
    input reset;
57
    output complex_grow y0, y1, y2, y3, y4, y5, y6, y7;
58
    reg[width-1:0] a00,a01,a10,a11,a20,a21,a30,a31,a40,a41,a50,a51,a60
59
     ,a61,a70,a71;
    wire [width-1:0] a0new_re,a1new_re,a2new_re,a3new_re,a4new_re,
60
     a5new_re, a6new_re, a7new_re;
    wire [width-1:0] a0new_im,a1new_im,a2new_im,a3new_im,a4new_im,
     a5new_im, a6new_im, a7new_im;
    wire complex_grow y0a,y1a,y2a,y3a,y4a,y5a,y6a,y7a;
63
    wire complex_grow y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7;
64
    wire complex_grow
                        b0,b1,b2,b3,b4,b5,b6,b7,b8;
65
    wire complex_grow c0,c1,c2,c3,c4,c5,c6,c7;
66
    wire complex_grow
                         d0,d1,d2,d3,d4,d5,d6,d7;
67
                        e0,e1,e2,e3,e4,e5,e6,e7;
    wire complex_grow
    wire complex_grow
                        f0,f1,f2,f3,f4,f5,f6,f7;
69
    reg [width-1:0] b0new_re,b1new_re,b2new_re,b3new_re,b4new_re,
     b5new_re, b6new_re, b7new_re;
    reg [width-1:0] b0new_im,b1new_im,b2new_im,b3new_im,b4new_im,
     b5new_im, b6new_im, b7new_im;
    reg [width-1:0] d0new_re,d1new_re,d2new_re,d3new_re,d4new_re,
     d5new_re,d6new_re,d7new_re;
    reg [width-1:0] d0new_im,d1new_im,d2new_im,d3new_im,d4new_im,
     d5new_im,d6new_im,d7new_im;
74
75
76
    always@(posedge clk)
77
      begin
78
        if(reset)
79
          begin
80
81
             a70 <= 0;
82
             a60 <= 0;
             a50 <= 0;
84
             a40 <= 0;
85
             a30 <= 0;
86
             a20 <= 0;
87
             a10 <= 0;
88
             a00 <= 0;
89
             a71 <= 0;
90
             a61 <= 0;
```

```
a51<=0;
92
               a41 <= 0;
93
              a31 <= 0;
94
95
              a21 <= 0;
               a11 <= 0;
96
              a01 <=0;
97
98
            end
99
          else
100
                 begin
102
                   a00 \le x0.re;
104
                   a10 \le x4.re;
                   a20 \le x2.re;
106
                   a30 \le x6.re;
107
                   a40 \le x1.re;
108
                   a50 <= x5.re;
                   a60 \le x3.re;
110
                   a70 \le x7.re;
111
                   a01 \le x0.im;
112
                   a11 \le x4.im;
113
114
                   a21 \le x2.im;
                   a31 <= x6.im;
115
                   a41 <= x1.im;
116
                   a51 <= x5.im;
                   a61 <= x3.im;
118
119
                   a71 <= x7.im;
                 end
120
123
       end
124
     assign a0new_re = {{3{a00[width-1]}}}, a00[15:3]};
125
     assign alnew_re = {{3{a10[width-1]}}}, a10[15:3]};
126
     assign a2new_re = {{3{a20[width-1]}}}, a20[15:3]};
127
     assign a3new_re = {{3{a30[width-1]}}}, a30[15:3]};
128
     assign a4new_re = {{3{a40[width-1]}}}, a40[15:3]};
129
     assign a5new_re = {{3{a50[width-1]}}}, a50[15:3]};
     assign a6new_re = {{3{a60[width-1]}}}, a60[15:3]};
     assign a7new_re = {{3{a70[width-1]}}}, a70[15:3]};
133
     assign a0new_im = {{3{a01[width-1]}}}, a01[15:3]};
134
     assign alnew_im = \{\{3\{a11[width-1]\}\}, a11[15:3]\};
     assign a2new_im = {{3{a21[width-1]}}}, a21[15:3]};
136
     assign a3new_im = {{3{a31[width-1]}}}, a31[15:3]};
137
     assign \ a4new_im = \{\{3\{a41[width-1]\}\}, \ a41[15:3]\};
138
     assign a5new_im = {{3{a51[width-1]}}}, a51[15:3]};
139
     assign a6new_im = {{3{a61[width-1]}}}, a61[15:3]};
140
     assign a7new_im = {{3{a71[width-1]}}}, a71[15:3]};
141
142
143
144
```

