EE5516 : VLSI Architectures for Signal Processing and Machine Learning

INDIAN INSTITUTE OF TECHNOLOGY
PALAKKAD

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,
- 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,
- 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:

- $x[n]$ is the reconstructed time-domain signal,
- $X[k]$ is the frequency domain signal (obtained from the DFT),
- 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 sub-sequences.
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 $X = [X_0, X_1, \ldots, X_{N-1}]$.
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 $x = [x_0, x_1, \ldots, x_{N-1}]$, representing the inverse FFT of X.

3.6 Architecture

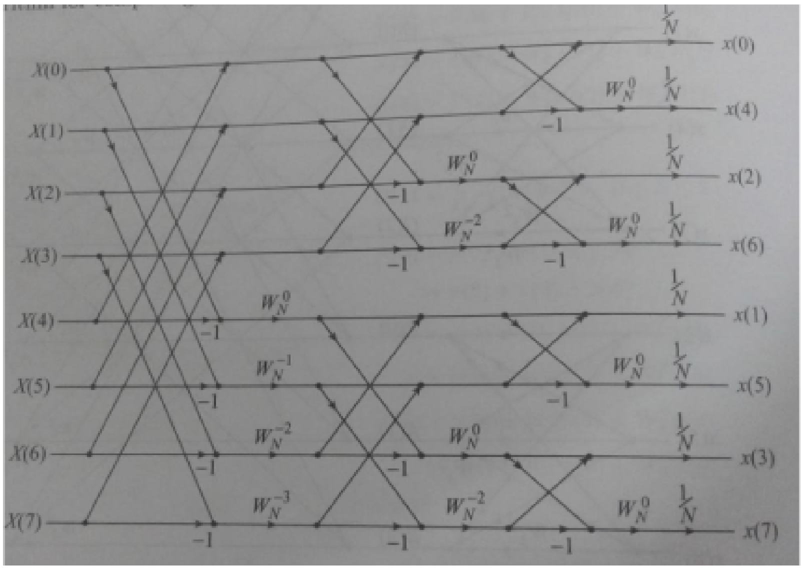

Figure 1: Architecture of DIT IFFT

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 fixed-point representation
  parameter integer F_width = 15; // fractional part width of fixed-point representation
  parameter QP = 15;


  ;
  typedef struct {
    logic signed [width-1:0] re;
    logic signed [width-1:0] im;
  } complex_grow;



  parameter complex_grow w0new='{re: 16'h7fff, im: 16'h0000};
  parameter complex_grow w1new='{re: 16'h5a82, im: 16'ha57e};
  parameter complex_grow w2new='{re: 16'h0000, im: 16'h8000};
  parameter complex_grow w3new='{re: 16'ha57e, im: 16'ha57e};


  typedef struct {

    logic signed [31:0] re;
    logic signed [2*(width)-1:0] im;

  } complex_temp_grow;


  function complex_grow complex_mult_grow(complex_grow a, complex_grow b);


    complex_temp_grow full;
    complex_grow full1;
    full.re = ($signed(a.re) * $signed(b.re)) - ($signed(a.im) * $signed(b.im));
```

```
    full.im = ($signed(a.re) * $signed(b.im)) + ($signed(a.im)
        * $signed(b.re));

    full1.re=full.re[30:15];
    full1.im=full.im[30:15];

    return full1;
  endfunction


  function complex_grow complex_add_grow(complex_grow a,
     complex_grow b);
    complex_grow result;
    result.re = a.re + b.re;
    result.im = a.im + b.im;
    return result;
  endfunction

  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;
  reg[width-1:0] a00,a01,a10,a11,a20,a21,a30,a31,a40,a41,a50,
     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,
     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;

  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;
  wire complex_grow  e0,e1,e2,e3,e4,e5,e6,e7;
  wire complex_grow  f0,f1,f2,f3,f4,f5,f6,f7;
  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;
```

```
always@(posedge clk)
  begin
    if(reset)
      begin

        a70<=0;
        a60<=0;
        a50<=0;
        a40<=0;
        a30<=0;
        a20<=0;
        a10<=0;
        a00<=0;
        a71<=0;
        a61<=0;
        a51<=0;
        a41<=0;
        a31<=0;
        a21<=0;
        a11<=0;
        a01<=0;

      end
    else


          begin

            a00<=x0.re;
            a10<=x4.re;
            a20<=x2.re;
            a30<=x6.re;
            a40<=x1.re;
            a50<=x5.re;
            a60<=x3.re;
            a70<=x7.re;
            a01<=x0.im;
            a11<=x4.im;
            a21<=x2.im;
            a31<=x6.im;
            a41<=x1.im;
            a51<=x5.im;
            a61<=x3.im;
            a71<=x7.im;
          end

  end
```

```
assign a0new_re = {{3{a00[width-1]}}, a00[15:3]};
assign a1new_re = {{3{a10[width-1]}}, a10[15:3]};
assign a2new_re = {{3{a20[width-1]}}, a20[15:3]};
assign a3new_re = {{3{a30[width-1]}}, a30[15:3]};
assign a4new_re = {{3{a40[width-1]}}, a40[15:3]};
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]};

assign a0new_im = {{3{a01[width-1]}}, a01[15:3]};
assign a1new_im = {{3{a11[width-1]}}, a11[15:3]};
assign a2new_im = {{3{a21[width-1]}}, a21[15:3]};
assign a3new_im = {{3{a31[width-1]}}, a31[15:3]};
assign a4new_im = {{3{a41[width-1]}}, a41[15:3]};
assign a5new_im = {{3{a51[width-1]}}, a51[15:3]};
assign a6new_im = {{3{a61[width-1]}}, a61[15:3]};
assign a7new_im = {{3{a71[width-1]}}, a71[15:3]};




assign e0 = complex_add_grow('{re: d0new_re, im: d0new_im},
   '{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},
   '{re: d3new_re, im: d3new_im});
assign e3 = complex_add_grow('{re: d2new_re, im: d2new_im},
   '{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},
   '{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)

  begin
    b0new_re <= {{4{b0.re[width-1]}}, b0.re};
    b1new_re <= {{4{b1.re[width-1]}}, b1.re};
    b2new_re <= {{4{b2.re[width-1]}}, b2.re};
    b3new_re <= {{4{b3.re[width-1]}}, b3.re};
    b4new_re <= {{4{b4.re[width-1]}}, b4.re};
```

```
      b5new_re  <=  {{4{b5.re[width-1]}}, b5.re};
      b6new_re  <=  {{4{b6.re[width-1]}}, b6.re};
      b7new_re  <=  {{4{b7.re[width-1]}}, b7.re};

      b0new_im  <=  {{4{b0.im[width-1]}}, b0.im};
      b1new_im  <=  {{4{b1.im[width-1]}}, b1.im};
      b2new_im  <=  {{4{b2.im[width-1]}}, b2.im};
      b3new_im  <=  {{4{b3.im[width-1]}}, b3.im};
      b4new_im  <=  {{4{b4.im[width-1]}}, b4.im};
      b5new_im  <=  {{4{b5.im[width-1]}}, b5.im};
      b6new_im  <=  {{4{b6.im[width-1]}}, b6.im};
      b7new_im  <=  {{4{b7.im[width-1]}}, b7.im};

      d0new_re  <=  d0.re;
      d1new_re  <=  d1.re;
      d2new_re  <=d2.re;
      d3new_re  <=  d3.re;
      d4new_re  <=  d4.re;
      d5new_re  <=  d5.re;
      d6new_re  <=  d6.re;
      d7new_re  <=  d7.re;

      d0new_im  <=d0.im;
      d1new_im  <=d1.im;
      d2new_im  <=d2.im;
      d3new_im  <=d3.im;
      d4new_im  <=d4.im;
      d5new_im  <=d5.im;
      d6new_im  <=d6.im;
      d7new_im  <=d7.im;


    end



  assign c0='{re: b0new_re, im: b0new_im};

  assign c1='{re: b1new_re, im: b1new_im};
  assign c2=complex_mult_grow('{re: b2new_re, im: b2new_im},'{re: w0new.re, im: w0new.im});
```