```
146
147
     assign b0=complex_add_grow('{re: a0new_re, im: a0new_im},'{re:
      alnew_re, im: alnew_im});
     assign b1=complex_add_grow('{re: a0new_re, im: a0new_im},'{re: -
      alnew_re, im: -alnew_im});
     assign b2=complex_add_grow('{re: a2new_re, im: a2new_im},'{re:
149
      a3new_re, im: a3new_im});
     assign b3=complex_add_grow('{re: a2new_re, im: a2new_im},'{re: -
      a3new_re, im: -a3new_im});
     assign b4=complex_add_grow('{re: a4new_re, im: a4new_im},'{re:
      a5new_re, im: a5new_im});
     assign b5=complex_add_grow('{re: a4new_re, im: a4new_im},'{re: -
      a5new_re, im: -a5new_im});
     assign b6=complex_add_grow('{re: a6new_re, im: a6new_im},'{re:
      a7new_re, im: a7new_im});
     assign b7=complex_add_grow('{re: a6new_re, im: a6new_im},'{re: -
      a7new_re, im: -a1new_im});
     always@(posedge clk)
         b0new_re <= {{4{b0.re[width-1]}}}, b0.re};
158
         b1new_re <= {{4{b1.re[width-1]}}, b1.re};
         b2new_re <= {{4{b2.re[width-1]}}}, b2.re};
160
         b3new_re <= {{4{b3.re[width-1]}}}, b3.re};
161
         b4new_re <= {{4{b4.re[width-1]}}}, b4.re};
162
         b5new_re <= {{4{b5.re[width-1]}}, b5.re};
163
         b6new_re <= {{4{b6.re[width-1]}}}, b6.re};
164
165
         b7new_re <= {{4{b7.re[width-1]}}, b7.re};
166
         b0new_im <= \{\{4\{b0.im[width-1]\}\}, b0.im\};
167
         b1new_im <= \{\{4\{b1.im[width-1]\}\}, b1.im\};
168
         b2new_im <= {{4{b2.im[width-1]}}, b2.im};
169
         b3new_im <= \{\{4\{b3.im[width-1]\}\}, b3.im\};
170
         b4new_im <= {{4{b4.im[width-1]}}}, b4.im};
171
         b5new_im \le {\{4\{b5.im[width-1]\}\}, b5.im\}};
172
         b6new_im <= {{4{b6.im[width-1]}}, b6.im};
173
         b7new_im <= {{4{b7.im[width-1]}}, b7.im};
174
175
         d0new_re <= d0.re;
176
         d1new_re <= d1.re;
177
         d2new_re <=d2.re;
178
         d3new_re <= d3.re;
179
         d4new_re <= d4.re;
180
         d5new_re <= d5.re;
181
         d6new_re <= d6.re;
182
         d7new_re <= d7.re;
183
184
         dOnew_im <=d0.im;
185
         d1new_im <=d1.im;</pre>
186
         d2new_im <=d2.im;
187
         d3new_im <=d3.im;
188
         d4new_im <=d4.im;
189
         d5new_im <=d5.im;
190
         d6new_im <=d6.im;
```