```
    assign c3=complex_mult_grow('{re: b3new_re, im: b3new_im},'{re: w2new.re, im: w2new.im});
    assign c4='{re: b4new_re, im: b4new_im};
    assign c5='{re: b5new_re, im: b5new_im};
    assign c6=complex_mult_grow('{re: b6new_re, im: b6new_im},'{re: w0new.re, im: w0new.im});
    assign c7=complex_mult_grow('{re: b7new_re, im: b7new_im},'{re: w2new.re, im: w2new.im});


    assign d0=complex_add_grow(c2,c0);
    assign d1=complex_add_grow(c3,c1);
    assign d2=complex_add_grow(c0,'{re: -c2.re, im: -c2.im});
    assign d3=complex_add_grow(c1,'{re: -c3.re, im: -c3.im});
    assign d4=complex_add_grow(c6,c4);
    assign d5=complex_add_grow(c7,c5);
    assign d6=complex_add_grow(c4,'{re: -c6.re, im: -c6.im});
    assign d7=complex_add_grow(c5,'{re: -c7.re, im: -c7.im});


    assign b0='{re: a0new_re, im: a0new_im};
    assign b1='{re: a1new_re, im: a1new_im};
    assign b2='{re: a2new_re, im: a2new_im};
    assign b3='{re: a3new_re, im: a3new_im};
    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},'{re: w1new.re, im: -w1new.im});
    assign b6=complex_mult_grow('{re: a6new_re, im: a6new_im},'{re: w2new.re, im: -w2new.im});
    assign b7=complex_mult_grow('{re: a7new_re, im: a7new_im},'{re: w3new.re, im: -w3new.im});


    assign f0=complex_add_grow(e4,e0);
    assign f1=complex_add_grow(e5,e1);
    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});
    assign f5=complex_add_grow(e1,'{re: -e5.re, im: -e5.im});
    assign f6=complex_add_grow(e2,'{re: -e6.re, im: -e6.im});
    assign f7=complex_add_grow(e3,'{re: -e7.re, im: -e7.im});

    assign y0='{re: -f0.re/8, im: f0.im/8};
    assign y1='{re: -f1.re/8, im: f1.im/8};
    assign y2='{re: -f2.re/8, im: f2.im/8};
    assign y3='{re: -f3.re/8, im: f3.im/8};
```

```
  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};




endmodule
```

3.8 Test Bench

Verilog Testbench Code

```
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];
  reg signed [width-1:0] imag[7:0];

  typedef struct {
    logic signed [width-1:0] re;
    logic signed [width-1:0] im;
  } complex_grow;

  complex_grow x0, x1, x2, x3, x4, x5, x6, x7;
  complex_grow y0, y1, y2, y3, y4, y5, y6, y7;

  // Instantiate the FFT module
  IFFT FFT_instance (
    .x0(x0),
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .x4(x4),
    .x5(x5),
    .x6(x6),
    .x7(x7),
    .clk(clk),
    .reset(reset),
    .y0(y0),
    .y1(y1),
    .y2(y2),
    .y3(y3),
```

```
    .y4(y4),
    .y5(y5),
    .y6(y6),
    .y7(y7)
  );

  // Generate clock signal
  always #5 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end

  task open();
    $readmemh("input.txt", inputs);

    for (integer i = 0; i < 16; i = i + 2) begin
      realv[i/2] = inputs[i];
    end
    for (integer i = 1; i < 16; i = i + 2) begin
      imag[i/2] = inputs[i];
    end

    // Assign input values to inputs of FFT module
    x0 = '{re: realv[0], im: imag[0]};
    x1 = '{re: realv[1], im: imag[1]};
    x2 = '{re: realv[2], im: imag[2]};
    x3 = '{re: realv[3], im: imag[3]};
    x4 = '{re: realv[4], im: imag[4]};
    x5 = '{re: realv[5], im: imag[5]};
    x6 = '{re: realv[6], im: imag[6]};
    x7 = '{re: realv[7], im: imag[7]};
  endtask

  task drive_reset();
    $display("Driving the reset");
    clk = 1'b0;
    @(negedge clk) reset = 0;
    @(negedge clk) reset = 1;
    @(negedge clk) reset = 0;
  endtask

  task display_output();
    $display("Output values :- \n");
    for (integer i = 0; i < 8; i++) begin
      $display("y%0d = %f + %f j", i, $itor(y[i].re) * SF, $itor(y[i].im) * SF);
    end

    $fwrite(write_data, "Output values :- \n");
    for (integer i = 0; i < 8; i++) begin
      $fwrite(write_data, "y%0d = %f + %f j\n", i, $itor(y[i].re) * SF, $itor(y[i].im) * SF);
```

```
    end
  endtask

  initial begin
    write_data = $fopen("output_tracker.txt", "w");
    drive_reset();
    open();
    repeat (30) @(negedge clk);
    display_output();
    $fclose(write_data);
    $finish;
  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

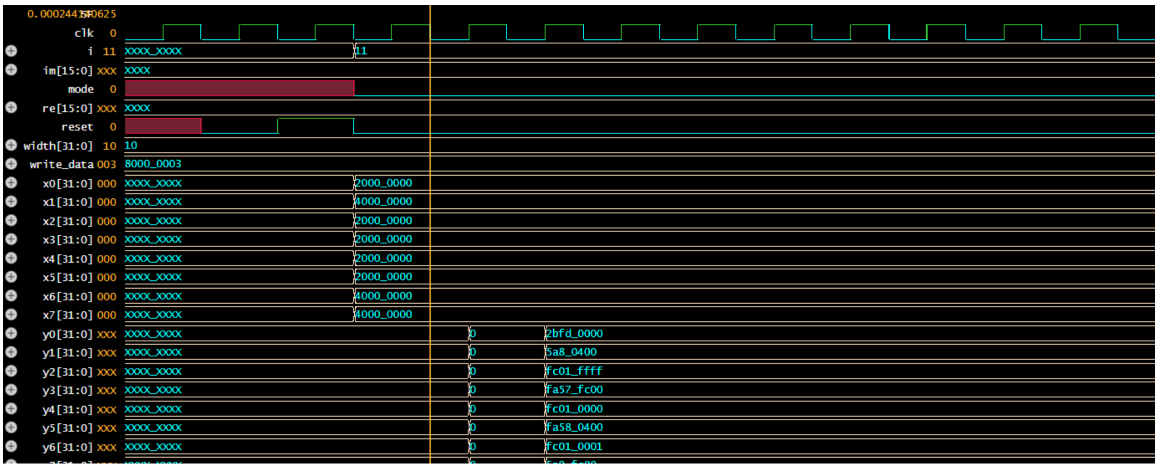

Figure 3: Waveform diagram of the DIT-FFT

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 :

- Normalize the output by dividing by N

4.2 Architecture

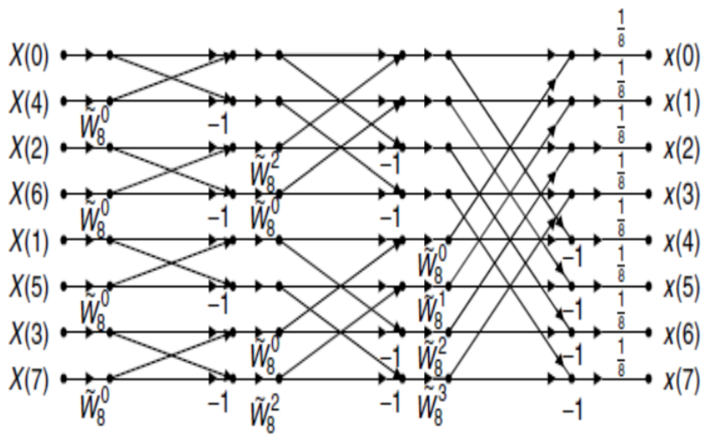

Figure 4: Output of the DIF-IFFT

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;


  ;
  typedef struct {
    logic signed [width-1:0] re;
    logic signed [width-1:0] im;
  } complex_grow;



  parameter complex_grow w0new='{re: 16'h7fff, im: 16'h0000};
  parameter complex_grow w1new='{re: 16'h5a82, im: 16'ha57e};
  parameter complex_grow w2new='{re: 16'h0000, im: 16'h8000};
  parameter complex_grow w3new='{re: 16'ha57e, im: 16'ha57e};


  typedef struct {

    logic signed [31:0] re;
    logic signed [2*(width)-1:0] im;