```
d7new_im <=d7.im;
192
194
       end
195
196
197
198
     assign c0='{re: b0new_re, im: b0new_im};
199
200
     assign c1='{re: b1new_re, im: b1new_im};
201
     assign c2=complex_mult_grow('{re: b2new_re, im: b2new_im},'{re:
202
      w0new.re, im: w0new.im});
203
204
205
206
207
     assign c3=complex_mult_grow('{re: b3new_re, im: b3new_im},'{re:
208
      w2new.re, im: w2new.im});
     assign c4='{re: b4new_re, im: b4new_im};
209
     assign c5='{re: b5new_re, im: b5new_im};
210
     assign c6=complex_mult_grow('{re: b6new_re, im: b6new_im},'{re:
211
      w0new.re, im: w0new.im});
     assign c7=complex_mult_grow('{re: b7new_re, im: b7new_im},'{re:
212
      w2new.re, im: w2new.im});
213
214
     assign d0=complex_add_grow(c2,c0);
215
     assign d1=complex_add_grow(c3,c1);
     assign d2=complex_add_grow(c0,'{re: -c2.re, im: -c2.im});
217
     assign d3=complex_add_grow(c1, '{re: -c3.re, im: -c3.im});
218
     assign d4=complex_add_grow(c6,c4);
219
     assign d5=complex_add_grow(c7,c5);
220
     assign d6=complex_add_grow(c4, '{re: -c6.re, im: -c6.im});
221
     assign d7=complex_add_grow(c5, '{re: -c7.re, im: -c7.im});
222
223
224
     assign e0='{re: d0new_re, im: d0new_im};
225
     assign e1='{re: d1new_re, im: d1new_im};
     assign e2='{re: d2new_re, im: d2new_im};
227
     assign e3='{re: d3new_re, im: d3new_im};
228
     assign e4=complex_mult_grow('{re: d4new_re, im: d4new_im},'{re:
229
      w0new.re, im: w0new.im});
     assign e5=complex_mult_grow('{re: d5new_re, im: d5new_im},'{re:
230
      w1new.re, im: w1new.im});
     assign e6=complex_mult_grow('{re: d6new_re, im: d6new_im},'{re:
      w2new.re, im: w2new.im});
     assign e7=complex_mult_grow('{re: d7new_re, im: d7new_im},'{re:
      w3new.re, im: w3new.im});
234
     assign f0=complex_add_grow(e4,e0);
235
     assign f1=complex_add_grow(e5,e1);
236
    assign f2=complex_add_grow(e6,e2);
```

```
assign f3=complex_add_grow(e7,e3);
     assign f4=complex_add_grow(e0, '{re: -e4.re, im: -e4.im});
239
     assign f5=complex_add_grow(e1, '{re: -e5.re, im: -e5.im});
240
     assign f6=complex_add_grow(e2, '{re: -e6.re, im: -e6.im});
241
     assign f7=complex_add_grow(e3,'{re: -e7.re, im: -e7.im});
242
243
     assign y0='{re: -f0.re/8, im: f0.im/8};
244
     assign y1='{re: -f1.re/8, im: f1.im/8};
     assign y2='{re: -f2.re/8, im: f2.im/8};
246
     assign y3='{re: -f3.re/8, im: f3.im/8};
247
     assign y4='{re: -f4.re/8, im: f4.im/8};
248
    assign y5='{re: -f5.re/8, im: f5.im/8};
     assign y6='{re: -f6.re/8, im: f6.im/8};
250
     assign y7='{re: -f7.re/8, im: f7.im/8};
251
252
254
  endmodule
```

Listing 2: 8-Point IFFT Module in Verilog

#### 4.4 Test Bench

#### Verilog Testbench Code

```
1 module testbench_FFT;
    parameter integer width = 16;
    reg clk;
    reg reset;
    integer write_data;
    localparam SF = 2.0**(-12);
    reg signed [width-1:0] inputs[15:0];
    reg signed [width-1:0] realv[7:0];
10
    reg signed [width-1:0] imag[7:0];
11
12
    typedef struct {
      logic signed [width-1:0] re;
14
      logic signed [width-1:0] im;
    } complex_grow;
16
17
    complex_grow x0, x1, x2, x3, x4, x5, x6, x7;
18
    complex_grow y0, y1, y2, y3, y4, y5, y6, y7;
20
    // Instantiate the FFT module
21
    IFFT FFT_instance (
22
      .x0(x0),
23
      .x1(x1),
24
      .x2(x2),
```