  } complex_temp_grow;


  function complex_grow complex_mult_grow(complex_grow a, complex_grow b);


    complex_temp_grow full;
    complex_grow full1;
    full.re = ($signed(a.re) * $signed(b.re)) - ($signed(a.im) * $signed(b.im));
    full.im = ($signed(a.re) * $signed(b.im)) + ($signed(a.im) * $signed(b.re));

    full1.re=full.re[30:15];
    full1.im=full.im[30:15];

    return full1;
  endfunction
```

```


function complex_grow complex_add_grow(complex_grow a, complex_grow b);
  complex_grow result;
  result.re = a.re + b.re;
  result.im = a.im + b.im;
  return result;
endfunction


input complex_grow x0,x1,x2,x3,x4,x5,x6,x7;
input reset;
output complex_grow y0,y1,y2,y3,y4,y5,y6,y7;
reg[width-1:0] a00,a01,a10,a11,a20,a21,a30,a31,a40,a41,a50,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,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;

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;
wire complex_grow  e0,e1,e2,e3,e4,e5,e6,e7;
wire complex_grow  f0,f1,f2,f3,f4,f5,f6,f7;
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;



always@(posedge clk)
  begin
    if(reset)
      begin

        a70<=0;
        a60<=0;
        a50<=0;
        a40<=0;
        a30<=0;
        a20<=0;
        a10<=0;
        a00<=0;
        a71<=0;
        a61<=0;
```

```
            a51<=0;
            a41<=0;
            a31<=0;
            a21<=0;
            a11<=0;
            a01<=0;

          end
        else

              begin

                a00<=x0.re;
                a10<=x4.re;
                a20<=x2.re;
                a30<=x6.re;
                a40<=x1.re;
                a50<=x5.re;
                a60<=x3.re;
                a70<=x7.re;
                a01<=x0.im;
                a11<=x4.im;
                a21<=x2.im;
                a31<=x6.im;
                a41<=x1.im;
                a51<=x5.im;
                a61<=x3.im;
                a71<=x7.im;
              end


      end

    assign a0new_re = {{3{a00[width-1]}}, a00[15:3]};
    assign a1new_re = {{3{a10[width-1]}}, a10[15:3]};
    assign a2new_re = {{3{a20[width-1]}}, a20[15:3]};
    assign a3new_re = {{3{a30[width-1]}}, a30[15:3]};
    assign a4new_re = {{3{a40[width-1]}}, a40[15:3]};
    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]};

    assign a0new_im = {{3{a01[width-1]}}, a01[15:3]};
    assign a1new_im = {{3{a11[width-1]}}, a11[15:3]};
    assign a2new_im = {{3{a21[width-1]}}, a21[15:3]};
    assign a3new_im = {{3{a31[width-1]}}, a31[15:3]};
    assign a4new_im = {{3{a41[width-1]}}, a41[15:3]};
    assign a5new_im = {{3{a51[width-1]}}, a51[15:3]};
    assign a6new_im = {{3{a61[width-1]}}, a61[15:3]};
    assign a7new_im = {{3{a71[width-1]}}, a71[15:3]};
```

```
assign b0=complex_add_grow('{re: a0new_re, im: a0new_im},'{re:
 a1new_re, im: a1new_im});
assign b1=complex_add_grow('{re: a0new_re, im: a0new_im},'{re: -
 a1new_re, im: -a1new_im});
assign b2=complex_add_grow('{re: a2new_re, im: a2new_im},'{re:
 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)

  begin
    b0new_re <= {{4{b0.re[width-1]}}, b0.re};
    b1new_re <= {{4{b1.re[width-1]}}, b1.re};
    b2new_re <= {{4{b2.re[width-1]}}, b2.re};
    b3new_re <= {{4{b3.re[width-1]}}, b3.re};
    b4new_re <= {{4{b4.re[width-1]}}, b4.re};
    b5new_re <= {{4{b5.re[width-1]}}, b5.re};
    b6new_re <= {{4{b6.re[width-1]}}, b6.re};
    b7new_re <= {{4{b7.re[width-1]}}, b7.re};

    b0new_im <= {{4{b0.im[width-1]}}, b0.im};
    b1new_im <= {{4{b1.im[width-1]}}, b1.im};
    b2new_im <= {{4{b2.im[width-1]}}, b2.im};
    b3new_im <= {{4{b3.im[width-1]}}, b3.im};
    b4new_im <= {{4{b4.im[width-1]}}, b4.im};
    b5new_im <= {{4{b5.im[width-1]}}, b5.im};
    b6new_im <= {{4{b6.im[width-1]}}, b6.im};
    b7new_im <= {{4{b7.im[width-1]}}, b7.im};

    d0new_re <= d0.re;
    d1new_re <= d1.re;
    d2new_re <=d2.re;
    d3new_re <= d3.re;
    d4new_re <= d4.re;
    d5new_re <= d5.re;
    d6new_re <= d6.re;
    d7new_re <= d7.re;

    d0new_im <=d0.im;
    d1new_im <=d1.im;
    d2new_im <=d2.im;
    d3new_im <=d3.im;
    d4new_im <=d4.im;
    d5new_im <=d5.im;
    d6new_im <=d6.im;
```

```
      d7new_im  <=d7.im;


    end



  assign c0='{re: b0new_re, im: b0new_im};

  assign c1='{re: b1new_re, im: b1new_im};
  assign c2=complex_mult_grow('{re: b2new_re, im: b2new_im},'{re:
   w0new.re, im: w0new.im});





  assign c3=complex_mult_grow('{re: b3new_re, im: b3new_im},'{re:
   w2new.re, im: w2new.im});
  assign c4='{re: b4new_re, im: b4new_im};
  assign c5='{re: b5new_re, im: b5new_im};
  assign c6=complex_mult_grow('{re: b6new_re, im: b6new_im},'{re:
   w0new.re, im: w0new.im});
  assign c7=complex_mult_grow('{re: b7new_re, im: b7new_im},'{re:
   w2new.re, im: w2new.im});


  assign d0=complex_add_grow(c2,c0);
  assign d1=complex_add_grow(c3,c1);
  assign d2=complex_add_grow(c0,'{re: -c2.re, im: -c2.im});
  assign d3=complex_add_grow(c1,'{re: -c3.re, im: -c3.im});
  assign d4=complex_add_grow(c6,c4);
  assign d5=complex_add_grow(c7,c5);
  assign d6=complex_add_grow(c4,'{re: -c6.re, im: -c6.im});
  assign d7=complex_add_grow(c5,'{re: -c7.re, im: -c7.im});


  assign e0='{re: d0new_re, im: d0new_im};
  assign e1='{re: d1new_re, im: d1new_im};
  assign e2='{re: d2new_re, im: d2new_im};
  assign e3='{re: d3new_re, im: d3new_im};
  assign e4=complex_mult_grow('{re: d4new_re, im: d4new_im},'{re:
   w0new.re, im: w0new.im});
  assign e5=complex_mult_grow('{re: d5new_re, im: d5new_im},'{re:
   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});


  assign f0=complex_add_grow(e4,e0);
  assign f1=complex_add_grow(e5,e1);
  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});
  assign f5=complex_add_grow(e1,'{re: -e5.re, im: -e5.im});
  assign f6=complex_add_grow(e2,'{re: -e6.re, im: -e6.im});
  assign f7=complex_add_grow(e3,'{re: -e7.re, im: -e7.im});

  assign y0='{re: -f0.re/8, im: f0.im/8};
  assign y1='{re: -f1.re/8, im: f1.im/8};
  assign y2='{re: -f2.re/8, im: f2.im/8};
  assign y3='{re: -f3.re/8, im: f3.im/8};
  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};




endmodule
```

Listing 2: 8-Point IFFT Module in Verilog

4.4 Test Bench

Verilog Testbench Code

```
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];
  reg signed [width-1:0] imag[7:0];

  typedef struct {
    logic signed [width-1:0] re;
    logic signed [width-1:0] im;
  } complex_grow;

  complex_grow x0, x1, x2, x3, x4, x5, x6, x7;
  complex_grow y0, y1, y2, y3, y4, y5, y6, y7;