```
.x3(x3),
26
27
       .x4(x4),
       .x5(x5),
28
       .x6(x6),
29
       .x7(x7),
30
       .clk(clk),
31
       .reset(reset),
32
       .y0(y0),
33
       .y1(y1),
34
       .y2(y2),
35
       .y3(y3),
36
37
       .y4(y4),
       .y5(y5),
38
      .y6(y6),
39
       .y7(y7)
40
    );
41
42
    // Generate clock signal
43
    always #5 clk = ~clk;
44
45
    initial begin
46
      $dumpfile("dump.vcd");
47
48
      $dumpvars(0);
49
50
    task open();
      $readmemh("input.txt", inputs);
52
53
      for (integer i = 0; i < 16; i = i + 2) begin
54
        realv[i/2] = inputs[i];
      end
56
57
      for (integer i = 1; i < 16; i = i + 2) begin
         imag[i/2] = inputs[i];
58
      end
59
60
      // Assign input values to inputs of FFT module
61
      x0 = '{re: realv[0], im: imag[0]};
62
      x1 = '{re: realv[1], im: imag[1]};
63
      x2 = '{re: realv[2], im: imag[2]};
64
      x3 = '{re: realv[3], im: imag[3]};
      x4 = '{re: realv[4], im: imag[4]};
66
      x5 = '{re: realv[5], im: imag[5]};
67
      x6 = '{re: realv[6], im: imag[6]};
68
      x7 = '\{re: realv[7], im: imag[7]\};
69
    endtask
70
71
    task drive_reset();
72
      $display("Driving the reset");
73
      clk = 1, b0;
74
      @(negedge clk) reset = 0;
75
      @(negedge clk) reset = 1;
76
      @(negedge clk) reset = 0;
77
    endtask
78
```

```
task display_output();
80
       $display("Output values :- \n");
81
       for (integer i = 0; i < 8; i++) begin
82
         display("y\%0d = \%f + \%f j", i, $itor(y[i].re) * SF, $itor(y[i])
      ].im) * SF);
       end
84
85
       $fwrite(write_data, "Output values :- \n");
       for (integer i = 0; i < 8; i++) begin
87
         fwrite(write_data, "y\%0d = \%f + \%f j\n", i, $itor(y[i].re) *
88
      SF, $itor(y[i].im) * SF);
89
       end
     endtask
90
91
     initial begin
92
       write_data = $fopen("output_tracker.txt", "w");
93
       drive_reset();
94
       open();
95
       repeat (30) @(negedge clk);
96
       display_output();
97
       $fclose(write_data);
98
       $finish;
99
100
     end
101 endmodule
```

Listing 3: IFFT Testbench

### 4.5 Results and graphs

```
# KERNEL: y0= 1.779785 + -0.219482 j

# KERNEL: y1= 0.219238 + -0.280518 j

# KERNEL: y2= 0.573242 + 0.572998 j

# KERNEL: y3= 0.426758 + -0.072998 j

# KERNEL: y4= -0.280273 + -0.780518 j

# KERNEL: y5= 0.280273 + 1.280518 j

# KERNEL: y6= -0.072754 + 0.427002 j

# KERNEL: y7= -0.926270 + -0.927002 j
```

Figure 5: Output of the DIF-FFT

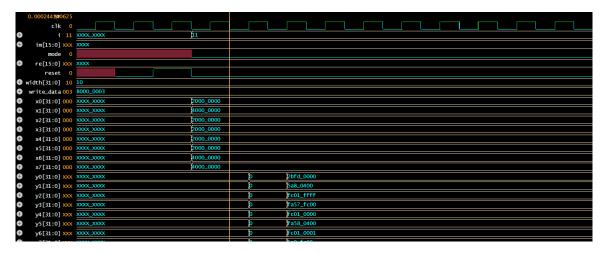


Figure 6: Waveform diagram of the DIF-FFT

## 5 Observations

- The FFT and IFFT are mathematically inverse operations of each other. Applying the FFT followed by the IFFT should return the original sequence, up to some scaling factor.
- In many implementations, the IFFT algorithm is similar to the FFT algorithm, but it works in reverse order and may require additional normalization.
- Both results of DIT and DIF are same.
- Based on our requirements we will use DIT or DIF.
- By doing this project learned about configuring complex numbers and adding, multiplying them.