  // Instantiate the FFT module
  IFFT FFT_instance (
    .x0(x0),
    .x1(x1),
    .x2(x2),
```

```
    .x3(x3),
    .x4(x4),
    .x5(x5),
    .x6(x6),
    .x7(x7),
    .clk(clk),
    .reset(reset),
    .y0(y0),
    .y1(y1),
    .y2(y2),
    .y3(y3),
    .y4(y4),
    .y5(y5),
    .y6(y6),
    .y7(y7)
  );

  // Generate clock signal
  always #5 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end

  task open();
    $readmemh("input.txt", inputs);

    for (integer i = 0; i < 16; i = i + 2) begin
      realv[i/2] = inputs[i];
    end
    for (integer i = 1; i < 16; i = i + 2) begin
      imag[i/2] = inputs[i];
    end

    // Assign input values to inputs of FFT module
    x0 = '{re: realv[0], im: imag[0]};
    x1 = '{re: realv[1], im: imag[1]};
    x2 = '{re: realv[2], im: imag[2]};
    x3 = '{re: realv[3], im: imag[3]};
    x4 = '{re: realv[4], im: imag[4]};
    x5 = '{re: realv[5], im: imag[5]};
    x6 = '{re: realv[6], im: imag[6]};
    x7 = '{re: realv[7], im: imag[7]};
  endtask

  task drive_reset();
    $display("Driving the reset");
    clk = 1'b0;
    @(negedge clk) reset = 0;
    @(negedge clk) reset = 1;
    @(negedge clk) reset = 0;
  endtask

```

```
  task display_output();
    $display("Output values :- \n");
    for (integer i = 0; i < 8; i++) begin
      $display("y%0d = %f + %f j", i, $itor(y[i].re) * SF, $itor(y[i].im) * SF);
    end

    $fwrite(write_data, "Output values :- \n");
    for (integer i = 0; i < 8; i++) begin
      $fwrite(write_data, "y%0d = %f + %f j\n", i, $itor(y[i].re) * SF, $itor(y[i].im) * SF);
    end
  endtask

  initial begin
    write_data = $fopen("output_tracker.txt", "w");
    drive_reset();
    open();
    repeat (30) @(negedge clk);
    display_output();
    $fclose(write_data);
    $finish;
  end
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

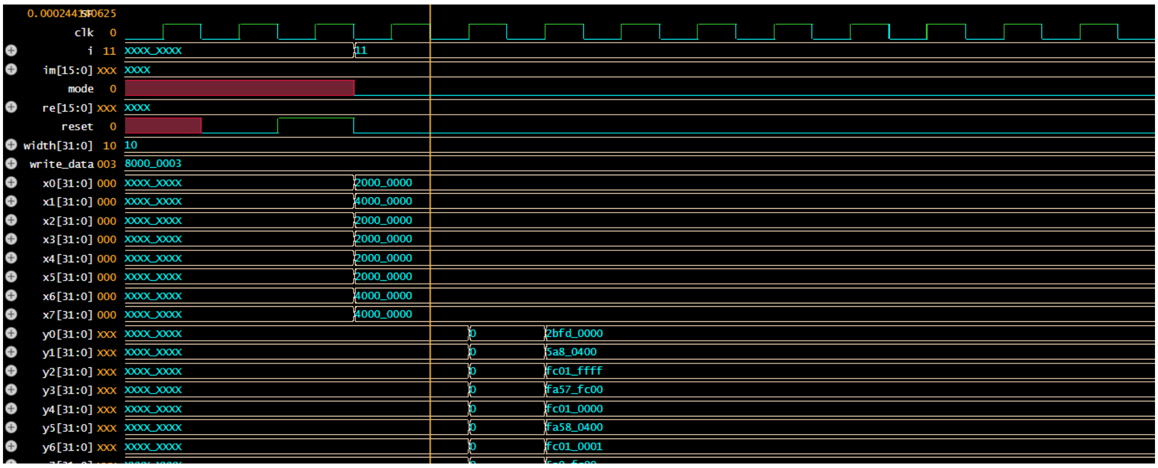

Figure 6: Waveform diagram of the DIF-FFT

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.